Theoretical and Applied Informatics
ISSN 1896–5334
Vol.23 (2011), no. 2
pp. 127–145
DOI: 10.2478/v10179-011-0009-5

Computational Aspects of GPU-accelerated Sparse Matrix-Vector Multiplication for Solving Markov Models

BEATA BYLINA, JAROSŁAW BYLINA, MAREK KARWACKI

Institute of Mathematics
Marie Curie-Skłodowska University
Pl. M. Curie-Skłodowskiej 5, 20-031 Lublin, Poland
beata.bylina@umcs.pl jaroslaw.bylina@umcs.pl marek.karwacki@gmail.com

Received 28 May 2011, Revised 20 June 2011, Accepted 29 June 2011

Abstract: In this article we investigate some computational aspects of GPU-accelerated matrix-vector multiplication where matrix is sparse. Particularly, we deal with sparse matrices appearing in modelling with Markovian queuing models. The model we use for research is a Markovian queuing model of a wireless device. This model describes the device's behavior during possible channel occupation by other devices.

We study the efficiency of multiplication of a sparse matrix by a dense vector with the use of an appropriate, ready-to-use GPU-accelerated mathematical library, namely CUSP. For the CUSP library we discuss data structures and their impact on the CUDA platform for the fine-grained parallel architecture of the GPU. Our aim is to find the best format for storing a sparse matrix for GPU-computation (especially one associated with the Markovian model of a wireless device).

We compare the time, the performance and the speed-up for the card NVIDIA Tesla C2050 (with ECC ON). For unstructured matrices (as our Markovian matrices), we observe speed-ups (in respect to CPU-only computations) of over 8 times.

Keywords: Markovian models, wireless network models, GPU, matrix-vector multiplication, sparse matrices

1. Introduction

Markov chains are a tool for modelling various complex systems, among others computer systems and networks. Recently they are commonly used for modelling wireless networks [3, 4, 10]. One of the problems appearing while using Markov chains to model complex systems making difficult the full utilization of the approach is a very large size of the model and thereby implied a long computation time (and memory consumption).

Modern graphic cards (GPU — graphics processing units) enable significant speeding up computations in scientific and engineering applications. They consist of many separate and independent (to some extent) computing cores. Originally used to image processing, now they are suitable to general purpose computations.

Within the domain of performance analysis there has been some work on the subject of GPU-enabled computations connected to modelling. For instance, in [1] authors show an implementation for LTL (linear temporal logic) model checking which accelerated computations on many-core GPU platforms.

To improve the performance we can use GPUs. In [6] some GPU-enabled steady-state parallel solver is described. The solver uses NVIDIA's Compute Unified Device Architecture (CUDA) to perform calculations on a graphics processing unit (GPU). To find steady-state (stationary) probabilities, authors used the most easily parallelisable iterative solution technique, namely Jacobi method.

Here we consider multiplication of a sparse matrix by a dense vector (SpMV). The operation SpMV is a main component for many various numerical algorithms [9] used both for computing stationary probabilities (e.g. by algorithms GMRES and CGS) and for transitional probabilities (e.g. by uniformization [8] and Krylov-based methods).

The operation of dense matrix-vector multiplication easily maps to many threads of GPU. However, multiplying a sparse matrix by a (dense) vector is a serious problem because coalescent access to the matrix elements is difficult — the elements are significantly dispersed. That is why it is very important to choose an adequate format of sparse matrix storage for GPU computations.

The aim of this paper is to present research on the efficiency of multiplying a sparse matrix by a dense vector with the use of GPU to gain high performance at low cost. We examine sparse matrices arising in different problems, particularly transition rate matrices describing Markovian models of wireless networks. The considered matrices span quite a wide spectrum: we investigated different data representations for such matrices and the influence of the representation on GPU-enabled implementations. We used an appropriate, ready-to-use GPU-accelerated mathematical library: CUSP.

The paper is structured as follows. Section 2 provides a brief overview of a background theory, a model of a wireless network device and a GPU architecture. Section 3 describes different data representations for sparse matrices and their usability for implementing the multiplication of a sparse matrix by a dense vector with the use of GPU. Section 4 considers problems related to SpMV with the use of GPU. Sections 5 presents some numerical results. Section 6 provides some concluding remarks.

2. Background

2.1. Numerical Solutions of Markov Chains

A CTMC (Continuous-Time Markov Chain) may be represented by a set of states and a transition rate matrix $\mathbf{Q}$ containing state transition rates as coefficients and can be analyzed by using probabilistic model checking.

While analyzing steady states (that is: independent of time, or: after a very long run) of Markov chains, we obtain a linear equation system:

$$\mathbf{Q}^T\mathbf{x} = \mathbf{0}, \qquad \mathbf{x} \geq \mathbf{0}, \qquad ||\mathbf{x}||_1 = 1, \tag{1}$$

where $\mathbf{Q}$ is a transition rate matrix and $\mathbf{x}$ is an unknown vector of stationary states probabilities. The matrix $\mathbf{Q}$ is a square one of size $n \times n$ (n being the number of CTMC's states), usually a big one, of rank $n-1$, sparse, with weakly dominant diagonal, singular, ill-conditioned. These traits of $\mathbf{Q}$ cause the need of a special treatment of the system (1). For solving the system (1), one generally uses direct, iterative, projection and decomposition methods [9]. The main computational operation for most of these methods is multiplying a (sparse) matrix by a (dense) vector.

Quite a similar computational situation takes place in transitional states analysis. Transitional states probabilities are probabilities dependent on time — that is, we search $\mathbf{x}(t)$. Now, we obtain another equation — an ordinal differential one with initial conditions (a Chapman-Kolmogorov system):

$$\mathbf{Q}^T\mathbf{x}(t) = \frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t}, \qquad \mathbf{x}(0) = \mathbf{x}_0. \tag{2}$$

For this equation we have an analytical solution:

$$\mathbf{x}(t) = e^{\mathbf{Q}t}\mathbf{x}_0. \tag{3}$$

Of course, to find $\mathbf{x}(t)$ we can compute right-hand expression of (3) or we can numerically solve the ODE (2). In the former case the problem is to compute $e^{\mathbf{Q}t}$ which can be expressed (using the Taylor formula) as an infinite series:

$$e^{\mathbf{Q}t} = \sum_{k=0}^{\infty} \frac{(\mathbf{Q}t)^k}{k!}. \tag{4}$$

The task of computing a power of a matrix is not a numerically stable one. So, we have to use other approaches as uniformization method [8], general ODE solvers and Krylov-based methods. Just like for solving (1), the most time-consuming part of the uniformization and Krylov-based methods (and some other solvers, too) is the matrix-vector product.

That is why it is very important to make computation of the matrix-vector products efficient.

2.2. WLAN Node Model

As an example of the use of Markov models, let us consider a following problem. We consider the performance of a wireless network where the access to the transmission medium follows well-known rivalisation procedure: DCF. The DCF (Distributed Coordination Function) mechanism is a part of 802.11 standard [5]. When a new packet is going to be sent, the medium is checked if it is busy for a fixed period of time (distributed interframe space, DIFS). If the channel is free, the packet is transmitted. However, if the channel happens to be busy, a collision control mechanism is employed. It consists in three steps:

- First, a backoff time is randomly chosen as an integer number of fixed time intervals σ, from $\langle 0; W-1\rangle$ (where W is a minimal value of the contention window).
- Second, the device waits the chosen time (freezing when the channel is busy).
- Third, if the channel is free after this countdown, the packet is sent. If the channel is not free, the whole procedure is repeated but the range for randomized backoff time is $\langle 0; 2 \cdot W - 1\rangle$. If the transmission also fails this time, the drawing range becomes $\langle 0; 2 \cdot 2 \cdot W - 1\rangle$ and so on, up to $\langle 0; 2^m \cdot W - 1\rangle$ (that is: $2^m \cdot W$ is a maximal value of the contention window).

Markovian queuing model [4] of such a mechanism is presented in Figure 1. The state of the model is described with four integers (c, k, f, s):

- $c \in \langle 0; C\rangle$ is the current number of packets in the system,
- $k \in \langle 0; m\rangle$ is the number of failed transmission attempts,
- $f \in \langle 0; 2^m \cdot W - 1\rangle$ is the current number of time slots left to the moment of the next transmission attempt;
- $s \in \{0; 1\}$ is a flag equal to 1 if and only if the current packet is being sent.

Besides C (maximum capacity of the system), m and W, the model has other parameters:

- λ is the rate of the new packet appearance in the device;
- μ is the rate of packets' transmission;
- p is the the collision probability;
- η is the rate of the transition between slots f and $f-1$.

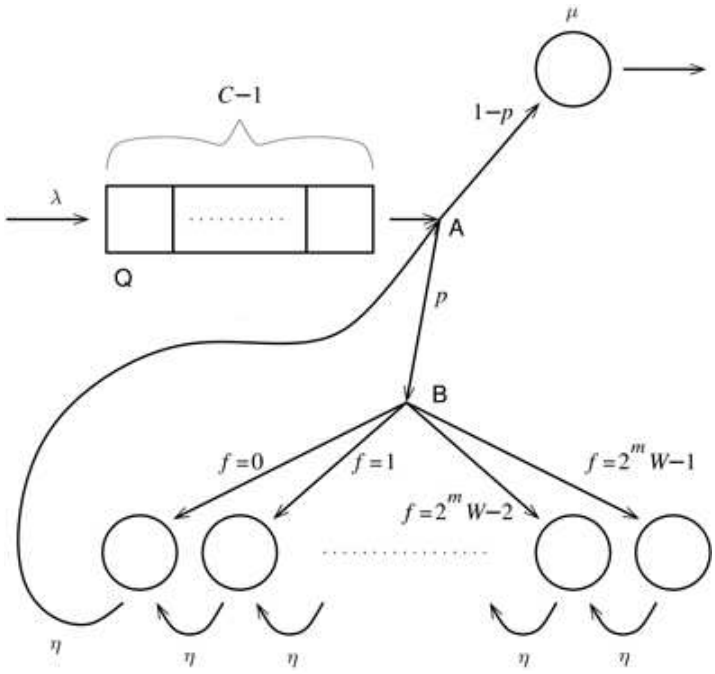

Fig. 1. The Markovian queuing model of the DCF mechanism

2.3. GPU and CUDA Architecture

Graphics Processing Units (GPUs) have recently been used for many applications beyond graphics, introducing the term *general-purpose computation on graphics processors* (GPGPU), owing to (among others) the CUDA architecture (*Compute Unified Device Architecture*) [7] prepared by NVIDIA.

Graphic processing units are manycore computing systems, being able to deal with thousands threads running in parallel. In contrast to CPUs, GPUs have a relatively higher effective memory clock in comparison to its computational core clock. The operations performed on each data element are mutually independent, and can be efficiently computed in parallel using many processors on the GPUs.

One of the major challenges in developing GPGPU algorithms is to create techniques which fully use pipelining, many cores and high memory bandwidth. It is not an easy task to build an efficient algorithm which uses all the GPU's features. Therefore, we propose using ready-to-use libraries — what means that problems of pipelining, memory access and block size do not involve us directly.

At present (June 2011) there are not many publicly available libraries providing SpMV on GPU. In this paper we choose for benchmarking CUSP [14] supporting multiple sparse matrices storage formats.

The library developed by the authors of [2] is for general sparse linear algebra problems, also with some graph algorithms, all implemented in C++ for CUDA. The other libraries we looked into are CUDPP [13], CUSPARSE [15] and OpenNL [12]. Unfortunately, all of them support SpMV only in CSR format. We have not seen any performance advantage over CUSP while making simple benchmarks of these different CSR SpMV implementations.

3. Sparse Matrix Storage Formats

There are many methods for storing sparse matrices data. The main distinction between them is storage pattern, number of non-zero elements per row and overall matrix density. On the basis of those parameters we select the most suitable format for our task. Formats described in this section are not novel, however GPU architecture is so much different than CPU that requirements for storing methods are also different. Having it in mind, we present some basic formats which are suitable for GPU. Figure 2 illustrates the primary sparse matrix in dense format on which we will present sparse formats.

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

Fig. 2. Primary matrix in dense format

3.1. Coordinate Format (COO)

Coordinate format is the simplest and most flexible format for general sparse matrices. However, compared with other formats, it is very memory inefficient and computationally intensive. It comprises the arrays $data$ for non-zero elements, row and col representing indices in primary dense matrix. This is popular format for representing sparse matrices files, for example in MATLAB [18]. Figure 3 represents COO storage scheme.

$$
\begin{array}{rcl}
data & = & [\ 4\ 1\ 2\ 9\ 5\ 1\ 3\ 2\ 1\ 8\ ] \\
col & = & [\ 0\ 4\ 1\ 3\ 3\ 0\ 1\ 3\ 4\ 4\ ] \\
row & = & [\ 0\ 0\ 1\ 1\ 2\ 3\ 3\ 2\ 1\ 4\ ]
\end{array}
$$

Fig. 3. COO storage scheme

3.2. Compressed Sparse Row Format (CSR)

Compressed Sparse Row format is a general-purpose sparse matrix format (just like COO). The difference in storage scheme between row-sorted COO and CSR is in row array. In CSR third array contains offsets of each row represented in $data$ (see Figure 4). Due to similarity between CSR and COO, the conversions are straightforward (if COO is row-sorted). Row offsets facilitate efficient querying of matrix values which is very important in matrix-vector products.

$$
\begin{array}{rcl}
data & = & [\ 4\ 1\ 2\ 9\ 5\ 1\ 3\ 2\ 1\ 8\ ] \\
col & = & [\ 0\ 4\ 1\ 3\ 3\ 0\ 1\ 3\ 4\ 4\ ] \\
ptr & = & [\ 0\ 2\ 4\ 5\ 9\ 10\ ]
\end{array}
$$

Fig. 4. CSR storage scheme

3.3. ELLPACK Format (ELL)

ELLPACK [17] stores a sparse matrix on two arrays dimension $M \times NNZ$, where M is the number of rows and NNZ is the maximum number of non-zero elements per row. Rows which contain less than NNZ elements are padded with zeros. This format is well suited for sparse matrices with similar density in each row. Figure 5 illustrates $data$ and $indices$ matrices in ELL format.

$$
data = \begin{bmatrix} 4 & 1 & * & * \\ 2 & 9 & * & * \\ 5 & * & * & * \\ 1 & 3 & 2 & 1 \\ 8 & * & * & * \end{bmatrix} \qquad indices = \begin{bmatrix} 0 & 4 & * & * \\ 1 & 3 & * & * \\ 3 & * & * & * \\ 0 & 1 & 3 & 4 \\ 0 & * & * & * \end{bmatrix}
$$

Fig. 5. ELL storage scheme

3.4. Hybrid ELL+COO Format (HYB)

Hybrid format combines efficient memory bandwidth of ELL and flexibility of COO. It is very often the most efficient format for general sparse matrices. HYB stores the most common number of non-zeros per row in ELL and the rest part in COO (see Figure 6).

The number of optimal non-zeros per row can be computed using a histogram of the row sizes (this method is used in CUSP library [14] we discuss later).

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

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

Fig. 6. HYB storage scheme

4. Sparse Matrix-Vector Multiplication on GPU

Sparse matrix-vector multiplication (SpMV) is the most commonly used operation in sparse matrix computations. Performance of many scientific and engineering applications highly depends on the operation:

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{y}, \tag{5}$$

where $\mathbf{A}$ is a sparse matrix and $\mathbf{x}$ and $\mathbf{y}$ are dense vectors.

In this section we show the differences in implementing SpMV on CPU and GPU.

CUDA architecture is highly different than CPU. To achieve relatively good performance on GPU in SpMV with storage formats taken from CPU we often need to strongly change the algorithm. On GPU it is very important to properly access the memory. In CUDA implementations we should always consider transforming the algorithm so we can use effectively shared memory and get coalesced access to global memory. It always requires to process data in a specific way and order.

SpMV using CSR storage format is one of the examples where CPU implementation is effective and naive CUDA code is inefficient. A simple performance comparison was shown in [11]. Figures 7 and 8 illustrate basic implementations for CPU and GPU. This straightforward CUDA kernel processes one row per thread, thus it is very difficult to properly utilize the GPU.

One of the solutions for this problem is to split row processing across multiple threads as it was proposed in [2] and implemented in CUSP library [14]. In this variant one row is assigned to one warp. The results from each thread are accumulated in shared memory. When all threads within warp finish processing their parts, they sum all elements from shared memory vector using parallel reduction algorithm. Figure 9 shows this optimized kernel.

```
void host_csr_spmv(float *data, int *col, int *ptr,
  float *x, float *y, int size) {
    float result = 0;
    for (int i = 0; i < size; i++) {
      for (int j = ptr[i]; j < ptr[i+1]; j++)
        result += data[j] * x[col[j]];
      y[i] = result;
    }
}
```

Fig. 7. CPU implementation of SpMV using CSR format

```
__global__ void gpu_csr_spmv(float *data, int *col,
  int *ptr, float *x, float *y, int size) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < size) {
      float result = 0;
      for (int j = ptr[i]; j < ptr[i+1]; j++)
        result += data[j] * x[col[j]];
      y[i] = result;
    }
}
```

Fig. 8. Inefficient GPU implementation of SpMV using CSR format

```
__global__ void spmv_csr_vector_kernel(const int num_rows,
  const int *ptr, const int *indices, const float *data,
  const float *x, float *y) {
  __shared__ float vals[];
  int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
  int warp_id = thread_id / 32;
  int lane = thread_id & (32 - 1);
  int row = warp_id ;
  if (row < num_rows) {
    int row_start = ptr[row];
    int row_end = ptr[row + 1];
    vals[threadIdx.x] = 0;
    for (int jj = row_start+lane; jj < row_end; jj += 32)
      vals[threadIdx.x] += data[jj] * x[indices[jj]];
    if (lane < 16)
      vals[threadIdx.x]+=vals[threadIdx.x+16];
    if (lane < 8) vals[threadIdx.x]+=vals[threadIdx.x+8];
    if (lane < 4) vals[threadIdx.x]+=vals[threadIdx.x+4];
    if (lane < 2) vals[threadIdx.x]+=vals[threadIdx.x+2];
    if (lane < 1) vals[threadIdx.x]+=vals[threadIdx.x+1];
    if (lane == 0)
      y[row] += vals[threadIdx.x];
  }
}
```

Fig. 9. Efficient GPU implementation of SpMV using CSR format [2]

5. Numerical Experiments

In this section we test the performance of SpMV on GPU using CUSP library with different storage formats and MKL (Math Kernel Library, [16]) with CSR on CPU. The input matrices are divided into two sets. The first set contains matrices from different scientific fields like finite element method, quantum chromodynamics, epidemiology and proteins [19]. These data were also explored using GeForce GTX 280 in [2]. The second set of matrices we consider consists of transition rate matrices describing the Markovian queuing model of the DFC mechanism for various parameters.

5.1. Testing Environment

We intend to find the most suitable storage format for effective SpMV on the second set and compare the performance with SpMV on the first set. We also want to check what speed-up can get GPU over CPU in this computational problem, so we use MKL as a CPU representative. Table 1 shows specification of hardware and software used in experiment.

Tables 2 and 3 present details about matrices, where n is number of rows, nz number of non-zero elements, $minnz$ minimum number of non-zero elements in column, $maxnz$ maximum number of non-zero elements in column, $d = nz/n$ represents matrix row (or column) density and $dispersion$ is the average distance of non-zero elements from diagonal divided by matrix size n.

CPU	Intel Core i7 950 3.07GHz (48.96 Gflop/s SP)
Host memory	24 GB DDR3 1333 MHz (10 GB/s)
GPU	Tesla C2050 (1030 Gflop/s SP, 515 Gflop/s DP)
GPU memory	3 GB GDDR5 (144 GB/s with ECC off)
OS	CentOS 5.5 (Linux 2.6.18-164.el5)
Libraries	CUDA Toolkit 3.2, CUSP 0.1.2, Intel MKL 10.3

Table 1. Specification of hardware and software used in experiment

$name$	n	nz	$minnz$	$maxnz$	$d = nz/n$	$dispersion$
cant	62 451	2 034 917	1	40	32.58	0.002666
consph	83 334	3 046 907	1	78	36.56	0.023845
dense2	2 000	4 000 000	2 000	2 000	2 000	0.333333
mc2depi	525 825	2 100 225	2	4	3.99	0.000447
pdb1HYS	36 417	2 190 591	1	162	60.15	0.025975
qcd5_4	49 152	1 916 928	39	39	39	0.026423
rma10	46 835	2 374 001	4	145	50.69	0.135432
shipsec1	140 874	3 977 139	1	78	28.23	0.014202

Table 2. Description of unstructured matrices (first set)

$name$	n	nz	$minnz$	$maxnz$	$d = nz/n$	$dispersion$
m_2.1K	2 160	11 232	5	8	5.20	0.048636
m_2.3K	2 337	11 143	3	6	4.77	0.031019
m_3K	3 016	13 194	3	12	4.37	0.059780
m_14K	14 256	89 856	6	10	6.31	0.041074
m_40K	40 385	182 743	3	28	4.53	0.044207
m_50K	50 225	283 430	4	21	5.64	0.054648
m_68K	68 117	273 159	3	68	4.01	0.064590
m_130K	129 921	644 727	3	12	4.96	0.010990
m_650K	649 611	2 994 421	3	84	4.61	0.035446
m_1M	1 034 273	4 660 479	3	132	4.51	0.042191
m_2.6M	2 595 296	11 951 234	3	164	4.60	0.035123
m_3.7M	3 690 141	17 745 419	3	100	4.81	0.019029

Table 3. Description of Markov matrices (second set)

5.2. Results

In all examined examples we find that SpMV on GPU offers much better performance than CPU. Tables 4 and 6 illustrate time in seconds for single precision SpMV using various storage format on GPU and CSR from MKL on CPU. The bold values are the fastest computation times. COO as a simplest format was outperformed by the others. However CSR, ELL, and HYB achieve similar performance depending on matrix structure and size. CSR has quite stable speed across multiple matrices and it seems to be a good choice when we do not know anything about non-zeros pattern. For bigger Markov matrices the fastest SpMV was in HYB format. HYB does not perform well in many potentially suitable cases. The reason is that HYB SpMV granularity is not sufficient (one thread per row) and matrix needs to be large enough to utilize the GPU. For comparison in CSR kernel one row is processed by a warp (32 threads).

Tables 5 and 7 show speedup results for single precision SpMV using various storage format on GPU and CSR from MKL on CPU. The bold values are the best speedup.

Figure 10 shows the performance of SpMV in Gflop/s on unstructured matrices. We can see that the best utilization of CPU in CSR format is 8.6% for ‘shipsec1’ and the worst 4.1% for ‘mc2depi’ both in single precision. On GPU this situation is much worse, the highest utilization is only 1.4% for ‘consph’ in CSR and the smallest 0.7% for ‘qcd5_4’.

Usually, double precision computations are not more than twice longer than single precision ones (in GPU case only on Fermi family cards). Here, we observe larger differences because SpMV rely heavily on memory bandwidth.

Performance of SpMV on Markov matrices is lower than on previous set. The main reason is that Markov matrix has higher sparsity. In Figures 11 and 12 we can see that

$name$	$CSR\,(CPU/MKL)$	COO	CSR	ELL	HYB
cant	1.061285	0.549938	0.359008	**0.254403**	0.316862
consph	1.875143	0.826806	0.543938	**0.342179**	0.348769
dense2	2.443000	0.975000	**0.565000**	2.258999	0.978000
mc2depi	2.542999	0.849999	0.434999	0.280999	**0.275999**
pdb1HYS	1.392071	0.551584	**0.300497**	0.357471	0.366546
qcd5_4	1.527000	0.702999	0.554999	**0.250999**	0.255000
rma10	1.584999	0.737000	**0.470999**	0.642999	0.556999
shipsec1	1.893026	1.073512	0.800680	0.630669	**0.534465**

Table 4. Time in seconds of single precision SpMV on first set of matrices

$name$	COO	CSR	ELL	HYB
cant	1.93	2.96	**4.18**	3.35
consph	2.27	3.47	**5.48**	5.37
dense2	2.51	**4.32**	1.08	2.50
mc2depi	2.99	5.85	9.05	**9.21**
pdb1HYS	2.43	**4.64**	3.90	3.80
qcd5_4	2.17	2.75	**6.08**	5.99
rma10	2.51	**3.93**	2.88	3.32
shipsec1	1.76	2.36	3.00	**3.54**

Table 5: Speed-up of single precision GPU-accelerated SpMV on first set of matrices, relative to CPU/MKL performance

$name$	$CSR\,(CPU/MKL)$	COO	CSR	ELL	HYB
m_2.1K	0.012902	0.206822	**0.006916**	0.011318	0.207554
m_2.3K	0.011268	0.214451	**0.005285**	0.008456	0.210905
m_3K	0.013545	0.212084	**0.005806**	0.009192	0.213355
m_14K	0.075937	0.241931	0.026829	**0.024107**	0.226387
m_40K	0.140423	0.284207	0.042110	**0.040607**	0.223787
m_50K	0.217406	0.299878	0.080342	**0.066884**	0.234196
m_68K	0.333999	0.335000	**0.104000**	0.115000	0.270999
m_130K	0.810000	0.452000	0.174000	**0.167999**	0.292000
m_650K	3.425000	1.604999	0.637000	0.634999	**0.589999**
m_1M	4.325999	2.330999	0.963999	0.976999	**0.815999**
m_2.6M	15.30099	5.964000	2.378999	2.576999	**1.771000**
m_3.7M	16.83800	9.013999	4.018000	3.625999	**2.628000**

Table 6. Time in seconds of single precision SpMV on Markov matrices

$name$	COO	CSR	ELL	HYB
m_2.1K	0.06	**1.86**	1.08	0.06
m_2.3K	0.05	**2.20**	1.38	0.05
m_3K	0.07	**2.33**	1.56	0.07
m_14K	0.31	2.81	**3.17**	0.34
m_40K	0.50	3.36	**3.53**	0.63
m_50K	0.72	2.68	**3.24**	0.93
m_68K	1.00	**3.21**	2.90	1.23
m_130K	1.79	4.66	**4.82**	2.77
m_650K	2.14	5.38	5.40	**5.81**
m_1M	1.86	4.49	4.42	**5.30**
m_2.6M	2.57	6.43	5.94	**8.64**
m_3.7M	1.87	4.19	4.64	**6.40**

Table 7. Speed-up of single precision GPU-accelerated SpMV on Markov matrices, relative to CPU/MKL performance

for small matrices the best formats are CSR and ELL, while for bigger sizes HYB format is more suitable.

6. Conclusion

We have described some research on sparse matrix-vector product (SpMV) on GPU. We have presented four formats for storage of sparse matrices, namely: COO, CSR, ELL, HYB. For these formats we tested SpMV operation form CUSP library. These formats were studied for two sets of matrices.

The first set consists of unstructured sparse matrices of various origins. We observe performance in excess of 18 Gflop/s and 13 Gflop/s in single and double precision respectively. The best format is ELL or HYB. It depends on sparsity of a given matrix — for sparser ones (lower d) the better format is HYB, and for denser ones ELL is better. We observe speed-ups over 9 times compared to the best CPU performance.

The second set of matrices arises from our Markovian queuing model of a wireless network. These matrices are very sparse (d is very small). The performance of SpMV is lower for this set of matrices. We observe performance over 14 Gflop/s and 8 Gflop/s in single and double precision respectively for large matrices. The performance is still lower for smaller matrices and is about 9 Gflop/s and 6 Gflop/s in single and double precision respectively. The best format is HYB for larger matrices and CSR for smaller matrices (up to 50000 rows). We observe speed-ups over 8 times compared to the best CPU performance.

The operation SpMV for matrices of our Markovian model (we can suppose it is similar in case of other models) has worse performance than for general sparse matrices. The best format for them is HYB — especially for larger matrices (but it is normal for Markovian matrices that they tend to be very large). It is so because their dispersion is

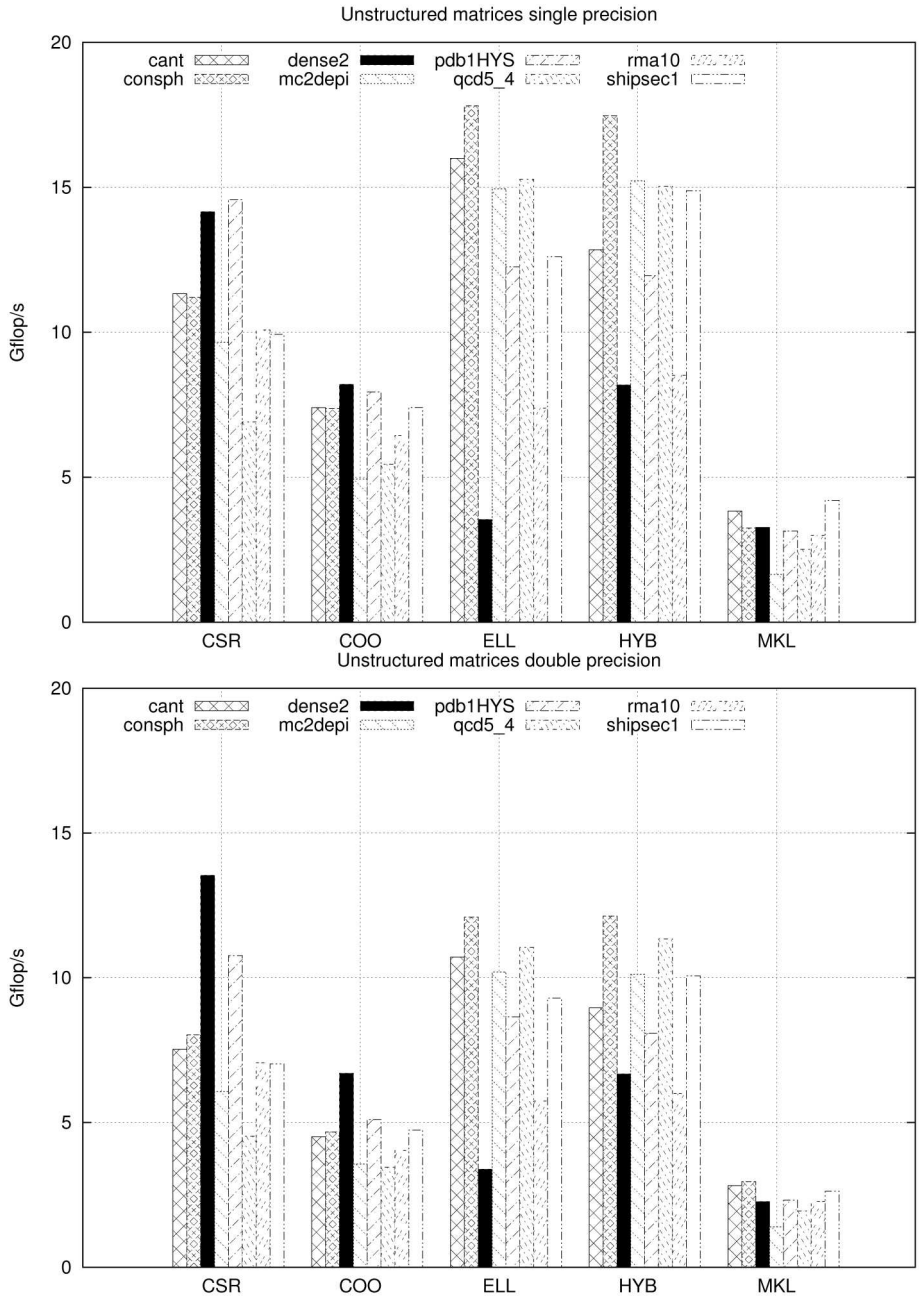

Fig. 10. Performance of SpMV on first set of matrices (top: single precision; bottom: double precision)

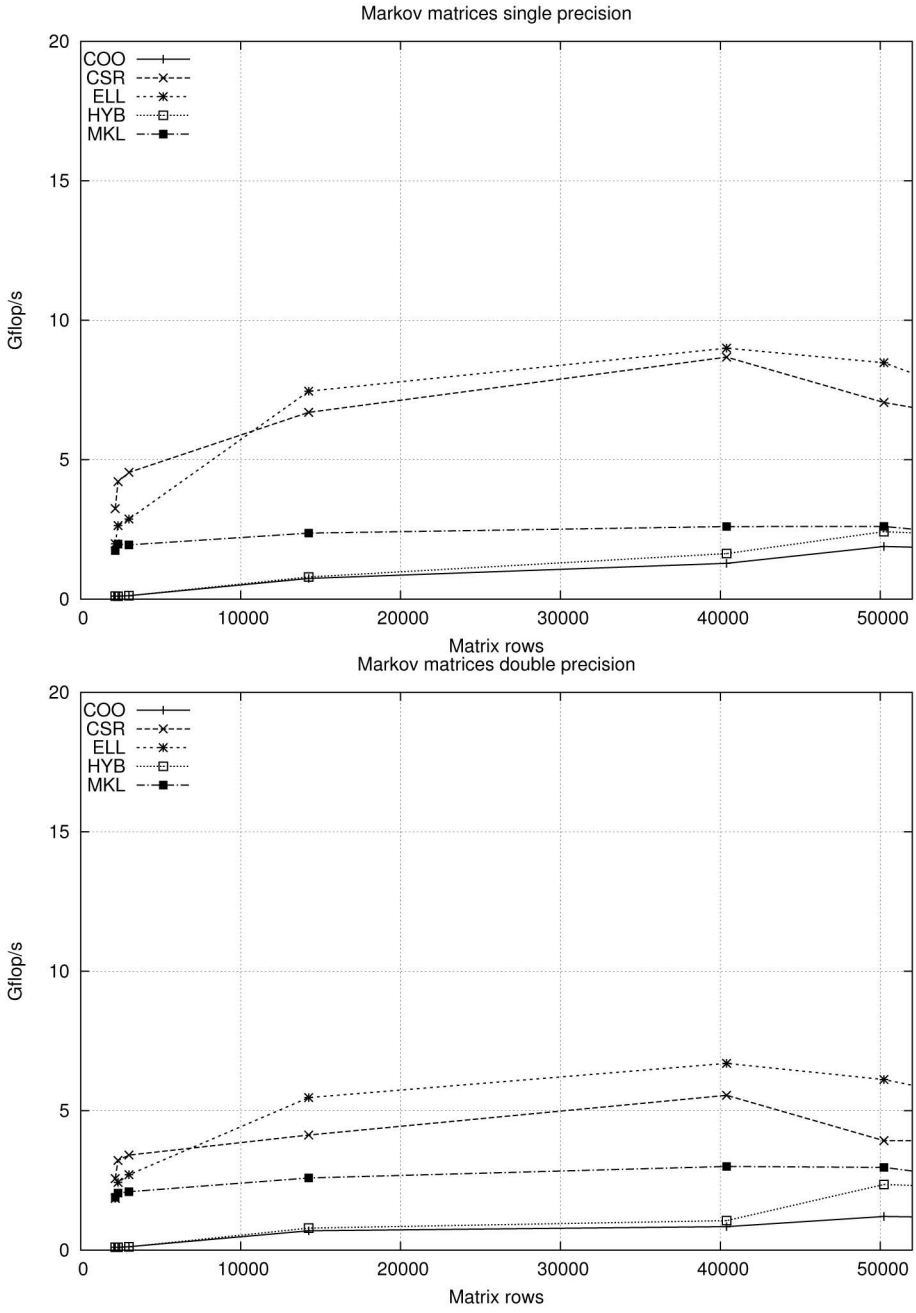

Fig. 11. Performance of SpMV on small Markov matrices (top: single precision; bottom: double precision)

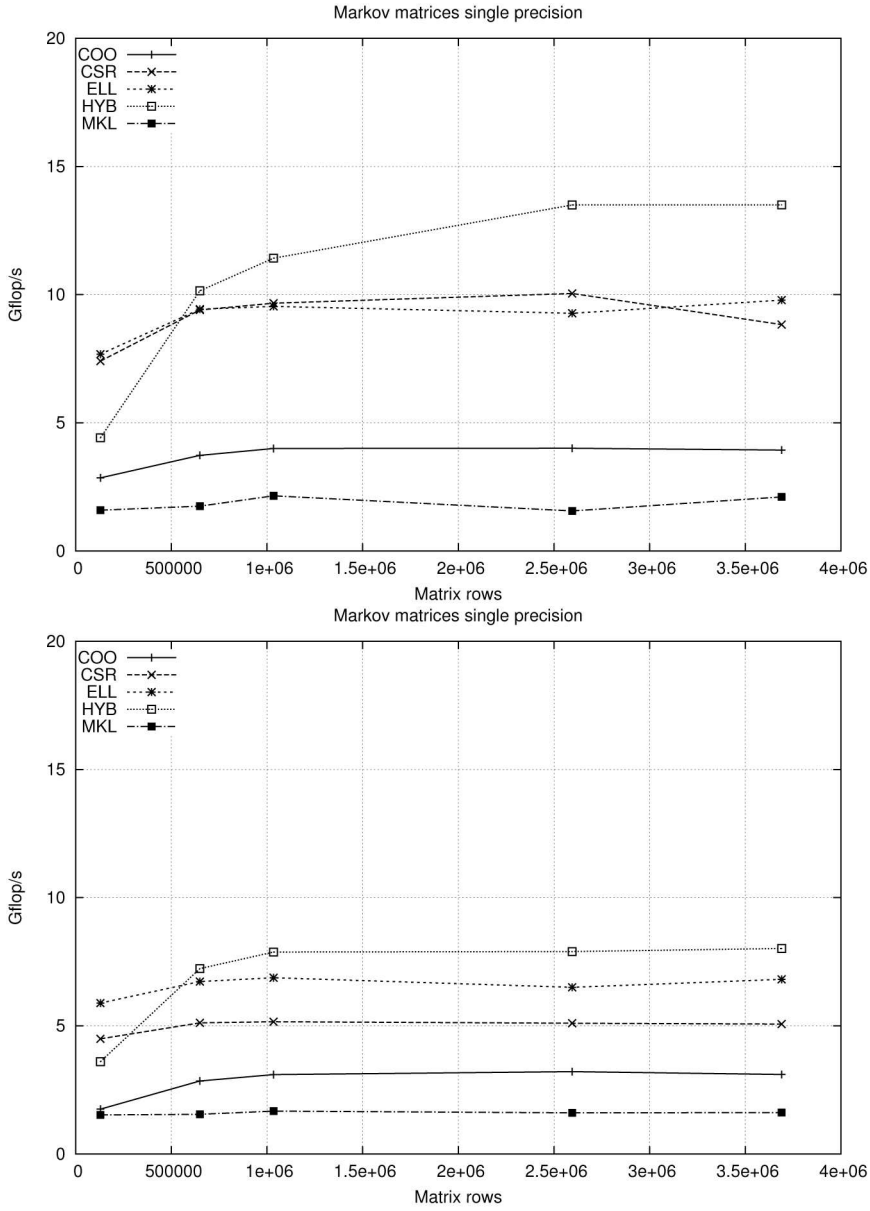

Fig. 12. Performance of SpMV on bigger Markov matrices (top: single precision; bottom: double precision)

quite high and their density is very low. But it seems that the future work connected to SpMV on GPU for Markovian matrices should consist in defining an entirely novel storage format which could even better use the possibilities of GPUs.

7. Acknowledgement

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".

References

1. J. Barnat, L. Brim, M. Ceska: *DiVinE-CUDA — a tool for GPU accelerated LTL model checking*, Proceedings of the 8th International Workshop on Parallel and Distributed Methods in Verification (PDMC'09), Eindhoven, November 2009, pp. 107-111.
2. N. Bell, M. Garland: *Efficient Sparse Matrix-Vector Multiplication on CUDA*, NVIDIA Tech. Report No. NVR-2008-004, 2008.
3. G. Bianchi: *Performance Analysis of the IEEE 802.11 Distributed Coordination Function*, IEEE Journal on Selected Areas in Communications, vol. 18, no. 3, March 2000, pp. 535-547.
4. J. Bylina, B. Bylina: *A Markovian Queuing Model of a WLAN Node*, CCIS 160, Computer Networks 2011, pp. 80-86.
5. IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Nov. 1997, P80211.
6. B. R. C. Magalhaes, N. J. Dingle, W. J. Knottenbelt: *GPU-enabled steady-state solution of large Markov models.*, 6th International Workshop on the Numerical Solution of Markov Chains (NSMC'10), 16-17 Sep 2010, Williamsburg, Virginia.
7. NVIDIA Corporation. CUDA Programming Guide. NVIDIA Corporation, 2009. `http://www.nvidia.com/`
8. 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).
9. W. J. Stewart: *Introduction to the numerical solution of Markov chains*, Princeton University Press, Princeton, NJ, 1994.
10. L.-C. Wang, S.-Y. Huang, A. Chen: *On the Throughput Performance of CSMA-based Wireless Local Area Network with Directional Antennas and Capture Effect: A Cross-layer Analytical Approach*, WCNC 2004 / IEEE Communications Society, pp. 1879-1884.
11. M. Wozniak, T. Olas, R. Wyrzykowski: *Parallel Implementation of Conjugate Gradient Method on Graphics Processors*, LNCS, PPAM 2009.

12. http://alice.loria.fr/index.php/software/4-library/
/23-opennl.html
13. http://code.google.com/p/cudpp/
14. http://code.google.com/p/cusp-library/
15. http://developer.nvidia.com/cuda-toolkit-40
16. http://software.intel.com/en-us/articles/intel-mkl/
17. http://www.cs.purdue.edu/ellpack/
18. http://www.mathworks.com/
19. http://www.nvidia.com/content/NV_Research/matrices.zip

Obliczeniowe aspekty mnożenia macierzy rzadkiej przez wektor dla rozwiązywania modeli Markowa przyspieszanego przez karty GPU

Streszczenie

Łańcuchy Markowa są przydatnym narzędziem do modelowania systemów złożonych, takich jak systemy i sieci komputerowe. W ostatnich latach łańcuchy Markowa zostały z powodzeniem wykorzystane do oceny pracy sieci bezprzewodowych. Jednym z problemów jaki się pojawia przy wykorzystywaniu łańcuchów Markowa w modelowaniu sieci są problemy natury obliczeniowej. W artykule zajmiemy się badaniem mnożenia macierzy rzadkiej przez wektor, które jest jedną z głównych operacji podczas numerycznego rozwiązywania modeli Markowowskich. Aby, przyspieszyć czas obliczeń mnożenia macierz rzadkiej przez wektor wykorzystano funkcje z biblioteki CUSP. Biblioteka jest zbiorem funkcji wykonywanych na GPU (ang.Graphics Processing Unit) celem skrócenia czasu obliczeń.

Do testowania operacji mnożenia macierzy rzadkiej przez wektor badano macierze z Markowowskiego modelu pracy sieci bezprzewodowej. Model ten opisuje zachowanie urządzenia, gdy kanał transmisyjnych może być zajęty przez inne urządzenia. Macierz przejść wspomnianego modelu jest macierzą rzadką i potrzeba specjalnej struktury danych do jej przechowywania, dlatego w artykule dyskutowane są różne struktury danych dla macierzy rzadkich i ich przydatność do obliczen na kartach graficznych.

W pracy porównano czas, wydajność i przyspieszenie jakie otrzymano podczas testowania biblioteki CUSP na karcie NVIDIA Tesla C2050 dla niestrukturalnych macierzy rzadkich opisujących model zajętości węzła w sieciach bezprzewodowych przy różnych formatach przechowywania macierzy rzadkich. Dla testowanych macierzy zauważono ośmiokrotne przyspieszenie obliczeń przy wykorzystaniu karty graficznej.