BLUE WATERS

SUSTAINED PETASCALE COMPUTING

Adaptive Thread Distributions for Sparse Matrix-Vector Multiply on a GPU

William Gropp and Dahai Guo

NCSA, UIUC

GREAT LAKES CONSORTIUM FOR PETASCALE COMPUTATION

Outline

- Introduction
- Challenges for SpMV on GPU
- Related work and the limitations
- Adaptive thread distributions for SpMV
- Test Results
- The AUTOCSR process
- Summary and Conclusions

Introduction

- Sparse Matrix and Vector multiply (SpMV) dominates the computations in many iterative methods , such as
  - The Krylov subspace method in scientific computing:
  - Google page rank algorithm used for web searches:
- The GPU is a powerful tool for accelerating computations with its many streaming multiprocessors.
- We present a simple auto-tuning process for SpMV on a GPU, based on the CSR format.

Challenges for SpMV on GPU

- Need to make efficient use of GPU global memory:

  Memory coalescing means sixteen consecutive threads (a half warp) access consecutive addresses in the GPU's global memory. It can significantly reduce the latency and improve the memory bandwidth for data loaded from global memory.

- Need to make efficient use of all GPU cores:

  32 threads (1 warp) physically run in parallel. Balanced workloads are needed to efficiently use parallel threads. Otherwise some threads are idle while other threads are busy.

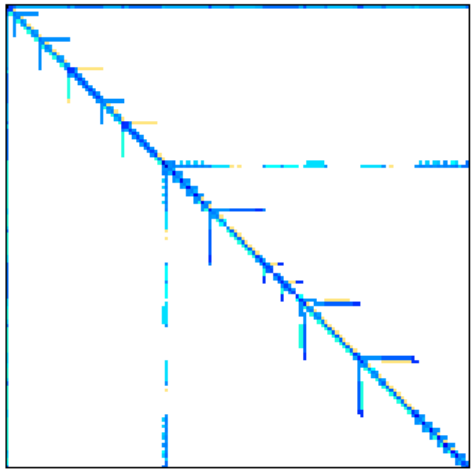
- However, using GPUs for SpMV computation is a challenge because of the randomness of nonzero element distributions in different matrices.

Related Work

- Nathan Bell and Michael Garland evaluated the performance of several storage formats on GPU (scalar and vector CSR, ELL, COO, HYB) and developed a SpMV package for GPUs.
- M. M. Baskaran and R. Bordawekar discussed some tuning approaches for SpMV on a GPU, such as synchronization-free parallelism, thread mapping, global memory access, data reuse, and achieved some performance improvement.
- J. W. Choi *et al* developed a blocked ELL format and described a model-driven auto-tuning framework.
- István Reguly and Miles Giles tuned the number of threads applied for each row of the whole matrix (TPR_C) using the vector-CSR format, and discussed how the different values of TPR and some other parameters affect the performance of SpMV on a GPU.

Limitations of the current formats

- The ELL format needs to add zeros in order to ensure that each row has the same number of elements so that the data can be loaded through “memory coalescing”. If too many zeros are added, the ELL format becomes inefficient.
- The HYB (ELL + COO) format partially solves the problem, but it requires more complex program logic.
- Because the TPR_C method uses a constant value of TPR for the entire matrix, it is very difficult to adjust and balance the workloads for matrices having very different numbers of nonzero elements per row, such as the matrix “ibm-dc1”.
- It can be expensive to convert matrices between different storage formats.



The matrix “ibm-dc1”

The Adaptive Thread Distributions

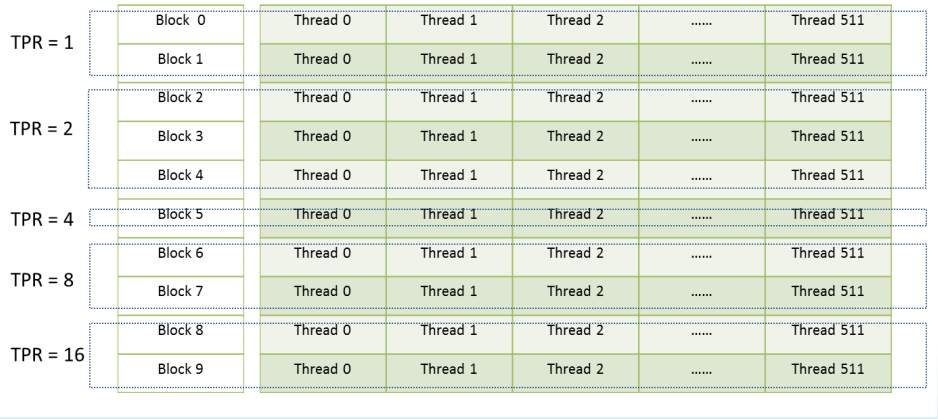
We adapt the number of threads used for each row of the matrix in order to balance the workload:

- Sort the matrix in the increasing order, based on the number of nonzero elements in each row.
- Partition the sorted matrix into several ranges of length; each range is adaptively assigned a number of threads per row (TPR_A) according to the range length.
- The number of GPU blocks is then calculated to handle the computation in the matrix range.

The Adaptive Thread Distributions (cont'd)

For example,

- The value of 512 (= 2^9) is set as the GPU block size.
- The matrix is partitioned into ten ranges with TPR = 2^0, 2^1, …, 2^i,…, or 2^9 respectively.
- TPR = 2^0 for the matrix range in which each row has at most 8 NNZ elements, 2^1 for the rows between 8 and 8*2, 2^2 for those in the range (8*2, 8*4], …, and 2^9 for the matrix rows in the range (8*512, max]. If there is no row in a certain range, then no GPU block is distributed to that range.
- In the right figure the matrix is signed with ten GPU blocks, two with TPR =1, three with TPR=2, …, and two with TPR = 16.



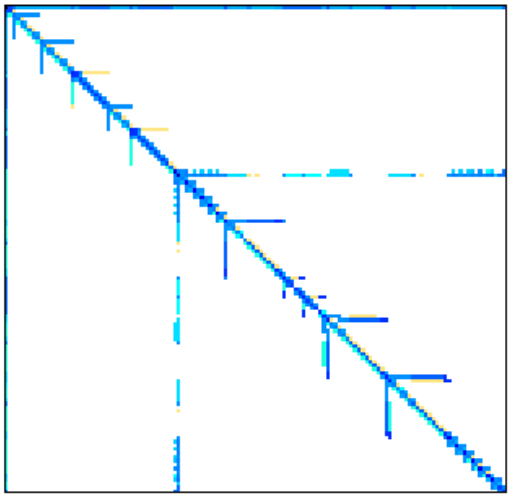
The Tested Matrices

- The Dell NVIDIA Linux cluster "*Forge*" at NCSA is used for the tests. It is equipped with AMD Opteron 2.4 GHz processors and NVIDIA Fermi M2070 GPU accelerators
- Test matrices are chosen from "The University of Florida Sparse Matrix Collection".

Matrix	Rows	Columns	Nonzeros	Nonzeros/Row
audikw_1	943695	943695	77651847	82.3
torso1	116158	116158	8516500	73.3
TSOPF_RS_b2383	38120	38120	16171169	424.2
cage14	1505785	1505785	27130349	18.0
Ldoor	952203	952203	46522475	48.9
Msdoor	415863	415863	20240935	48.7
ibm-dc1	116835	116835	766396	6.6
raefsky3	21200	21200	1488768	70.2
Pwtk	217918	217918	11634424	53.4
pdb1HYS	36417	36417	4344765	119.3
mac_econ_fwd500	206500	206500	1273389	6.2
mc2depi	525825	525825	2100225	4.0
Scircuit	170998	170998	958936	5.6
webbase-1M	1000005	1000005	3105536	3.1
web-Google	916428	916428	5105039	5.6
Stanford	281903	281903	2312497	8.2
Stanford_Berkeley	683446	683446	7583376	11.1

Matrix "ibm-dc1"

- We begin the tests with the matrix "*ibm-dc1*", which is one of the matrices we find hard to speed up, both on CPUs and GPUs.
- It has a very irregular nonzero distribution along the rows. Most of the rows have fewer than 288 nonzero elements. However, there are two rows that have extremely large numbers of nonzero elements, 47193 and 114190 respectively.
- It is hard to balance the workload of each thread if a constant number of threads per row (TPR_C) is applied to the whole matrix.



Two adaptive TPR methods for "ibm-dc1"

Two adaptive thread methods (TPR_A) are used to handle the computations for those two long rows:

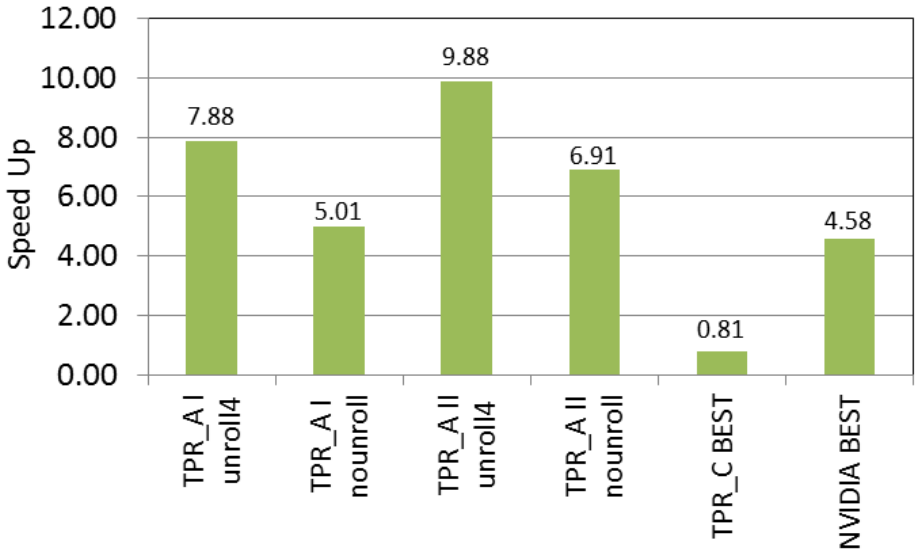
- (1) TPR_A I: one GPU block for each row. The average nonzero elements handled by those two blocks (80691 per block) are still much larger than other blocks.
- (2) TPR_A II: multiple GPU blocks for each row. Only a little additional work is needed to collect the sum from each block and obtain the final result for the row. It results in much better workload balance (4296 per block).

Row NNZ range	TPR	Rows	TPR_A I		TPR_A II	
			GPU blocks	Ave. NNZ per block	GPU blocks	Ave. NNZ per block
(0, 8*1]	1	103968	203	2096	203	2096
(8*1, 8*2]	2	10011	39	2891	39	2891
(8*2, 8*4]	4	2794	21	2730	21	2730
(8*4, 8*8]	8	8	0	0	0	0
(8*8, 8*16]	16	22	1	2190	1	2190
(8*16, 8*32]	32	24	1	5370	1	5370
(8*32, 8*64]	64	6	1	1710	1	1710
(8*64 , 8*128]	128	0	0	0	0	0
(8*128, 8*256]	256	0	0	0	0	0
(8*256, 8*512]	512	0	0	0	0	0
(8*512, MAX]	512	2	**2**	**80691**	**38**	**4246**

Table 2: Range partition of the matrix "ibm-dc1" and assigned GPU blocks information

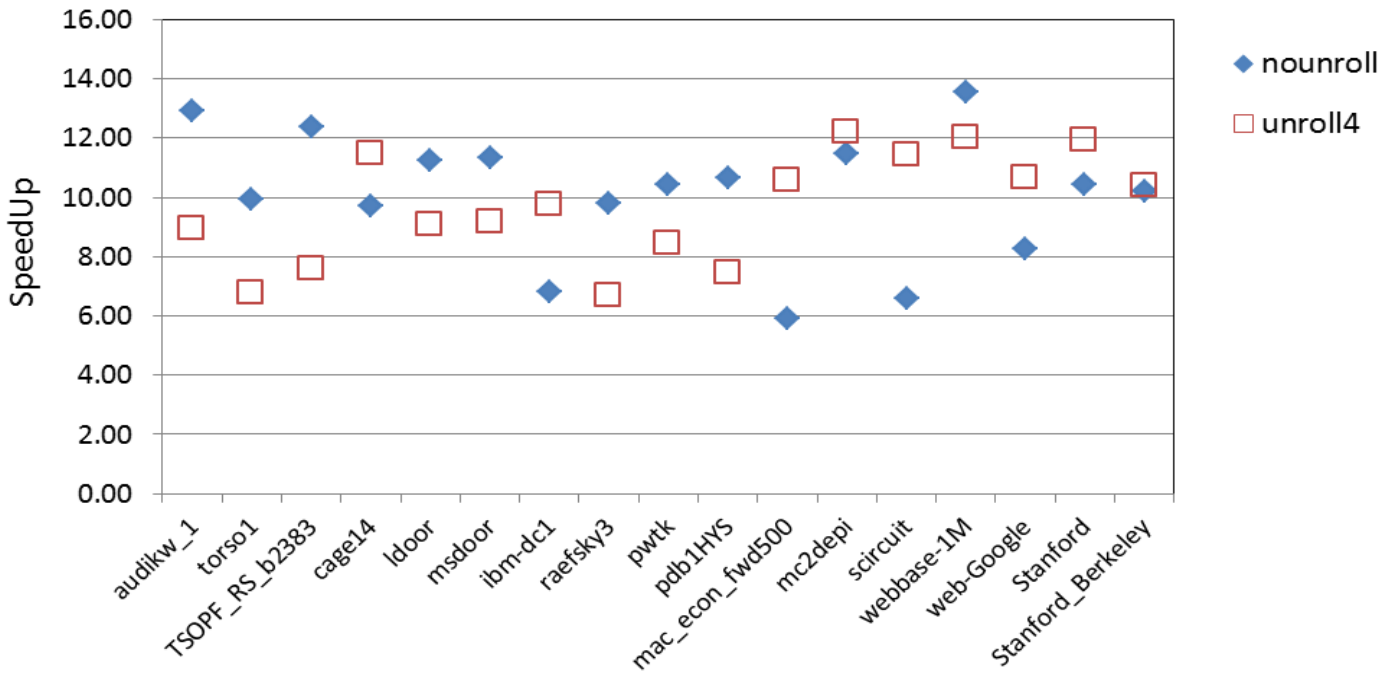
Speed Up on GPU for "ibm-dc1"

- The best speedup in the NVIDIA SpMV package is 4.58.
- The performance of the method with the constant TPR (TPR_C) is even worse than the CPU serial run.
- The adaptive multiple TPR method (TPR_A) can significantly improve the performance on GPU.
- The speedup from TPR_A I is 7.88.
- When the multiple GPU blocks (TPR_A II) are employed for the two long rows and the inner loop is unrolled by 4, it achieves a speedup of 9.88.
- Over 100% improvement compared to the NVIDIA best result



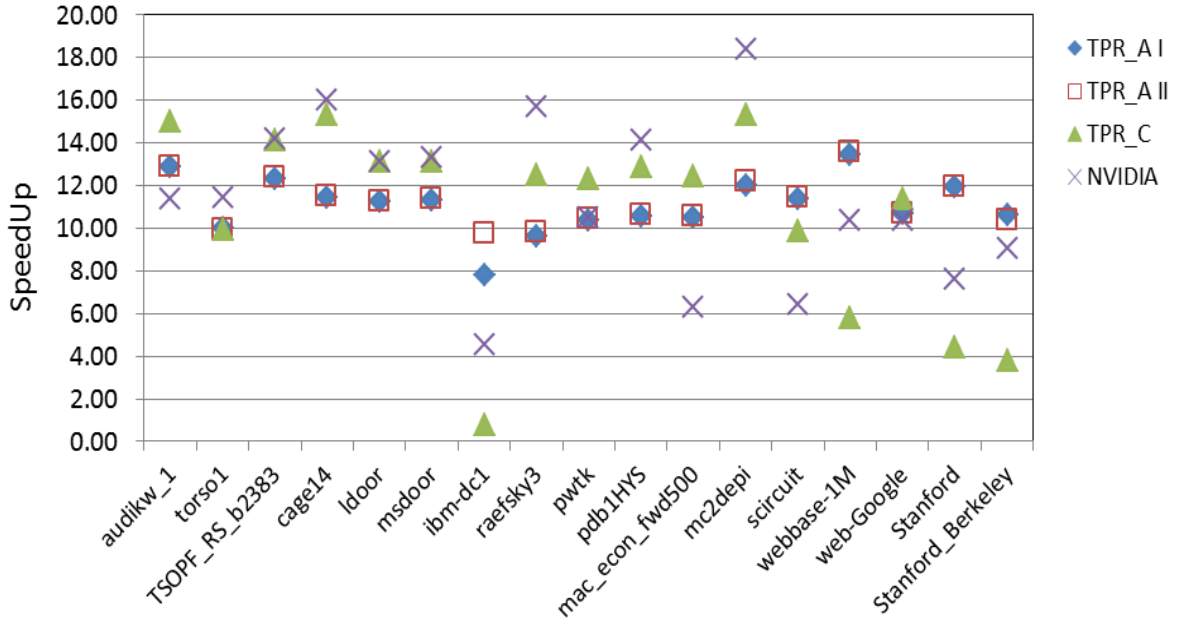
Speed Up for More Matrices: unrolling

- Unrolling the inner loop results in mixed performance; better for six of the matrices, worse for eleven.



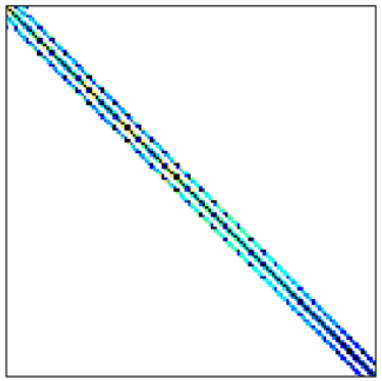
Speed Up for More Matrices: TPR choice

- The TPR methods (either TPR_C or TPR_A) achieve as good as or significantly better performance for fifteen matrices, compared to the best result from the NVIDIA SpMV package.

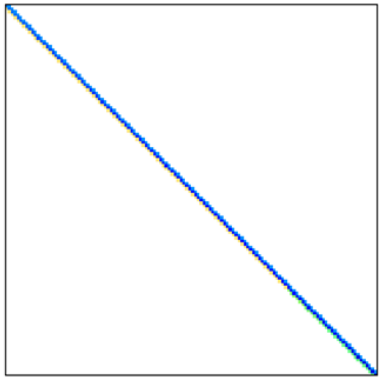


The ELL Format in NVIDIA SpMV Package

- The ELL format in NVIDIA results in significantly better performance than "autocsr" for two matrices "*raefsky3*" and "*mc2depi*".
- Those two matrices both have similar numbers of nonzero elements per row, and it is easy to store them in the ELL format without adding too many zeros.
- The advantage of the ELL format is that it can fully exploit "memory coalescing" when loading data.
- However, it is difficult to take advantage of the ELL format in general because of the irregularity of matrix sparsity patterns.



(a) raefsky3



(b) mc2depi

The Format Choices in NVIDIA SpMV Package

- The NVIDIA SpMV package achieves the best results with either csr_vector, ELL, COO or HYB formats (*_tex* means the texture cache is used for vector X):
  - five matrices with the csr_vector format
  - five with the ELL format
  - five with the COO format
  - two with the HYB (ELL+HYB) format
- No format is significantly better than others.

Matrix	Format
audikw_1	csr_vector_tex
torso1	csr_vector_tex
TSOPF_RS_b2383	csr_vector_tex
cage14	ell
ldoor	ell_tex
msdoor	ell
ibm-dc1	coo_flat_tex
raefsky3	ell
pwtk	csr_vector_tex
pdb1HYS	csr_vector_tex
mac_econ_fwd500	hyb_tex
mc2depi	ell
scircuit	coo_flat_tex
webbase-1M	hyb
web-Google	coo_flat_tex
Stanford	coo_flat_tex
Stanford_Berkeley	coo_flat_tex

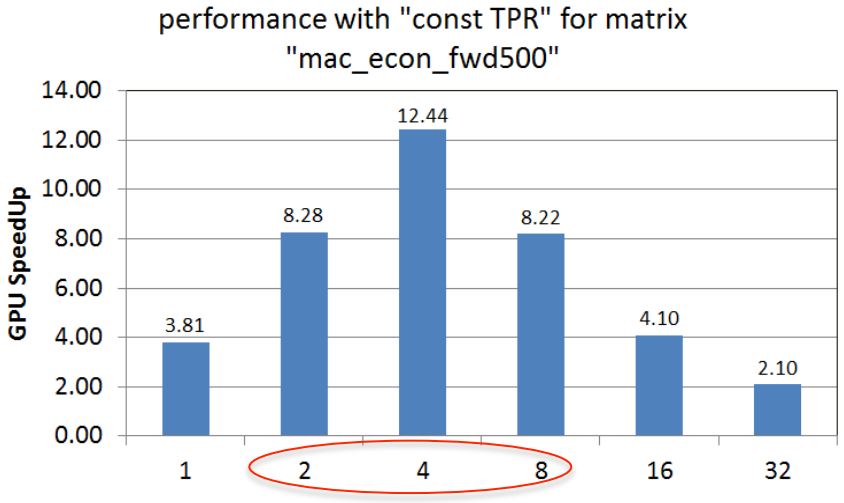
The AUTOCSR Process

- The methods “TPR_A” and “TPR_C” are combined together for auto-tuning, since both of them are based on the CSR format.
- In the first several iterations, different methods or TPR values are tried to decide the optimal method/value for the rest of iterations.
- However, the exhaustive search for the optimal value of the “TPR_C” may significantly slow down the tuning process.

The AUTOCSR Process (Cont'd)

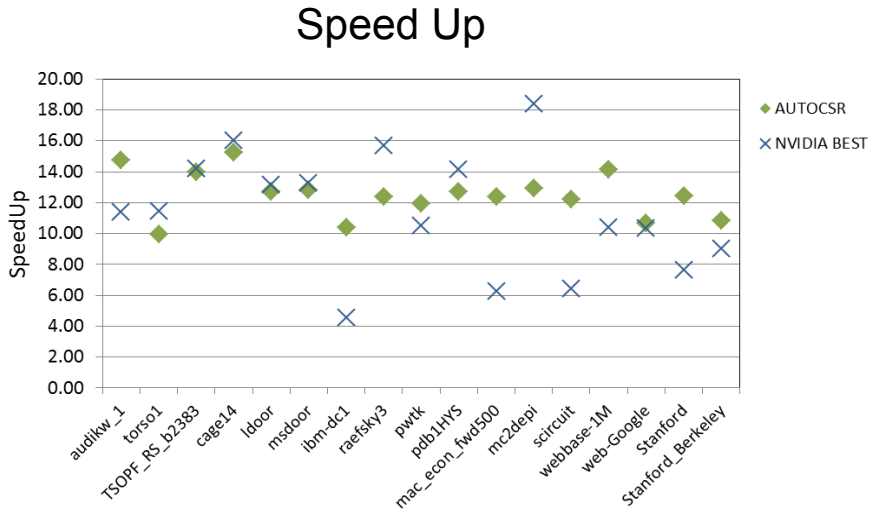
- For "TPR_C", only three values of TPR based on the average number of nonzero elements per row (*ave_nnz* = *NNZ / Rows*) are tried, in order to accelerate the tuning process.

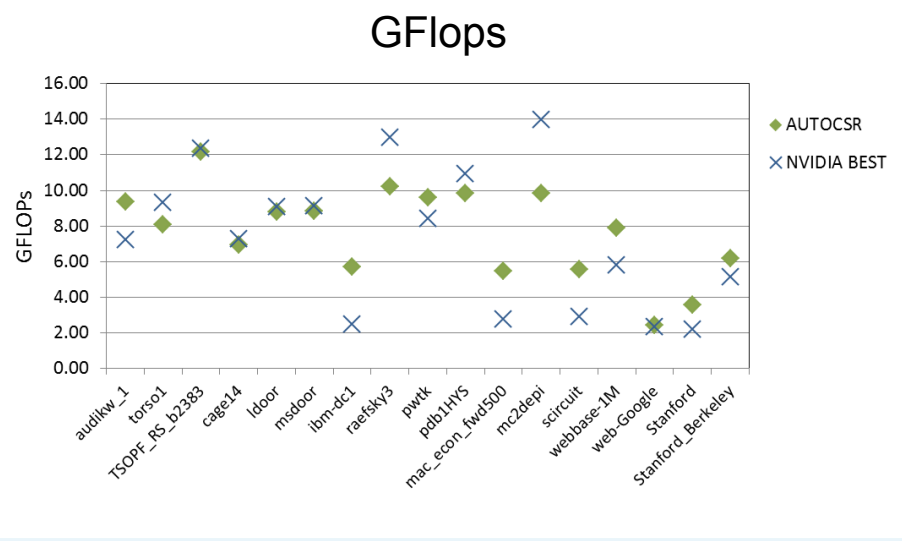
- Generally, when "*ave_nnz*" is smaller than 16, the TPR values of 2, 4 and 8 are tried; 4, 8 and 16 are used when *ave_nnz* is between 16 and 32, and 8; 16 and 32 are tested when *ave_nnz* is larger than 32.



Performance of the AUTOCSR Process

- The results match what we obtained before. The "AUTOCSR" data shows the average speedup over 200 iterations achieved with our tuning program on the GPU, including the first several iterations which search for the best kernels and the optimized parameters.





Summary and Conclusions

- The “AUTOCSR” process achieves good performance for most of the tested matrices.
- It is easy to implement and the tuning process is fast.
- We do not expect that it will always result in the optimal performance.
- Further optimization of the method to balance the workload on each GPU thread may obtain more performance benefit for SpMV on a GPU.

Acknowledgements

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications and the Great Lakes Consortium for Petascale Computation.

Thanks.

Questions?