

Automatic Performance Tuning of SpMV on GPGPU

Xianyi Zhang¹, Yunquan Zhang^{1,2}, Xiangzheng Sun^{1,2,3}, Fangfang Liu¹, Shengfei Liu^{1,2,3},
Yuxin Tang^{1,2,3} and Yucheng Li¹

¹Laboratory of Parallel Computing, Institute of Software, Chinese Academy of Sciences

²State Key Laboratory of Computer Science, Chinese Academy of Sciences

³Graduate University of Chinese Academy of Sciences
{zxy, zyq, sxz, liuff, lsf, tangyx, lyc}@mail.rdcps.ac.cn

Abstract

Sparse Matrix-Vector Multiplication (SpMV) is an important computational kernel in scientific applications that tends to perform poorly on modern processors because of irregular memory accesses. GPU have evolved into a very attractive hardware platform for general purpose computations due to their high floating-point computation performance, which results in that GPGPU becomes the hot and popular topic in HPC. Therefore, we need to parallelize and optimize SpMV on GPGPU to get a better performance. In this paper, we studied the register-level blocking algorithm and the heuristic algorithm to optimize SpMV performance on GPGPU. Based on AMD Stream Computing, We propose an automatic performance tuning SpMV software package on GPGPU, with its name GOSpMV. Experimental results on AMD Radeon HD 3690 shows that, compared with the CSR format SpMV implementation on CPU, our GOSpMV can achieve an average speedup with 3.11 and the max speedup is 5.95.

Keywords: SpMV, self adapting algorithm, heuristic-register blocking algorithm, GPGPU, AMD Stream Computing

1. Introduction

Sparse Matrix-Vector Multiplication (SpMV) is one of the most important kernels that are widely used in a lot of scientific applications, such as PDE solver, image processing and the simulation of a large variety of physical, financial, social etc. SpMV performs poorly on modern processors because of its low computation-to-memory-access ratio and irregular memory access patterns, and the deeper memory hierarchy makes optimization even more difficult to perform.

The rapid evolution of GPUs in performance, architecture and programmability can provide application potential beyond their primary purpose of graphics processing. GPU-based general-purpose computing (GPGPU, General-Purpose Computation on GPU) refers to the use of the graphics card to achieve general computing, rather than simply drawing [1]. Graphic processing units (GPU) have evolved into a very attractive hardware platform for general purpose computations due to their extremely high floating-point processing performance, huge memory bandwidth and their comparatively energy low cost.

AMD Stream Computing [2] is the GPGPU solution on AMD hardware. It could be applied for financial analysis, seismic migration analysis, and life sciences applications. GPUs such as AMD RV770 can achieve single-precision 1Tera Flops, which is much faster than the traditional CPU computing platform. GPGPU has the natural advantage of multithreading in parallel computing. Therefore, how to parallelize and optimize SpMV on GPGPUs has become very important issue.

The rest of this paper is organized as follows: In section 2 we present a brief overview of related work about SpMV and GPGPU; In section 3 we propose the architecture and module details of GOSpMV; In section 4 we perform experiments with 8 sparse matrices selected from different application area, and analyze their performance; Section 5 is our conclusions and the future work.

2. Related Work

2.1. Introduction to SpMV Algorithms

We consider the SpMV operation $y=y+Ax$, if y is initialized with 0, the SpMV becomes $y=Ax$. Here A is an $m \times n$ sparse matrix with nz non-zero elements. The x and y are dense vectors.

The most frequently applied storage format for sparse matrices is the Compressed Sparse Row (CSR) format. The nz non-zero elements are stored in row-major order contiguously in memory. It needs three arrays as follows:

$A_val[nz]$: stores the value of each non-zero element in the sparse matrix;

$A_col[nz]$: contains all column indices of non-zero element in the sparse matrix;

$A_ptr[m+1]$: stores indices indicating the location in $A_val[nz]$ and $A_col[nz]$ of the first element of each row, and $A_ptr[m]=nz$ (the index values of A_ptr start at 0).

The SpMV in CSR format is shown in Figure 1.

```

for(i=0;i<m;i++)
{
    value = y[i];
    for(j = A_ptr[i];j<A_ptr[i+1];j++)
        value = value + A_val[j]*x[A_col[j]];
    y[i] = value;
}

```

Figure 1: SpMV in CSR format

In our paper, we use Blocked Compressed Sparse Row (BCSR) format, which is blocked variant of CSR. It divides the matrix A into some $r \times c$ blocks and stores every block sequentially. If a block has only a part of non-zeros element, we should fill the other location with 0. Therefore, the 0 values should be stored explicitly. It also needs three arrays as follows:

A_val : stores the value of every block sequentially;

A_col : records the column index of the first non-zeros element of every block, every block only needs one column index;

A_ptr : stores indices indicating the location of the first block of each block row.

2.2. Performance Optimization of SpMV on CPU

There are many existing SpMV optimization techniques, including Register Blocking [3-7], Cache Blocking [3,4,8,9], Multi-vector Technology [4], Symmetric Structure [6], Diagonal Technology [6], Re-arrangement [6], and so on. SPARSITY package [4] optimized the SpMV core and solved the SpMV and SpMM with automatic optimization technology. Based on SPARSITY, the BeBOP (Berkeley Benchmarking and Optimization Group) [10] developed OSKI (The Optimized Sparse Kernel Interface) interface [11, 12], which is an automatic performance optimization package to optimize SpMV. Both SPARSITY and

OSKI adopt a heuristic algorithm to determine the optimal block size of sparse matrix, in order to improve the performance of SpMV.

Yuan [13] shows that, using BCSR format can indeed improve the SpMV performance, and the heuristic algorithm can also choose a high performance block size for matrices with regular non-zero items. However, OSKI is a sequential software package and the performance doesn't satisfy the large scale application demand. Therefore, how to parallelize and optimize the SpMV to get a better performance on GPGPU becomes a very important problem need to be further studied.

2.3. Basic Linear Algebra Kernels on GPU

In 2001, Larsen [14] used multi-texture technology to process matrix operation. With the emergence of programmable vertex, Thompson and others [1] implemented an algebra framework by vertex programming, including vector operations and matrix multiplications. The emergence of programmable pixel accelerated the research in this area. Krieger [15] used pixels to do basic algebra. Hall [16] has made a lot of optimizations for matrix multiplication using pixels, in order to take full advantage of the GPU cache.

In recent years, there are more and more researches in this area. Informed readers can refer to [17] and [18] for more details. Ding and Kennedy introduced some GPGPU-related technology optimizations in [19] and [20]. Several GPU-based algorithms for sparse matrix multiplication on emerging architectures have been proposed e.g. in [21], [22] and [23]. Especially, Buatois [23] implemented SpMV in BCSR 2×2 and BCSR 4×4 formats on GPGPU. Based on these researches, we designed and implemented an automatic performance tuning SpMV software package GOSpMV on GPGPU platform.

3. GOSpMV Overview

GOSpMV takes advantage of AMD Stream Computing (GPGPU) computation power to accelerate SPMV calculation. For the achievement of SpMV speedup on GPGPU, we implemented SpMV based on BCSR format with automatic tuning technology.

In Figure 2, it depicts the architecture of GOSpMV software package which is composed of four components. They are listed as follows.

(1) Block Based SpMV Implementation on GPGPU. In this component, it includes many different SpMV implementations with different block sizes. The details are shown in Section 3.1.

(2) Automatic Performance Tuning component. It automatically selects the suitable block size to get the best performance. Section 3.2 describes the detail of this technology.

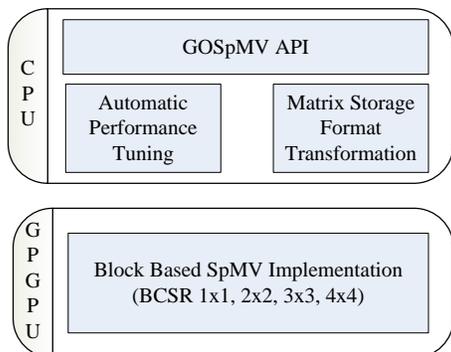


Figure 2. GOSpMV system architecture

(3) Matrix Storage Format Transformation. It supports the feature of transforming sparse matrix from CSR into BCSR storage.

(4) GOSpMV API. It exposes some APIs for user calling.

3.1. Block Based SpMV Implementation on GPGPU

On AMD Stream Computing, GOSpMV can support 4 block sizes which are 1×1 , 2×2 , 3×3 and 4×4 . The sparse matrix A is transformed from CSR into corresponding BCSR format on CPU before calculation on GPGPU.

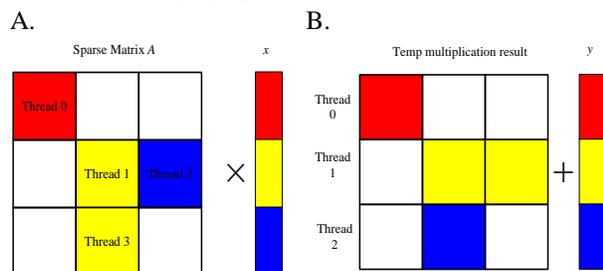


Figure 3. Block based SpMV on GPGPU. (A) Multiplication kernel. (B) Sum kernel.

To use processing units on GPGPU more efficiently, we applied fine-grained parallelism on SpMV. The method is calculating SpMV in 2 kernels. As shown in Figure 3A, the first kernel runs multiplications of each matrix block and vector x in every thread. The vector x is read by gather access, and the sparse matrix nonzero values & column index are direct mapping method.

Figure 3B depicting the second kernel, according to the index pointers of sparse matrix, it sums the first

kernel multiplication result and the vector y values. The thread is created by each row of vector y on GPGPU. Finally, we get the SpMV result.

3.2. Automatic Performance Tuning

To choose the best block size for a given sparse matrix to get the best performance on a special GPGPU, this package refers to the OSKI's solution and applies off-line/run-time heuristic search to GPGPU. The workflow is shown in Figure 4: 1) Benchmark on GPGPU is running the testing function at the install-time (Section 3.2.1); 2) Evaluation the sparse matrix and choose the best block size from the collected information on run-time (Section 3.2.2).

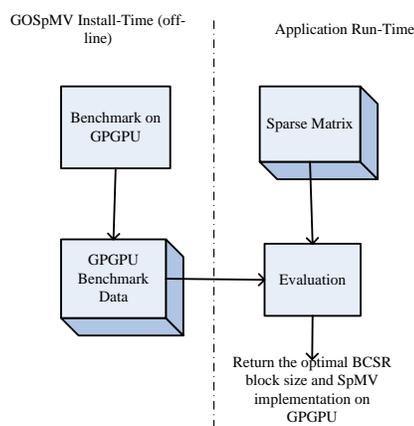


Figure 4. The workflow of GOSpMV automatic performance tuning

3.2.1. Benchmark on GPGPU. Similar to OSKI, we transform the dense matrix into sparse matrix CSR format and run SpMV with different BCSR block sizes on GPGPU. Differently, we find that the performance has relationship with the matrix size (the count of non-zero elements) nz . This relationship on the experimental platform described in section 4.1 is shown in Figure 5.

SpMV performance information on GPGPU is not only determined by the block size, we should also take the size of sparse matrix into account. The implementation of this software package is that: getting the performance (in MFLOPS) by using different sizes of dense matrix under different block sizes and store the GPGPU benchmark data for the future evaluation and selection of the best block size for a given sparse matrix.

Assume the block size is $block-format(r \times c)$, where r presents row counts in block and c presents column counts in block. The dense matrix size is nz_d , then $P_{dense}(block-format, nz_d)$ presents the benchmark data on GPGPU.

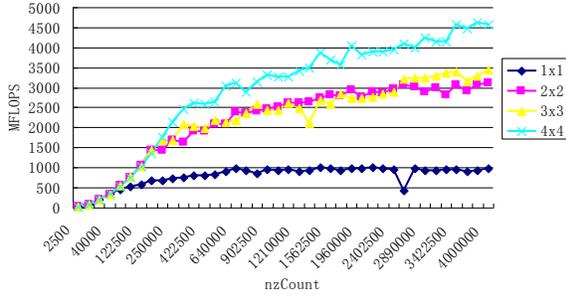


Figure 5. The performance with different block size and non-zero count

3.2.2. Run-Time Evaluation. It searches the optimal BCSR block size and implementation on GPGPU. As shown in Figure 6, for the given sparse matrix A the run-time evaluation calculates performance estimate $P(A, block-format, \sigma)$ for each block size, then chooses the BCSR $r \times c$ that makes it maximum.

Firstly, it calculates the estimated matrix fill ratio $f_{Erc}(A, \sigma)$ with sample rate σ by the method in reference [6], and nz_{EBCSR} represents the non-zeros number of sparse matrix A in this BCSR block size.

Secondly, it reads GPGPU benchmark data $P_{dense}(block-format, nz_d)$, and sets its value to block based SpMV performance estimate $P_{sp}(block-format, nz_{EBCSR})$ where nz_d is the nearest to nz_{EBCSR} .

Thirdly, $P(A, block-format, \sigma)$ equals $P_{sp}(block-format, nz_{EBCSR})$ divided by $f_{Erc}(A, \sigma)$.

Input: Sparse Matrix A , GPGPU Benchmark data $P_{dense}(block-format, nz_d)$
Output: the maximum $P(A, block-format, \sigma)$, BCSR block size
For each BCSR $r \times c$
do
 calculate estimated fill ratio $f_{Erc}(A, \sigma)$ with sample rate σ
 $P_{sp}(block-format, nz_{EBCSR}) = P_{dense}(block-format, nz_d)$, where nz_d is the nearest to nz_{EBCSR}
 $P(A, block-format, \sigma) = \frac{P(block-format, nz_{EBCSR})}{f_{Erc}(A, \sigma)}$
done

Figure 6. The GOSpMV run-time evaluation

4. Experiment and Analysis

4.1. Experimental Platform and Data Sets

Our Experimental platform equips Intel Pentium Dual Core E2160/1.8GHz, 2.0GB memory and AMD Radeon HD 3690 graphic card which peak performance is 428.8 GigaFLOPS (single precision). The software environment is Linux 2.6.24, AMD Stream SDK v1.1-beta and compiler GCC 4.2.3.

Table 1. Experimental matrices

BCSSTK17			
	row	10974	
	column	10974	
	nonzeros	219812	
	type	real symmetric	
BCSSTK28			
	row:	4410	
	column:	4410	
	nonzeros	111717	
	type	real symmetric	
epb1			
	row	14734	
	column	14734	
	nonzeros	95053	
	type	real unsymmetric	
FIDAP037			
	row	3565	
	column	3565	
	nonzeros	67591	
	type	real unsymmetric	
raefsky2			
	row	3242	
	column	3242	
	nonzeros	294276	
	type	real unsymmetric	
raefsky3			
	row	21200	
	column	21200	
	nonzeros	1488768	
	type	real unsymmetric	
twotone			
	row	120750	
	column	120750	
	nonzeros	1224224	
	type	real unsymmetric	
venkat01			
	row	62424	
	column	62424	
	nonzeros	1717792	
	type	real unsymmetric	

According to the different sparse matrix size (small, medium and large), we collected 8 sparse matrices. They are all from popular sparse matrix collections. The matrix *BCSSTK17*, *BCSSTK28* and *FIDAP037* are from Matrix Market [24], and others are from UF Sparse Matrix Collection [25]. The details of Experimental matrices are shown in Table 1. The small matrices (nonzero<100,000) are *epb1* and *FIDAP037*. The medium matrices (100,000<nonzero<1,000,000) are *BCSSTK17*, *BCSSTK28* and *raefsky2*, others are large matrices (nonzero > 1,000,000).

4.2. Experimental Results and Analysis

GOSpMV has tested SpMV performance with 4 different block sizes (1×1, 2×2, 3×3, and 4×4) and only supports single precision, so the tested matrices should be transformed into single precision. For comparison, we also test the SpMV performance of single precision on CPU with CSR format as shown in Code 1.

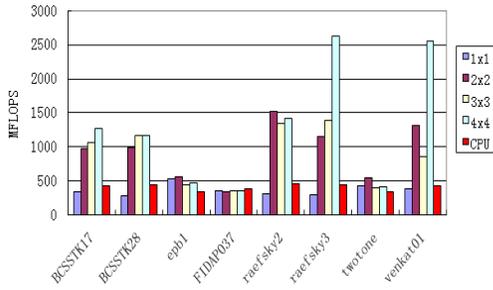


Figure 7. Performance for 8 test matrices with 1500 iterations

Figure 7 shows the SpMV performance when the iteration number reaches 1500. In most situations, the performance with block size 1×1 is the worst on GPU.

The main reason is that there is no x reused and x is accessed irregularly for block size 1×1. In contrast, the performance can be improved by better locality and reuse x for block size 2×2, 3×3 and 4×4.

With different matrix sizes and structures, we can get different speedups on GPGPU. As shown in Figure 7, the matrix *epb1* and *FIDAP037* with small size have no obvious speedup, especially for the *FIDAP037* matrix, the performance on GPGPU of which is lower than that on CPU. The sparse matrix with small size and less non-zero elements cannot make use of GPGPU very well. The sparse matrix *BCSSTK17*, *BCSSTK28*, and *raefsky2* with medium size can get speedup about 3 and the sparse matrix *raefsky3* and *venkat01* with large size can get speedup nearly 6. The reason why sparse matrix *twotone* with large size cannot get good speedup is that the block size 2×2, 3×3 and 4×4 are not suitable for its non-zero structure and its filling ratio is 3.190, 5.621 and 7.787 respectively. In BCSR format the performance is affected by too many filled zero.

We also tested the performance of different block sizes with the iteration number 100, 300, 500, 1000, and 1500. The result is given in Figure 8. Apparently, the performance can be improved by increasing iteration number, until they reach the stable performance. The reason is the cost used in kernel initialization and schedule on GPGPU can be amortized by more iteration number.

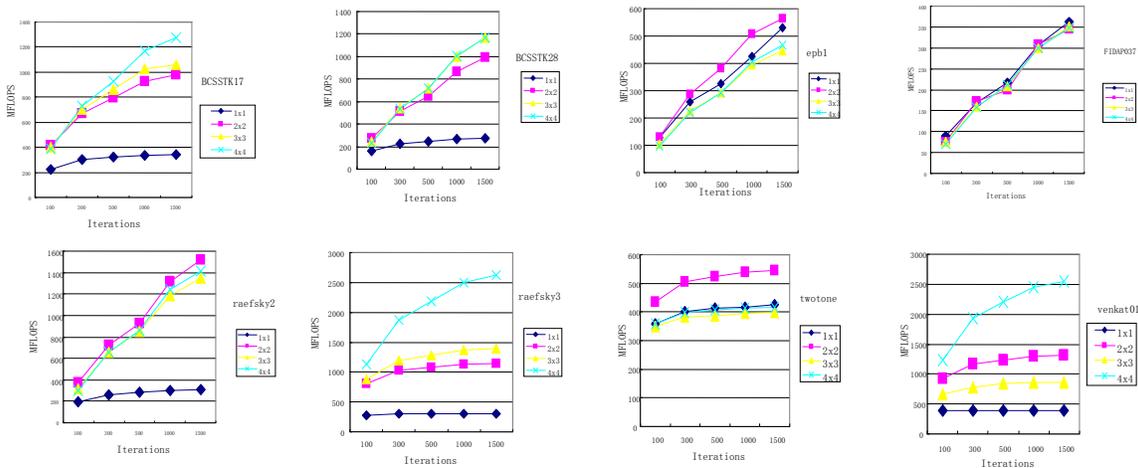


Figure 8. Performance with different testing sparse matrices and iteration number

The effect of GOSpMV automatic performance tuning is examined which result is described at Table 2. Except the matrix *raefsky2*, the rest all choose the optimal block size and get the max speedup with 5.95 in matrix *venkat01*. In matrix *raefsky2* test case, it chooses the suboptimal block size 4×4, but as Figure 7 shows, the performance is very close to that optimal block size 2×2. The average speedup can reach 3.11 for the 8 testing sparse matrices and the tuning cost is very small compared to the computing time.

Table 2. The final performance result of automatic tuning (Iteration number: 1500)

Datasets	BCSR Block size	GOSpMV Speedup	Computing time on GPGPU (s)	Tuning Time Cost (s)
BCSSTK17	4×4	2.9851	1.006	0.023
BCSSTK28	4×4	2.5957	0.564	0.01
epb1	2×2	1.6275	0.488	0.008
FIDAP037	2×2	0.9378	0.559	0.007
raefsky2	4×4	3.2471	0.602	0.015
raefsky3	4×4	5.9195	1.699	0.064
twotone	2×2	1.6193	6.75	0.118
venkat01	4×4	5.9456	2.02	0.112

5. Conclusion and Future work

When the iteration number reaches 1500, SpMV performance used GOSpMV on 8 testing sparse matrices can get average speedup with 3.11 and the max speedup is 5.95. From the experimental results we can find that if the iteration number is large enough and the sparse matrix is larger and regular, GOSpMV can get higher performance. Basically, GOSpMV is suitable for more than 300 iterations, medium and large sparse matrices and low fill ratio.

GOSpMV can always get the best block size in most situations and in rare time gets the suboptimal block size. The SpMV performance under the suboptimal block size is close to that under the best block size. As the GOSpMV uses the automatic performance tuning technique, it can be easily transformed to other new GPGPU platform and has better portability with high SpMV performance.

Now the block sizes supported by GOSpMV are limited. The relationship between performance and matrix size is not clear enough, and our GOSpMV doesn't support double precision float data because of time limit. In the future we will continue our work on above issues to get higher performance for SpMV computation on GPGPU.

Acknowledgements

We thank the anonymous reviewers for their very invaluable feedback. We want to thank AMD for the donated hardware and technical support about AMD Stream Computing, especially John Li, Gloria Le, Di-Yong Fu and Michael Houston.

This work was partially supported by the National Nature Science Foundation of China (No.60303032), Key Program of National Nature Foundation of China (No. 60533020), the National 863 Plan of China (No. 2006AA01A102 and No. 2006AA01A125), and Opening Foundation (2005-05) of State Key Lab of Networking and Switching Technology in Beijing University of Posts and Telecommunications.

References

- [1] Thompson CJ, Hahn S, Oskin M. "Using modern graphics architectures for general-purpose computing: A framework and analysis". MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society Press, 2002, 306-317
- [2] AMD Stream Computing. <http://ati.amd.com/technology/streamcomputing/>
- [3] Eun-Jin Im, Katherine Yelick, Richard Vuduc, "Sparsity: Optimization Framework for Sparse Matrix Kernels", *International Journal of High Performance Computing Applications*, Vol. 18, No. 1, 135-158 (2004)
- [4] E.-J. Im and K.A.Yelick, "Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY", *In proceedings of the International Conference on Computational Science*, volume 2073 of LNCS, pages 127-136, San Francisco, CA, May 2001.Springer.
- [5] R.Vuduc, J.W.Demmel, K.A.Yelick, S.Kamil, R.Nishtala, and B.Lee, "Performance optimizations and bounds for sparse matrix-vector multiply", *In proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.
- [6] Richard Wilson Vuduc. "Automatic Performance Tuning of Sparse Matrix Kernels." Ph.D. diss., Computer Science Division, U.C. Berkeley, December 2003.
- [7] Richard Vuduc, James W. Demmel,Jeff Bilmes, "Statistical Models for Empirical Search-Based Performance Tuning", *International Journal of High Performance Computing Applications*, Vol. 18, No. 1, 65-94 (2004).
- [8] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, Katherine A. Yelick. "Performance Modeling and Analysis of Cache Blocking in Sparse Matrix Vector Multiply". Report No.UCB/CSD-04-1335.
- [9] Rajesh Nishtala, Richard Vuduc, James W. Demmel, and Katherine A. Yelick. "When Cache Blocking of Sparse Matrix Vector Multiply Works and Why". *In Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing*, 2004.
- [10] Berkeley Benchmarking and Optimization (BeBOP) Project. <http://Bebop.cs.berkeley.edu>.

- [11] Richard Vuduc, James Demmel, Katherine Yelick, "OSKI: A library of automatically tuned sparse matrix kernels", *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, June 2005.
- [12] Optimized Sparse Kernel Interface.
<http://bebop.cs.berkeley.edu/oski/>.
- [13] E Yuan, Yun-quan Zhang and Xiangzheng Sun, "Memory Access Complexity Analysis of SpMV in RAM (h) Model", in *proceedings of 10th IEEE International Conference on High Performance Computing and Communications*, 2008, 913-920
- [14] Larsen E.S, McAllister D, "Fast matrix multiplies using graphics hardware," *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ACM, 2001, 55-55
- [15] Krüger, J. & Westermann, R. "Linear algebra operators for GPU implementation of numerical algorithms", *ACM Trans. Graph.*, ACM, 2003, 22, 908-916
- [16] Jesse D. Hall, Nathan A. Carr, J. C. H. "Cache and Bandwidth Aware Matrix Multiplication on the GPU", *University of Illinois Dept. of Computer Science*, 2003
- [17] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris. "A survey of general-purpose computation on graphics hardware". *Computer Graphics Forum*, 26(1):80–113, 2007.
- [18] Wu EH, Liu YQ. "General purpose computation on GPU". *Chinese Journal of Computer Aided Design & Computer Graphics*, 2004, 16(5): 601~612.
- [19] C. Ding and K. Kennedy, "The memory bandwidth amelioration by a compiler", in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, May 2000, pp:181-190.
- [20] C. Ding and K. Kennedy, "Improving effective bandwidth through compiler enhancement of global cache reuse", *Journal of Parallel and Distributed Computing*, January, 2004, vol. 64, no.1, pp: 108-134.
- [21] Göddeke, D.; Strzodka, R. & Turek, S. "Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations", *Int. J. Parallel Emerg. Distrib. Syst.*, Taylor & Francis, Inc., 2007, 22, 221-256
- [22] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM Press, 2005, p. 171.
- [23] Buatois, L., C. G. L. B. "Concurrent number cruncher: An efficient sparse linear solver on the GPU", in *Proceedings of the High-Performance Computation Conference (HPCC)*, 2007.
- [24] Matrix Market <http://math.nist.gov/MatrixMarket/>
- [25] UF Sparse Matrix Collection
<http://www.cise.ufl.edu/research/sparse/matrices/>