Fast Conjugate Gradients with Multiple GPUs

Ali Cevahir[1], Akira Nukada[1], and Satoshi Matsuoka[1,2]

[1] Tokyo Institute of Technology,
{ali,nukada}@matsulab.is.titech.ac.jp, matsu@is.titech.ac.jp
[2] National Institute of Informatics, Japan

Abstract. The limiting factor for efficiency of sparse linear solvers is the memory bandwidth. In this work, we utilize GPU's high memory bandwidth for implementation of a sparse iterative solver for unstructured problems. We describe a fast Conjugate Gradient solver, which runs on multiple GPUs installed on a single mainboard. The solver achieves double precision accuracy with single precision GPUs, using a mixed precision iterative refinement algorithm. To achieve high computation speed, we propose a fast sparse matrix-vector multiplication algorithm, which is the core operation of iterative solvers. The proposed multiplication algorithm efficiently utilizes GPU resources via caching, coalesced memory accesses and load balance between running threads. Experiments on wide range of matrices show that our matrix-vector multiplication algorithm achieves up to 9.9 Gflops on single GeForce 8800 GTS card and CG implementation achieves up to 22.6 Gflops with four GPUs.

1 Introduction

Recently, GPUs have attracted HPC community, because of their peak compute capability and high memory bandwidth, compared to conventional CPUs. Moreover, today's GPUs achieve relatively small cost and power consumption vs. their performance. APIs developed by manufacturers like CTM [1] and CUDA [2] made GPUs easy to program as a highly parallel multi-core coprocessor, not only for graphic applications but also for non-graphic applications.

There have been several attempts to solve unstructured sparse linear systems utilizing GPUs [4, 5, 8]. Several advantages of GPU computing is mentioned above. On the other hand, it is not easy to achieve high utilization of GPU resources. The performance of the sparse matrix-vector multiplication (MxV), which is in the core of sparse linear iterative solvers, is limited by the memory bandwidth rather than peak computation power. Although new generation GPUs are capable of being programmed for general purpose computations, they are originally optimized for graphics applications. Therefore, to achieve high performance for computations over irregularly sparse matrices, thread level parallelism and memory access methods on a set of streaming multi-processors should be carefully thought.

Another drawback of GPUs is, for most of them, lack of double precision support for floating point operations. Hence, solvers without CPU support suffer from loss of accuracy in solution. In order to achieve double precision accuracy, we adopt a mixed precision iterative refinement algorithm [6].

In this work, we implement a fast Conjugate Gradient (CG) solver with multi-GPU support. To the best of our knowledge, this is the first multi-GPU solver

for unstructured sparse systems. All basic operations of the single precision CG solver are implemented on GPUs. We propose a fast MxV algorithm.Proposed MxV algorithm achieves high utilization of GPU resources by coalesced memory accesses, caching, and load balancing between working threads.

We evaluate performance of the proposed algorithms over a wide range of well-known matrices. We compare the performance of our MxV algorithm on the GPU with CPU implementations and several naïve GPU implementations. Experiments confirm that our algorithm on the GPU is several times faster than other implementations. Performance of the parallel solver degrades to some extent due to the communication between GPUs and the CPU in each iteration. Still, we achieve average speedup of 2.47 over single GPU on 4 GPUs for big matrices in the dataset. Although we have reported results only for CG algorithm in this work, our approaches can be efficiently applied for other sparse linear methods with GPU support.

2 Background

2.1 Sparse Matrix Storage Formats

Since many of the entries in sparse matrices are zero, there is no need to explicitly store them. There are many compressed storage formats for sparse matrices [19]. In this section, we only mention two of them related to our work.

Compressed Storage by Rows (CSR) stores nonzeros of the matrices in row order. For indexing nonzeros, two arrays are used. The elements in the row pointer array point the first nonzero in each row. There are number of rows + 1 elements in this array where the last element is kept for indicating boundary of the last row. The other array stores the column indices of the nonzeros in row order. Fig. 1 depicts the pseudocode of MxV ($y = Ax$) for a CSR-stored matrix A with n rows. In the example, row_ptr and col_ind stand for row pointers and column indices, respectively.

JDS format is not as straightforward as CSR. It can result with better performance for MxV on vector processors. In order to store a matrix in JDS format, matrix rows are reordered according to the number of nonzeros in each row in decreasing order. Then, all nonzeros of the matrix are shifted to the left. Columns of the new compressed matrix are called jagged diagonals. Nonzero values of the compressed matrix are stored in an array in column order. Corresponding column indices of each nonzero in the original matrix are written in another array. One more array is kept to point the beginning indices of each jagged diagonal. Finally row permutation is stored in an array, where elements of the array that correspond to the rows of the compressed matrix, point the row number in the original matrix. Fig. 2 depicts the pseudocode for MxV of JDS-stored matrices. In the figure, perm, jd_ptr and col_ind respectively stand for permutation,

```
for i ← 0 to n − 1 do
    y[i] ← 0
    for j ← row_ptr[i] to row_ptr[i + 1] − 1 do
        y[i] ← y[i] + values[j] × x[col_ind[j]]
```

Fig. 1. Pseudocode of MxV for CSR-stored matrices

```
for i ← 0 to max_nz − 1 do
    for j ← jd_ptr[i] to jd_ptr[i + 1] − 1 do
        r ← j − jd_ptr[i]
        y[perm[r]] ← y[perm[r]] + values[j] × x[col_ind[j]]
```

Fig. 2. Pseudocode of MxV for JDS-stored matrices

jagged diagonal pointer and column index arrays. max_nz is the maximum of the number of nonzeros of each row.

2.2 Mixed Precision Iterative Refinement for Conjugate Gradients

Conjugate Gradient method is used to solve linear systems $\mathbf{Ax} = \mathbf{b}$, where matrix $\mathbf{A}$ is symmetric and positive definite. Since generally GPUs do not support double precision floating point operations, solvers on GPUs suffer from loss of accuracy in the result. Therefore, in this work we adopt a mixed precision iterative refinement algorithm for CG [6] which is based on inner-outer iteration method [9]. The algorithm explained in [6] is tested on conventional processors, but reported to be applicable on GPUs, also. Authors report that the mixed precision algorithm achieves faster solution of the same or even better accuracy compared to the full double precision solver.

Basically, mixed precision algorithm runs the preconditioned CG. However, instead of using a fixed preconditioner, preconditioner is solved using a single precision sparse iterative method, in each iteration. Operations other than the inner solver run in double precision. Single precision inner solver may also use preconditioned CG method if a preconditioner is available or any other iterative method that result in symmetric and positive definite operations. Inner solver runs for a predetermined number of iterations and takes most of the time of the overall solution.

2.3 General Purpose Computation on GPUs and CUDA

New generation GPUs can be thought as many-core stream-processing units. Taking into account the superiority of peak performance over conventional CPUs, GPUs are great resources not only for graphics processing, but also for data-parallel computing. Using GPUs for non-graphics applications is not a new idea, but with development of new APIs that hide the graphics-related interface and drastic increase in hardware performance, usability and popularity of general purpose computing on GPUs increased significantly [7].

Compute Unified Device Architecture (CUDA) is NVIDIAs new generation GPU architecture. It is also the name of the software for programming this architecture. A CUDA GPU contains number of SIMD multiprocessors. GPU has a device memory that is accessible by all processors. Each multiprocessor contains its own shared memory and read-only constant and texture caches that are accessible by all processors within the multiprocessor. CUDA API supports programming different memory types.

CUDA GPU devices are capable of running high number of threads in parallel. Threads are grouped together as thread blocks, so that each block of threads

are executed on the same multiprocessor. As a result, threads in the same block can communicate through fast shared memory.

Threads in different blocks can communicate through device memory. However, access to the device memory is very slow compared to the shared memory. Hence, device memory accesses should be refrained as possible and accesses should be coalesced to attain high performance. If memory access is organized in the right pattern, half of the threads that are scheduled to execute instructions in the same time and in the same block can access to the device memory in a single coalesced read or write instead of many simultaneous accesses. Coalescing is possible if threads access consecutive memory addresses of 4, 8 or 16 bytes and base address for coalesced access should be multiple of 16 times size of the memory type accessed by each thread.

CUDA supports single precision floating point operations based on IEEE 754 standard, with some deviations [2].

2.4 Sparse Iterative Solvers on GPU

GPU memory can be efficiently utilized for solvers where the matrix has a regular structure [10,11]. In this work, our target is to solve systems with irregular sparsity.

Bolz et al. [4] propose a Conjugate Gradient solver for unstructured matrices. They use two textures to store the matrix, one for diagonal and one for off-diagonal entries of the sparse matrix stored by CSR. To utilize memory bandwidth, blocked CSR (BCSR) is used in [5] instead of CSR. BCSR decreases number of memory fetches from the device memory to some extent, however number of elements to be multiplied increases. They achieve 1 to 6.5 Gflops CG performance on QuadroFX 5600 card with a limited dataset of 5 matrices.

Both of the above-mentioned works solve systems in single precision floating point. Göddeke et al. propose mixed precision solutions for FEM simulations on banded matrices [10, 11]. They extend the multi-grid solver to run on a GPU-enhanced cluster in [12]. Georgescu and Okuda [8] use an iterative refinement algorithm [15] to obtain double precision accuracy, for general matrices. They do not make considerable afford for faster kernel operations, instead prefer a naïve implementation based on CSR format.

3 GPU-Enhanced Conjugate Gradient Solver

We implement the mixed precision algorithm explained in [6] and summarized in Section 2.2. Core operations of single precision inner solver run on the GPU, while double precision refinement iterations run on the CPU. We implement CG for inner solver, assuming that we have no preconditioner readily available.

CG consists of several kernel operations: MxV, SAXPY, vector dot product, norm and scalar operations. Since MxV dominates the running time, fast implementation of MxV is required for faster CG.

3.1 Efficient Sparse Matrix-Vector Multiplies on GPU

We propose an efficient MxV algorithm on GPUs based on JDS storage and CSR-like multiplication. The matrix is stored in JDS format to achieve coalesced

```
mult ← 0
for i ← 0 to nz_count[t] − 1 do
    j ← jd_ptr[i] + t
    mult ← mult + values[j] × x[col_ind[j]]
done
y[perm[t]] ← mult
```

Fig. 3. CSR-like multiplication of JDS-stored matrix.

reads from the device memory. Each GPU thread multiplies one row of the matrix and computes one output vector element. The proposed MxV procedure for each GPU thread t is depicted in Fig. 3. Note that, for each thread to realize multiplication we need to have one more array (called nz_count in the figure) to store number of nonzeros of each row.

In this multiplication scheme, consecutive threads access to the consecutive indices in arrays values, col_ind, nz_count and perm. So, reads from these arrays can be coalesced, instead of many simultaneous reads. Note that, as mentioned in Section 2.3, to achieve coalescing base addresses of coalesced reads should be multiple of 16 times size of the data type to be read. Namely, elements of jd_ptr should be multiple of 16. We pad zeros to arrays values and col_ind for each jagged diagonal to have multiple of 16 entries.

Unfortunately, writing to the output vector y cannot be coalesced because of the irregular access caused by perm array. Still, unlike JDS multiplication given in Fig. 2, CSR-like multiplication has an advantage of writing the output vector only once.

Since in JDS format matrix rows are sorted according to their nonzero count in decreasing order, better computational load balance is naturally obtained. The variation of nonzero counts between threads within the same block is smallest.

In JDS format, row indices are not consecutively ordered. To access values in row t, indices should be calculated using jd_ptr array. Many accesses to this array may be costly if it is deployed to the device memory. We place this array to read-only constant cache to avoid slow reads. Reading from constant cache is as fast as reading from registers, if active threads in the same block read the same address. In case of cache miss, cost of reading from constant memory is equal to reading from device memory. Size of the cached array jd_ptr is equal to the maximum of the number of nonzeros of rows. For all of our experimental data, this array completely fits into constant cache, hence no cache miss occurs.

We bind x array to the texture cache in our implementation.

3.2 Other Operations

Not only MxV, but all operations of the inner CG solver other than scalar division operation are efficiently implemented on the GPU. Dot products and norms are implemented as in the parallel reduction example of NVIDIA's CUDA SDK [13], in $\log n$ steps, where n is the size of the vectors in computations. Each output element of SAXPY operation is calculated by a different thread.

3.3 Multi-GPU Algorithm

CUDA supports multiple GPUs run together for an application. Compared to main memory, GPUs have limited device memory. For applications which require

a lot of storage, device memory limitations may be a bottleneck for GPUs. For this reason, sometimes running algorithms on multiple GPUs is not only required for faster applications but to overcome memory bottleneck.

We propose a data-parallel CG algorithm to run on multiple GPUs and a CPU located on the same board. Rows of the matrix and corresponding vector entries are distributed amongst GPUs. Since MxV takes most of the iteration time, we assign nonzeros of the matrix equally to each GPU, so that loads of MxV is balanced amongst GPUs.

CPU creates a thread for each GPU and coordinates the communication amongst them. In every iteration, each GPU communicates with the CPU for them to exchange input vector entries of the matrix-vector multiply. Host CPU holds a global array, where each GPU writes to and reads from for communicating vector entries. When inner solver terminates, solution vector computed by the inner solver is copied to the CPU and refinement iteration on the CPU begins.

Other than above-mentioned communications, scalars are communicated between GPUs and the CPU to compute global results of the locally computed values. In the resulting algorithm, threads synchronize in three points: once before MxV to exchange input vector of the multiplication, and two times to compute scalars.

4 Experimental Results

We evaluate performance of proposed MxV and CG methods on single and multiple GPUs. In our experiments, an AMD Phenom 9850 2.5 GHz Quad-Core processor, 4 GB main memory and four GeForce 8800 GTS 512 GPU cards are used in the hardware platform. CUDA version 1.1 is used for coding on Linux 2.6.23 OS. Pthread library on C language is used for CPU threading.

42 matrices that are symmetric and positive definite with real value entries from Sparse Matrix Collection of University of Florida [3] are used for performance evaluation. Matrix dimensions vary from 1,440 to 1,585,478 and number of nonzeros vary from 46,270 to 55,468,422.

4.1 MxV Performance

We compare performance of the proposed algorithm with CSR implementations on the CPU and one SMP node of the TSUBAME supercomputer [16]. To make fair comparisons with the quad-core CPU, we implement CSR multiplication on the CPU using 4 threads. One node of TSUBAME has 8 AMD 2.4 GHz Opteron dual core processors, hence we use 16 threads in our implementation on the SMP node. To demonstrate the validity of the algorithmic improvements, we also compare our algorithm with basic JDS and CSR implementations on the GPU. Each thread block contains 32 threads for all implementation on the GPU. In the CSR implementation, each GPU thread multiplies one row of the matrix. For JDS implementation, each thread computes one element of the matrix and we pad zeros to matrices to achieve coalescing, as explained in Section 3.1. Since output vector y is accessed number of nonzero times for basic JDS multiplication,

accesses on this array in irregular order drastically degrades the performance. Hence, we let each thread t multiplying a nonzero in row r to write r^{th} index of a temporary array y_temp. By this way, writes on y_temp can be coalesced. In the end of the multiplication, y_temp is reordered into y, using the row permutation array.

Comparison results are given in Gflops in Fig. 4. Matrices on x axis are sorted according to the number of nonzeros they contain. GPU-Proposed stands for our proposed multiplication algorithm based on JDS storage and row multiplication. GPU-CSR and GPU-JDS stand for CSR and JDS implementations on GPU, respectively. TSUBAME-1 node stands for 16 core SMP implementation on one TSUBAME node and Quad core CPU stands for our single CPU implementation.

During MxV, for each nonzero, one multiplication and one addition operation is executed. Therefore, we calculate flops by dividing two times number of nonzeros by execution time. SMP, quad core CPU, CSR-based GPU, JDS-based GPU and our algorithm respectively achieve 0.84, 1.16, 1.37, 2.7 and 6.52 Gflops, on average for matrices in the dataset.

Variation of the performance of our algorithm and JDS-based implementation on test matrices is greater than other implementations. Utilization of GPU resources is lower for smaller matrices, hence the algorithm is slower. This is more obvious for JDS-based implementation. On the contrary, with the increase in matrix size, SMP implementation becomes slower, due to the insufficient cache for matrix data. Dramatically low performance of the SMP node of 16 cores indicates the importance of memory bandwidth utilization for MxV.

Matrix structure greatly affects our algorithm's performance. For sparser matrices such as G3_circuit and thermal2 (4.8 and 7 nonzeros per row, respectively), and denser matrices such as exdata_1 and nd24k (378.2 and 398.8 nonzeros per row, respectively), algorithm is slower than the average. The algorithm is also slower for matrices whose nonzeros are distributed over the whole matrix, such as F1. This is due to the large number of cache misses on the input vector of multiplication. On the other hand, algorithm is faster for matrices whose consecutive rows (columns) share many columns (rows). For instance, s3dkq4m2, whose nonzeros are ordered around the diagonal, is the fastest in our dataset.

Note that the main reason behind the outstanding performance of our algorithm is memory coalescing. In our experiments, we found out that CSR imple-

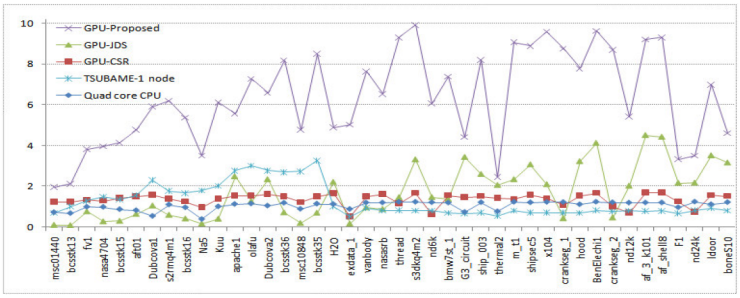

Fig. 4. MxV performance comparison of the proposed algorithm.

mentation on GPU performs slightly better than our algorithm, if we do not pad 0s to jagged diagonals hence coalesced memory reads do not occur.

Effective memory bandwidth of the proposed MxV algorithm is 38.9 GB/s for matrices in the dataset, on average. The GPU used in experiments has maximum memory bandwidth of 64 GB/s, that is, bandwidth utilization of our algorithm is 61%.

4.2 CG Performance

In this section, we evaluate the performance of single and multiple GPU CG algorithms and CG on the CPU. Performance of inner CG solver iterations of the mixed precision algorithm is given in flops. We do not make convergence analysis of the mixed precision algorithm, since exclusive analysis is already done in [6] and the algorithm is shown to be faster than double precision CG on conventional CPUs while not sacrificing accuracy of the solution. Also, we do not observe any increase in number of iterations for GPU-enhanced algorithms to achieve same accuracy with CPU implementation of both inner and outer solver. Since double precision operations occupy only a small portion of the overall execution time, the performance of the mixed precision algorithm increases speeding up the single precision inner solver. We implement double precision outer iterations on the CPU. On average, one iteration of double precision CG achieves 1.42 Gflops for matrices in the dataset.

Performance of the inner CG solver on different platforms is given in Fig. 5. The chart on the left depicts CG performance in Gflops and the chart on the right depicts the speedups of the multi-GPU algorithm over single GPU algorithm. For smaller matrices in the dataset, communication cost between CPU and GPU dominates the overall execution time of multi-GPU algorithm, hence employing more GPUs do not increase performance of the solver. For this reason, in the figure, we omit results for matrices that have less than 5 million nonzeros. CG performance is 6.02, 9.58 and 14.84 Gflops with single GPU, 2 GPUs and 4 GPUs, respectively, on average. This means that 2 GPUs speeds up single GPU algorithm by 1.59 and 4 GPUs by 2.47.

It is interesting to observe the big variation on the speedups. We observe speed down for some matrices, while for one matrix we observe superlinear speedup. The most important factor affecting the performance of the multi-GPU algorithm is number of communicating input vector entries between the CPU and GPUs. Some of the other factors affecting the performance are the balance of vector sizes between processors and the cache affect.

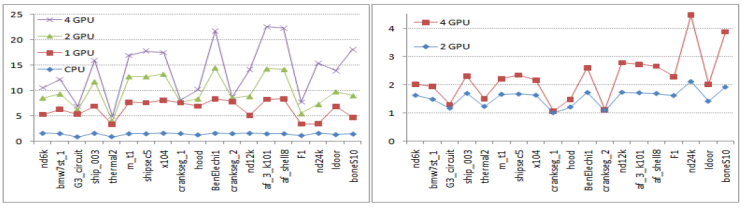

Fig. 5. CG (inner solver) performance. Left: comparison of single and multi-GPU algorithms with CPU implementation Right: Multi-GPU speedups over single GPU

For the densest matrix in the dataset, nd24k, we observe superlinear speedups, while for the sparsest matrix, G3_circuit, we cannot observe any speedup. The imbalance of vector sizes of nd24k on 4 GPUs is 24%. Maximum sending GPU sends 86 KB to the CPU and maximum receiving GPU receives 166 KB from the CPU. Since MxV dominates the total CG time, imbalance of vector sizes become negligible. On the other hand, for the sparsest matrix G3_circuit imbalance of vector sizes is 6%, maximum sending GPU sends 730 KB and maximum receiving GPU receives 732 KB, for 4 GPUs. Here, GPU communication dramatically affects the performance, since computation count per communicating data is very low. For crankseg_2, where we do not observe any speedups for this matrix on 4 GPUs, the imbalance of vector sizes is 35%, maximum sending GPU sends 86 KB and maximum receiving GPU receives 207 KB receives almost all vector entries that it does not compute. The density and number of nonzeros of this matrix is about half of nd24k. Still, we cannot explain the performance difference between these two matrices by just the communication and vector imbalance. The speed down of the other crankseg matrix implies that matrix structure also affects the performance difference among matrices. We found out that the variation of number of jagged diagonals across matrices assigned to different GPUs is incomparably high for crankseg matrices, where for nd24k, there is almost no variation. Sparsity patterns of distributed matrices across GPUs are too different for crankseg matrices, so that even we assume no communication, ideal speedup is impossible due to the difference in cache utilization of parallel GPUs. For these types of matrices, a clever row distribution algorithm emerges.

5 Conclusion and Future Work

In this work, we have demonstrated efficient utilization of GPUs for solution of general sparse symmetric linear systems with double precision solution accuracy. We have implemented a mixed precision CG solver on multiple GPUs and evaluated its performance on a wide range of matrices. The performance of the proposed algorithm reveals that stream processing on modern GPUs can be useful for increasing memory bandwidth utilization for sparse linear solvers. The proposed MxV algorithm, which is the most time-consuming operation of the CG solver, utilizes constant and texture caches, as well as coalesced memory reads from the device memory. As a result, we achieve the fastest MxV implementation for unstructured sparse matrices on the GPU, to the best of our knowledge.

Number of cache misses on input vector considerably affects the run time of MxV. It is very difficult to find algorithms to increase the cache utilization for input vector. There are some works dedicated to decrease the number of cache misses for CSR multiplication, which is an NP-complete problem [17, 18]. We plan to study on better cache utilization for the input vector of the proposed MxV algorithm in future.

Our multi-GPU algorithm achieves 14.8 Gflops for CG on 4 GPUs, on average. Communication between GPUs and the CPU significantly degrades the performance of the multi-GPU algorithm. Performance of the multi-GPU algorithm can be further increased by direct GPU to GPU communication. We await

CUDA support for direct inter-GPU communication instead of communication through the CPU.

In the future, we plan to study on scalable implementation of the parallel algorithm to run on a GPU cluster. Although we demonstrate results only for the CG solver in this work, proposed techniques can also be applicable for other symmetric or asymmetric iterative solvers.

References

1. ATI CTM Guide Technical Reference Manual. Advanced Micro Devices, Inc. (2006)
2. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. NVIDIA Corporation (2007)
3. University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/
4. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. ACM Transactions on Graphics (2003)
5. Buatois, L., Caumon, G., Lévy, B.: Concurrent Number Cruncher: A GPU Implementation of a General Sparse Linear Solver. International Journal of Parallel, Emergent and Distributed Systems, to appear
6. Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P., Tomov, S.: Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. ACM Transactions on Mathematical Software (2008).
7. Blythe, D.: Rise of the Graphics Processor. Proc. of the IEEE **96(5)** (2008)
8. Georgescu, S., Okuda, H.: GPGPU-Enhanced Conjugate Gradient Solver for Finite Element Matrices. Proc. iWAPT, Tokyo (2007)
9. Golub, G. H., Ye, Q.: Inexact Preconditioned Conjugate Gradient Method with Inner-Outer Iterations. SIAM J. on Scientific Computing **21(4)** (2000) 1305–1320
10. Göddeke, D., Strzodka, R., Türek, S.: Accelerating Double Precision FEM Simulations with GPUs. Proc. ASIM 2005
11. Göddeke, D., Strzodka, R.: Performance and Accuracy of Hardware Oriented Native-, Emulated-, and Mixed Precision Solvers in FEM Simulations (Part 2: Double Precision GPUs). Tech. Rep. Ergebnisberichte des Instituts für Angewandte Mathematik Nr. 370, TU Dortmund (2008)
12. Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Wobker, H., Becker, C., Türek, S.: Using GPUs to Improve Multigrid Solver Performance on a Cluster. International Journal of Computational Science and Engineering (2008)
13. Harris, M.: Optimizing Parallel Reduction in CUDA. NVIDIA Dev. Tech. (2007)
14. Krüger, J., Westermann, R.: Linear Algebra Operators for GPU implementation of Numerical Algorithms. ACM Transaction on Graphics **22(3)** (2003) 908–916
15. Martin, R. S., Peters, G., Wilkinson, J. H.: Iterative Refinement of the Solution of a Positive Definite System of Equations. Numerische Mathematik **8** 1966 203–216
16. Matsuoka, S.: The Road to TSUBAME and Beyond. Petascale Computing: Algorithms and Applications, Chapman & Hall Crc Comp. Sci. Series (2008) 289–310
17. Pichel, J. C., Heras, D. B., Cabaleiro, J. C., Rivera, F. F.: Improving the Locality of the Sparse Matrix-Vector Product on Shared Memory Multiprocessors. Proc. PDP'04 (2004)
18. Pinar, A., Heath, M. T.: Improving the performance of Sparse Matrix-Vector Multiplication. Proc. SC'99 (1999)
19. Saad, Y.: SPARSEKIT: A Basic Tool for Sparse Matrix Computation. Tech. Rep. CSRD TR 1029, Univ. of Illionis, Urbana, IL (1990)