Efficient symmetric sparse matrix-vector product on a GPU

Mironowicz P., Dziekonski A., Mrozowski M.

Wireless Communication Engineering (WiComm) Center of Excellence,
Department of Microwave and Antenna Engineering, Faculty of Electronics, Telecommunications and Informatics,
CUDA Research Center for Computational Electromagnetics at Gdansk University of Technology, Poland

1. Introduction

Sparse matrix-vector product (SpMV) is one of the most widely used operations in computational sciences since it is a corner stone operation in solvers dedicated for **large systems of linear equations.**

Matrices that come from problems from various scientific disciplines are **large, sparse general or symmetric**. The most popular format for sparse matrices is CRS (Compressed Row Storage). There are many implementations of SpMV both for CPUs (*e.g.* Intel MKL) and GPUs (*e.g.* nVidia CUSPARSE).

Using sparse matrix symmetry one may both:
- Increase the SpMV performance
- Reduce the amount of memory reqired for storage

Currently only one implementation of **sparse matrix vector product on** a GPU that makes use of the symmetry exists in CuSparse Library, but it is slower even than CPU solutions.

The goal of our work is to introduce a GPU subroutine for a sparse symmetric matrix-vector product (SpMV) that both has high performance and reduced memory consumption.

2. Task scheduling approach for SpMV

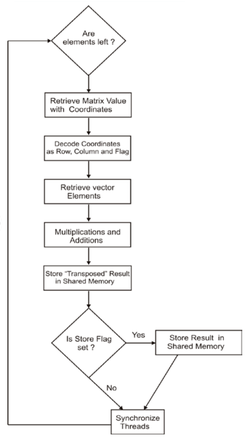
Main features of our approach:
- **Task scheduling** method is used to improve SpMV
- New matrix **storage format** for symmetric sparse matrices
- Allows one to **save** about **15-35%** of the memory.
- TSMV achieves performance:

+ Similar or significantly higher than CuSparse for general matrices.
+ The order of magnitude higher than CuSparse (symmetric), Intel MKL(general/symmetric)

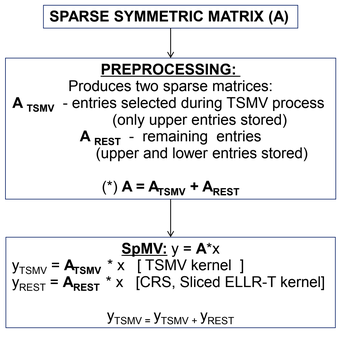
Method overview

1.The key point, is to find regions in which **nonzero entries are concentrated**.
2. It is possible to select a **small number** of square submatrices (***blocks***) that **contain a significant part** of the non-zero entries
3. For each block it is possible to apply task scheduling to **reorganize entries** and as a result to prevent from memory access conflicts (which requires **atomic operations**).

TSMV iteration:



SpMV with TSMV - scheme:



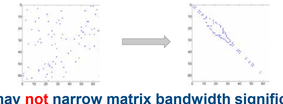
TSMV - Implementation details :

Preprocessing on a CPU: square submatrices with nonzero entries concentrated are preprocessed on a CPU. The task scheduling procedure assures that no two threads are accessing the same value in the shared memory on a GPU. Preprocessing is performed once on a matrix of given values pattern, it is parallelized with *OpenMP* and can be hidden by the CUDA initialization.
TSMV kernel on a GPU: each square submatrix is assigned to a single block of CUDA threads. Multiplications on the values are performed in a loop shown in a picture. The results are stored in shared memory. Each value has coordinates encoded into one integer value with a *store flag*. In subsequent iterations a single thread is simultaneously performing multiplication corresponding to *normal* (rows) and *transposed* (columns) *order* of the values, so each value and coordinate has to be retrieved from global memory only once.
To each of the threads, the subsequent assigned values tend to be taken from the same row, so the sum of the results of multiplications may be accumulated in a register, and stored when the thread moves to another row, which is signalized by the *store flag*. In consequence in each iteration no two threads may finish its work on elements with the same row, nor work on elements with the same column.
Sometimes it is impossible to the task scheduler to assign elements satisfying these conditions, to all threads. In this situation *fake values* equal to 0 with arbitrary, non-conflicting coordinates are assigned (*fake values are also stored in the memory, which decreases memory savings. It is planned to be improved in further version).*

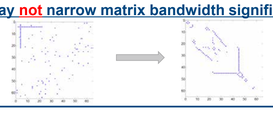
RCM - Reverse Cuthill-McKee ordering

- With RCM ordering applied to a sparse matrix **(A)** a new sparse matrix **(A*) can have narrower bandwidth.** As a result, square submatrices that **contain a significant part** of the non-zero entries are denser, which naturally bring fruits in the TSMV approach, since more nonzero entries land in $\mathbf{A^*_{TSMV}}$ matrix and higher performance may be achieved and more memory can be saved with the TSMV approach
- In some cases (see the picture on the right) the RCM ordering cannot narrow sparse matrix bandwidth, which is caused by the structure of the original matrix.

RCM may narrow matrix bandwidth significantly



RCM may not narrow matrix bandwidth significantly



3. Numerical Results

Test problems - 9 large sparse symmetric matrices:
- **7** matrices from the Williams collection
- **2** matrices from FEM discredited GSM filter

Workstation :
GPU: 1x Tesla C2075
CPU: 2x Opteron 6174 (24 cores)

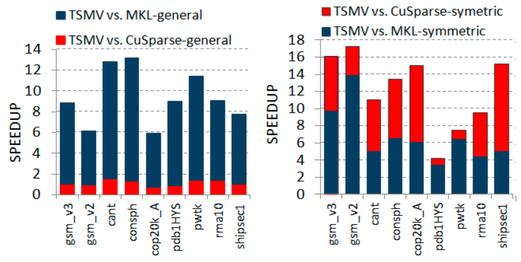
Matrix	Matrix size	Nonzeros
gsm_v3	806 811	56 170 107
gsm_v2	589 446	21 758 924
cant	62 451	4 007 383
consph	83 334	6 010 480
cop20k_A	121 192	2 624 331
pdb1HYS	36 417	4 344 765
pwtk	217 918	11 524 432
rma10	46 835	2 374 001
shipsec1	140 874	2 655 299

Preprocessing issue

Matrix	CuSparse [s]	TSMV – CPU preporcessing [s]
gsm_v3	1,29	1,56
gsm_v2	1,29	0,90
cant	1,29	0,04
consph	1,29	0,08
cop20k_A	1,29	0,09
pdb1HYS	1,29	0,06
pwtk	1,29	0,13
rma10	1,29	0,04
shipsec1	1,29	0,04

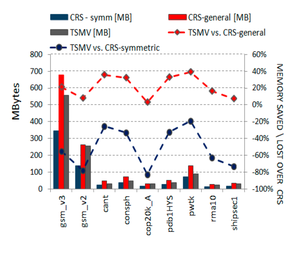
1) An **initialization of the CUSPARSE library** cannot be neglected
2) For matrices smaller than **0.7 mln unknowns** the time of TSMV preprocessing is smaller than CUSPARSE initialization

TSMV vs. other CPU , GPU implementations



1) TSMV vs. implementations dedicated for **general** matrices:
+ TSMV achieves similar or better performance than CuSparse for general matrices, and much better than Intel MKL
2) TSMV vs. implementations for **symmetric** matrices:
+ TSMV achieves **from several to dozen or so** better performance than CuSparse and Intel MKL

Memory



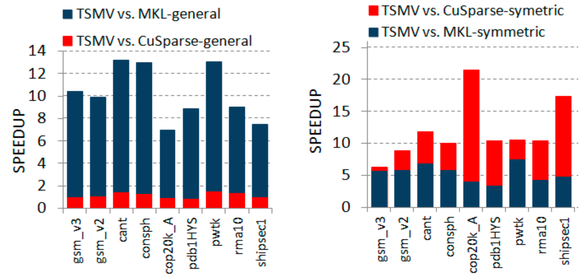
1) For matrices with narrow bandwidth about 20-40 % memory may be saved with TSMV approach **(cant, consph, pwtk, rma10)**

IMPROVEMETS DUE TO THE RCM REORDEING

SpMV time reduced thanks to RCM

Matrix	MKL general	MKL symmetric	CuSparse general	CuSparse symmetric	TSMV
gsm_v3	-29%	-65%	-34%	-76%	**-39%**
gsm_v2	-18%	-79%	-38%	-74%	**-49%**
cant	6%	41%	-1%	10%	0%
consph	3%	-7%	3%	-22%	0%
cop20k_A	-13%	-51%	-2%	6%	**-26%**
pdb1HYS	-4%	-3%	-2%	141%	**-2%**
pwtk	0%	0%	-2%	23%	**-13%**
rma10	-3%	-6%	-1%	5%	**-3%**
shipsec1	-2%	-3%	4%	16%	0%

SPEEDUP



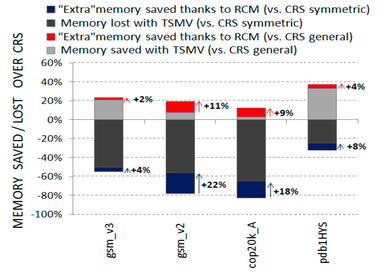
RCM ordering allowed to achieve better :

+ performance (time) of SpMV with TSMV approach especially for matrices with originally wide bandwidth (gsm_v3, gsm_v2, cop20k_A, pdb1HYS) [similar conclusion for MKL, and CuSparse)

+ speedup achieved for TSMV over Intel MKL, CuSparse general and symmetric

+ memory utilization for matrices with originally wider bandwidth **(gsm_v3, gsm_v2, cop20k_A, pdb1HYS**)

MEMORY



Research supported by NVIDIA Inc. and the Polish Government with statutory research funds and the Polish National Research and Development Center (Lider project) and the Polish National Science Center (Etiuda scholarship) and the Foundation for Polish Science (START scholarship)