GPU Acceleration for FEM-Based Structural Analysis

Article *in* Archives of Computational Methods in Engineering · May 2013

Impact Factor: 3.68 · DOI: 10.1007/s11831-013-9082-8

3 authors:

Serban Georgescu
Fujitsu Ltd.
10 PUBLICATIONS **64** CITATIONS
SEE PROFILE

Peter Chow
Fujitsu Laboratories of Europe Ltd.
39 PUBLICATIONS **284** CITATIONS
SEE PROFILE

Hiroshi Okuda
The University of Tokyo
112 PUBLICATIONS **476** CITATIONS
SEE PROFILE

Noname manuscript No.
(will be inserted by the editor)

GPU Acceleration for FEM-based Structural Analysis

Serban Georgescu · Peter Chow · Hiroshi Okuda

Received: 15 April 2013 / Accepted: 16 April 2013 / Published online: 30 April 2013

Abstract Graphic Processing Units (GPUs) have greatly exceeded their initial role of graphics accelerators and have taken a new role of co-processors for computation-heavy tasks. Both hardware and software ecosystems have now matured, with fully IEEE compliant double precision and ECC correction being supported and a rich set of software tools and libraries being available. This in turn has lead to their increased adoption in a growing number of fields, both in academia and, more recently, in industry. In this review we investigate the adoption of GPUs as accelerators in the field of Finite Element Structural Analysis, a design tool that is now essential in many branches of engineering. We survey the work that has been done in accelerating the most time consuming steps of the analysis, indicate the speedup that has been achieved and, where available, highlight software libraries and packages that will enable the reader to take advantage of such acceleration. Overall, we try to draw a high level picture of where the state of the art is currently at.

Serban Georgescu
Fujitsu Laboratories of Europe Ltd.
Hayes Park Central, Hayes UB4 8FE, United Kingdom
Tel: +44 (0) 20 8606 4846
Fax: +44 (0) 20 8606 4539
E-mail: Serban.Georgescu@uk.fujitsu.com

Peter Chow
Fujitsu Laboratories of Europe Ltd.
Hayes Park Central, Hayes UB4 8FE, United Kingdom
E-mail: Peter.Chow@uk.fujitsu.com

Hiroshi Okuda
Dept. of Human and Engineered Environmental Studies,
Graduate School of Frontier Sciences,
The University of Tokyo
Kashiwanoha 5-1-5, Kashiwa, Chiba 277-8563, Japan
E-mail: okuda@race.u-tokyo.ac.jp

Keywords GPGPU · CAE · FEM · Structural Analysis · Matrix Solvers

1 Introduction

In the last decades, structural testing and analysis has become a basic requirement across a wide range of industries. From the structural analysis of buildings and vehicles to the testing of the effects of loads on mobile phones and computers, Finite Element Method based Structural Analysis (FEM-SA) is now a common tool spanning most engineering disciplines. With a trend for increasing model size and complexity and with FEM-SA being just one element in the design and optimization process, there is an ever increasing need for acceleration. In order to achieve this acceleration, code developers in both academia and industry are looking towards accelerators like Graphic Processing Units (GPUs) that provide increased processing power and memory bandwidth while at the same time reducing the overall power consumption.

Considering the potential benefits brought by GPU acceleration, it is not surprising that the last five years have shown an almost exponential increase in the number of publications in the field of GPU-accelerated FEM-SA and in the number of software packages, both free and commercial, that enable the user to take advantage of such acceleration. Although most effort is concentrated around the acceleration of the linear matrix solver, generally the most compute-intensive and time-consuming part of the analysis, significant progress has also been made in the acceleration of other parts of the process, like the computation and assembly of the global stiffness matrix. It is thus the goal of this review to survey the work that has been done in all stages of the FEM-SA process and to draw a high level picture of the current state of the art. In particular, this review aims to highlight the progress for each stage, the speedup that has been achieved or can be achieved and, where available, software tools that can help the reader realize such levels of acceleration in his or her own work.

After a brief introduction to computing on the GPU in Section 2, this review follows the flow of a typical FEM-SA, shown in Fig. 1. The CAD model, created using specialized CAD software, first undergoes a pre-processing step, in which the model is converted to a form suitable for FEM-SA. This process might contain steps like shape simplification, where small features of the CAD model, which are not important for the analysis, are removed. Following, a mesh of the model is created. GPU acceleration in the pre-processing stage is possible to some extent. Previous work in FEA-SA oriented accelerated meshing is surveyed in Section 3.1 while accelerated model simplification is tackled in section 3.2. Following, solver acceleration, the most important part of this review, is surveyed in Section 4. In an implicit code, the type considered here, the solver computes the element stiffness matrices and performs an assembly procedure, which results in the creation of a large system of equations. Acceleration in this creation phase is overviewed in Section 4.1. The system of equations is then solved by a matrix solver, which can be of one of two types: iterative, analyzed in Section 4.2 or sparse direct, in Section 4.4. Work on preconditioners, indispensable for most FEM-SA, is also reviewed in Section 4.3. As a complement to the aforementioned surveys, an extensive list of software tools that can be used to obtain GPU acceleration is given in a compact form in Section 5. Finally, a high-level picture of the current state of the art in GPU-accelerated FEM-SA is provided, together with our conclusions, in Section 6.

2 GPU computing and FEM-SA

GPU computing or GPGPU is the use of GPUs as accelerators for general purpose computation as opposed to computer graphics, the original target of GPUs. Since the year 2007, when NVIDIA introduced their Compute Unified Device Architecture (CUDA)[48], computational resources available on the GPU can be directly used for general computation. Running CUDA applications requires a CUDA-capable GPU (all recent NVIDIA GPUs are CUDA capable), an appropriate driver and a toolkit (which contains the CUDA compiler and required libraries), all of which can be obtained for free. In 2008, the OpenCL[58] framework was launched as a vendor-independent alternative to CUDA. OpenCL has the advantage of being open and supported across multiple architectures. In particular, while software accelerated with CUDA can only work with GPUs from NVIDIA, software accelerated via OpenCL can be used with GPUs from multiple vendors, including both NVIDIA and AMD. In the same way as for CUDA, all the software necessary for OpenCL acceleration can be obtained at no additional cost. GPUs from both NVIDIA and AMD can be split into three categories: consumer, professional and High Performance Computing (HPC). Consumer class GPUs (GeForce series from NVIDIA and Radeon series from ATI) are relatively cheap, have high memory bandwidth and single precision performance but have much lower double precision performance and lack features like ECC memory correction. At the opposite end, HPC class GPUs (Tesla series from NVIDIA and FirePro series from AMD) are capable of much faster double precision computation and are more reliable.

While CUDA and/or OpenCL capable GPUs can now be found in any laptop with discrete graphics, the general target for FEM-SA users are workstations and departmental clusters. Depending on how many full width (16x) PCI Express slots are available and considering the size, power and cooling constraints, in general between one and four GPUs can be added to each workstation or cluster node. To get an idea of what this means in terms of performance, consider that adding two last generation NVIDIA Tesla K20 GPUs to a high-end 16 core Sandy Bridge workstation or cluster node will increase its double precision floating point performance from ∼350 GFLOP/s to ∼2.6 TFLOP/s (that is, a 7.5x increase) and the aggregate theoretical peak memory bandwidth from ∼100GB/s to ∼500GB/s (that is, a 5x increase). Hence, ideally, we would expect a fully GPU-accelerated FEM-SA to get a speedup of a factor from 5x to 7.5x. However, unfortunately, this is rarely the case.

Having the hardware is just half of the way. The most challenging part is the development of software capable of taking advantage of GPU acceleration. Fortunately, at present the GPGPU software ecosystem has reached a stage where a FEM-SA developer can get away with writing little or no CUDA or OpenCL code and rely on libraries and software packages instead. The usual approach taken by developers of general purpose FEM-SA software is to accelerate the matrix solver, as this in general proves to be the most time consuming step in the analysis. This is the case for most major GPU-accelerated FEM-SA packages currently offered by Independent Software Vendors (ISVs), like ABAQUS, ANSYS Mechanical, MSC Nastran and so on. There are two ways such acceleration is usually achieved: by replacing the entire matrix solver with one

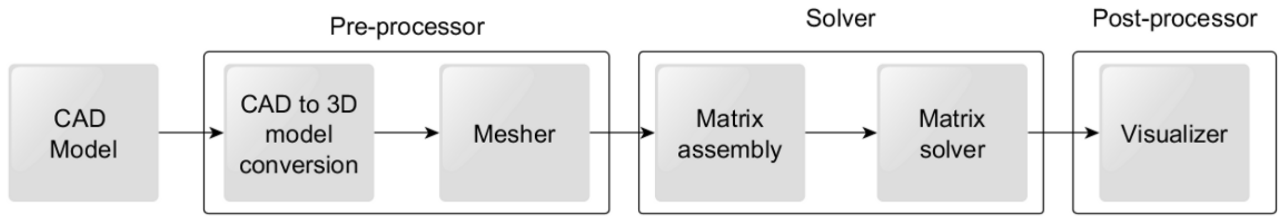

Fig. 1 The flow of a typical FEM-SA

capable of GPU acceleration or by accelerating the current matrix solver using GPU targeted libraries of algebraic operations. We remark that for both of these tasks the use of CUDA instead of OpenCL provides the developer with a much broader set of choices of software tools to use.

The first and easiest way of adding support for GPU acceleration in a FEM-SA solver is to replace the entire matrix solver with one capable of GPU acceleration. In previous work [20] we have reported the acceleration of the open source FEM-SA solver FrontISTR by simply replacing the built-in Conjugate Gradients (CG) solver with the PETSc package [44], which is now capable of GPU acceleration. Such work requires little familiarity with GPU programming. A list of packages which can be used for this purpose will be given later, in Section 4.4.

The second way involves developing solvers from building blocks which are targeted towards the GPU. For FEM-SA, developing an un-preconditioned GPU-accelerated CG solver is not difficult since all the required pieces, that is dot products, vector updates and the Sparse Matrix Vector Multiplication (SPMV) routines can be found in libraries like CUBLAS and CUSPARSE, both already part of the CUDA Toolkit or CUSP[15]. Note that in order to obtain good performance on the GPU, an efficient matrix format like the Hybrid format or the Blocked-Hybrid format are required. For details about sparse matrix formats on GPUs we direct the reader to [4,11].

3 Acceleration in the Pre-processing stage

Performing a FEM-SA requires the creation of a computational mesh. This is done using automatic or semi-automatic meshers that can construct 3D tetrahedral or hexahedral meshes out of arbitrarily shaped CAD parts or assemblies of parts. For complex models like the ones found in industry, mesh generation is a very time consuming, computationally heavy step that would greatly benefit from the computational power provided by a GPU. One major issue concerning automatic mesh generation from CAD models is that, in many cases, the available CAD models are as manufactured, having been created with the objective of manufacturing (i.e., molding or increasing the product's aesthetic appeal) and not with the one of being used for FEM-SA. As such, they contain many small features like small holes, fillets, small protrusions and the like. These, although having a negligible impact on the results of the analysis, make the CAD model difficult to mesh and the resulting FEM mesh unnecessarily large and complex. In order to remove such undesired features, simplifications algorithms are being used. In the following two sections we review what has been achieved in the GPU acceleration of meshing and model simplification while we provide a summary in Table 1.

3.1 Meshing

Several meshing algorithms are currently available, however almost all of them are restricted to serial execution. As parallel mesh generation, even in the context of CPUs, is still an open issue, porting such algorithms to the GPU is a very challenging task. Due the difficulty of parallel mesh generation, to this date no attempt to accelerate 3D mesh generation with accelerators like the GPU has been reported.

The closest publications on the subject are [54] followed by [53], where a 2D Delaunay triangulation and constrained 2D Delaunay triangulation, respectively, have been accelerated using a GPU. The speedup obtained for the 2D Delaunay triangulation in [54] is very limited, with a NVIDIA GeForce 8800 GTX being at best 55% faster than an Intel Core2 Duo at 1.86GHz and actually slower for some of the test cases. However, the constrained 2D Delaunay triangulation reported in [53] is able to deliver more than an order of magnitude acceleration for a last generation NVIDIA GeForce GTX 580 GPU over an Intel Core i7 2600K CPU at 3.4GHz.

Table 1 Overview of GPU accelerated FEM-SA pre-processing. The first and second columns show the pre-processing stage. Relevant references for each stage are given in column 3 while acceleration reported for that stage is given in column 4. Finally, available relevant software, if any, for that stage, is given in column 5.

Pre-processing stage	Type	References	Acceleration	Software
Meshing	2D	[54]	1.55x vs 1 core [54]	n/a
	Constrained 2D	[53]	14-28x vs 1 core[53]	n/a
	3D	n/a	n/a	n/a
Model simplification	CAD simplification	n/a	n/a	n/a
	Mesh simplification	[6,29,16]	8x vs 1 core [29]	n/a

3.2 Model simplification

Model simplification algorithms approach the simplification problem in three ways.

The first way works directly at the CAD level, by making transformations to the underlying geometry. From a computational point of view, this type of transformations is the most efficient, since they are performed before starting the meshing process. However, their implementation is complex and usually takes the form of a rule-base system, the structure of which is very unsuitable for GPU computations. Hence, no reports of GPU acceleration for this type of methods exist to this date.

The second way is to start from the opposite direction, with a complete 3D FEM mesh made prior to removing the unwanted features. Features can be removed through operations like the collapsing of the faces of tetrahedrons. This is the least efficient method, since it only reduces the time spent in the FEM solver but not in the meshing part. As for the previous approach, no attempt of GPU acceleration has been reported so far.

Finally, the third way is in between the previous two, with the algorithm starting from a polygonal description of the surface of the CAD model (i.e., from a surface mesh of the skin of the model) followed by the application of certain simplification operations that preserve desired quantities relevant to the analysis. The only work targeting FEM-SA reported so far is [29], where shape simplification using an iterative vertex removal algorithm is accelerated by porting the two most expensive operations, the validation of the removal criteria and the re-meshing of the remaining hole, to a GPU. By using the GPU, an acceleration of up to 8x (for large models) is obtained for a NVIDIA GeForce 7800GT GPU as compared to a AMD Athlon 4400+ CPU at 2.2GHz. Similar work but with a focus outside of the field of CAE was first reported in [6] where the computation of the distance between the original mesh and the simplified one is computed on the GPU, in [16] which accelerated a vertex clustering algorithm and, very recently, in [57] where mesh simplification by marking and inverse reduction is accelerated using a GPU.

4 Acceleration in the Solver stage

After a FEM mesh has been created from the CAD model, the FEM solver uses the connectivity information, material properties and boundary conditions to build the global stiffness matrix, also called the global matrix or stiffness matrix. The global stiffness matrix contains the contributions of all individual elements and, after being assembled together with the load vector (the right-hand side), it will be solved by a matrix solver. If the analysis is static and linear, the matrix assembly is done only once. However, for non-linear cases or when the matrix coefficients are time-dependent, the assembly has to be done a large number of times, becoming a costly part of the solving procedure.

Solving linear systems of equations, which are large and sparse, is arguably the most time consuming part of the FEA-SA process. Solvers used for this kind of problems fall into two categories: iterative solvers and sparse direct solvers. Sparse direct solvers are usually the solvers of choice in industry, as problems that are being solved on a daily basis are small to medium in size. Moreover, direct solvers do not suffer from the convergence problems that plague iterative solvers when the matrix to be solved is either very ill-conditioned or contains zeros on the main diagonal, as it is the case in simulations that involve mechanical contacts. On the other hand, iterative solvers are much more scalable, both in the sense of algorithmic complexity and in the sense of parallelization. Hence, iterative solvers are usually the solvers of choice for large scale problems where the use of sparse direct solvers is not feasible. An important topic in conjunction with iterative solvers is the one of preconditioners, which are almost indispensable when dealing with ill-conditioned and hard to converge problems like the ones usually encountered in industry.

The most time consuming parts for iterative solvers are the Sparse-Matrix Vector Multiplication (SPMV) operation and the preconditioning. For general unstructured meshes, the resulting matrices are unstructured hence, due to poor data locality, the performance of the SPMV operation is only a small percentage of the total machine peak. Specialized data formats that exploit patterns found in the structure of the matrix (the placement of non-zero elements, like diagonals or dense blocks) are usually used to make this operation more efficient. Data formats aside, the SPMV operation is inherently parallel thus its implementation on multicore CPUs, GPUs and computer clusters does not pose great difficulty. Unfortunately, this is not true in the case of factorization-based preconditioners (e.g., ILU/IC) where the substitution steps necessary for their application are serial in nature. As such, the development and efficient implementation of powerful parallel preconditioners is currently the most challenging aspect in porting iterative solvers to massively parallel architectures like the GPU.

In the following four subsection we review the work that has been done and the speedups that have been achieved by using GPUs to accelerate the creation and assembly of the global stiffness matrix, iterative solvers and preconditioners and sparse direct solvers. A summary is provided in Table 2.

4.1 Element and global stiffness matrix computation

The building of the global stiffness matrix is done in two steps. First, an element stiffness matrix is computed for each individual element. The computation of the element stiffness matrices is relatively simple for low order standard element types, as the values of the integrals needed to compute the elements of the matrix can be pre-computed. However, the operation can get very costly for elements with high orders and with complex shapes, like curved boundaries, or where, for various reasons, the previously mentioned integrals cannot be pre-computed. Next, following a mapping, the element stiffness matrices are assembled (summed together) into the global stiffness matrix. This operation is data intensive, and care must be taken when accessing the data in a regular fashion and to avoiding race conditions which can happen when adding the contribution of multiple elements to the same matrix entry.

There are few reports on the GPU acceleration of the element matrix computation and stiffness matrix assembly, the primary reason being that, for most FEM-SA solvers, the time spent in this step is actually one or two orders of magnitude shorter than the time spent in the matrix solver. This is especially the case for the computation of the element stiffness matrices, where element types (e.g. hexahedral, tetrahedral or prism) of 1^{st} and 2^{nd} order are usually employed. Moreover, for a general purpose FEM-SA solver, supporting many element shapes and material types, the effort required to port the corresponding code to the GPU is usually much greater than what is required the port the matrix solver.

The acceleration of the computation of the element stiffness matrix has been attempted in [51, 42] in an FEM-SA setting, however there are several other such reports in the the field of FEM-based electro-magnetics where higher order elements are more often employed. The type of computations involved in calculating the element stiffness matrices is inherently parallel, since there is no data dependence between the elements, so all element stiffness matrices can be computed in parallel. If we consider 2^{nd} order elements, as often employed in FEM-SA, a speedup of around 4x for a NVIDIA GeForce GTX 285 GPU versus one core of an Intel Xeon E5520 CPU at 2.26GHz was reported in [51] for a 2D problem. As both CPU and GPU performance scales linearly with the number of cores and multi-processors, we can extrapolate that if the all four cores of the CPU would have been used, the performance of the GPU would be similar to that of one CPU, which shows that there is no much to be gained by accelerating the computation of the element stiffness matrices for low-order 2D elements. This changes however for 3D elements, for which [42] reported a 18x speedup for a NVIDIA GeForce 8800 GTX GPU as compared to both cores of a AMD X2 CPU at 2.6GHz.

The first global matrix assembly on the GPU in the context of FEM-SA was reported in [18], for a hyperelastic material. When assembling the global stiffness matrix for 1^{st} order tetrahedral elements, their best implementation achieves a speedup of around 15x for a NVIDIA GeForce 280 GTX GPU as compared to 1 core of an Intel Core2 Quad Q9550 CPU running at 2.83GHz, which would amount to around 2x acceleration if all four cores of the CPU were used. More recently, several assembly strategies have been analyzed and compared in [8]. There, for 2D elements of low order (1^{st} and 2^{nd}) and large mesh sizes, the best strategy provides a 65x speedup for a NVIDIA Tesla C1060 GPU versus one core of a Intel Core 2 Quad CPU Q9450 at 2.66GHz. As the assembly is done in single precision and only one core of the quad-core CPU is used, we can extrapolate a 8x speedup if double precision and all CPU cores were used. Efficient GPU implementation for FEM matrix assembly were also discussed in [41] in the context of accelerating the MiniFE FEM code [45], where a speedup of around 4x for a NVIDIA Tesla M2090 GPU was obtained against an Intel Xeon E5-2680 running at 2.7GHz and using all 8 cores. Recently,

Table 2 Overview of GPU accelerated FEM-SA solvers. The first and second columns show the solver stage/type. Relevant references for each solver stage/type are given in column 3 while acceleration reported for that stage is given in column 4. Finally, available relevant software, if any, for that stage, is given in column 5. "SP" denotes single precision computation, as opposed to the usual double precision.

Solver stage	Type	References	Acceleration	Software
Global stiffness matrix creation	Computation	[51,42]	1x vs 4 cores for 2D [51] 18x vs 2 cores for 3D [42]	MiniFE
	Assembly	[18,8]	15x vs 1 core [18] 65x vs 1 core (SP) [8] 4x vs 8 core [41]	MiniFE
Iterative solvers	CG	[5,33,23] [24,25,7] [9,21,10,59]	5x over 4 cores [21] 2-3x over 16 cores for 2 GPUs [10]	PETSc, ViennaCL CUSP, LAMA LAToolbox, CULA Sparse Acceleware
Preconditioners	Block Jacobi	[63,38]	20x over 1 core [63]	
	ILU/IC/SOR/SSOR	[38,46]	2-3x over 4 cores [46]	CUSPARSE, ViennaCL PETSc, LAToolbox Acceleware, CULA Spasrse
	Polynomial	[38,30]	n/a	CUSP, Acceleware
	Multigrid	[22,28,27] [47,31,2,62]	5-10x over 5 cores [22]	CUSP, ViennaCL PETSc, LAMA Acceleware
	SPAI	[17,55]	same as 16 cores [17]	PETSc, CUSP LAToolbox, Acceleware
Sparse direct solvers	Multi-frontal	[40,32,19,61]	2.9x vs 8 cores [32] 3x vs 1 core [61]	MatrixPro, BCSLIB-EXT
	Super-nodal	[56,34]	about same as 4 cores [34]	CHOLMOD, PASTIX

a new sparse matrix format and method for quickly performing matrix assemblies on the GPU has been reported in [64]. The method targets dynamic applications, where the matrix assembly has to be recomputed at every time step.

4.2 Matrix solvers - Iterative solvers

Being relatively easy to parallelize and port to accelerators like the GPU, iterative solvers were the first type of matrix solvers to be accelerated. There are many reports of acceleration for several types of iterative solvers, however here we concentrate on the Conjugate Gradient solver which is the usual choice for systems of linear equations which are symmetric and positive definite, as are the one arising from FEM-SA.

The first CG solver implementations using the GPU was reported in [5,33], at a time when technologies like CUDA and OpenCL where not yet available and general purpose algorithms had to take the form of shaders. The matrix structure was diagonal and only single precision could be used. Two years later, while targeting FEM applications that require double precision accuracy, which GPUs at that time did not support, [23] proposed the use of mixed-precision iterative refinement. While performing most of the work in single precision on the GPU and just a few double precision correction iterations on the CPU, they succeeded for the first time in implementing a GPU-accelerated CG solver providing full double precision accurate solutions. In later work [24], the same authors explored the emulated double precision and iterative refinement as alternative ways of obtaining double precision accuracy out of a CG solver ran on GPUs supporting single precision alone. They concluded that iterative refinement is the best of the two. Currently, most GPUs support double precision computation, however this is much slower than single precision, with a ratio ranging from 1/2 for the NVIDIA Tesla cards and 1/24 for NVIDIA GeForce series. Hence, the methods described in the previous papers are as relevant today as they were five years ago [25]. However, using mixed-precision in the context of the CG solver has to be done carefully so as not to increase the number of iterations too much.

While previous approaches used a structured data format, the first CG solver for unstructured meshes was reported in [7]. Good efficiency was obtained by using an optimized blocked format, suitable for matrices coming from fields such as FEM-SA. A solver running on multiple GPUs (all inside the same node) and working with unstructured matrices was then reported in [9] and [21], with the latter considering a data format suitable for overlapping communication and computation during a multi-GPU execution. A similar implementation, this time running on an entire GPU cluster with multiple GPUs per node was reported in [10]. Good performance and scalability were obtained by automatically choosing the most appropriate sparse matrix data structure and partitioning the data using a hyper-graph partitioning algorithm.

A good understanding of the speedup that can be obtained for accelerating an un-preconditioned CG solver can be obtained from the detailed performance reports provided in [10]. While testing their distributed CG solver on the TSUBAME supercomputer, for large matrices, they obtained an acceleration of up to 2-3x for two NVIDIA Tesla S1070 GPUs as compared to 8 AMD dual-core Opteron processors at 2.4GHz. In other words, the performance of one NVIDIA Tesla GPU was similar to that of 16-24 CPU cores. These results are consistent with the ones reported in [21], where the performance of one NVIDIA GeForce 280 GTX GPU was, on average, 5x faster than an Intel Core i7 975 running at 3.2 GHz and using all four cores, hence again the performance on one GPU being roughly equivalent to that of 20 CPU cores. Faster execution on both CPU and GPU can be obtained by using a blocked sparse matrix format. An in-depth analysis of blocking strategies for both single and multi-GPU execution is given in [59] where the authors also propose an accurate model for the performance a GPU-accelerated CG solver.

Currently there are several solver packages that provide a GPU accelerated CG solver (and several other iterative solvers like BiCGStab or GMRES). Open-source alternatives are PETSc, ViennaCL, CUSP, LAMA and LA Toolbox while commercial packages are being offered by Accelaware or CULA Sparse.

4.3 Matrix solvers - Preconditioners

Factorization based preconditioners like IC and SSOR are the usual choice for FEM-SA running on the CPU. Implementing such preconditioners on a GPU is very challenging because of the serial operations found in the triangular solves. One way to achieve a large amount of parallelism is to simply drop all elements from the factorization but the ones found in blocks along the main diagonal. GPU implementations for this type of preconditioning, also known as Block-Jacobi preconditioning, were reported in [63] and later in [38]. Performance-wise, the accelerated Block-Jacobi preconditioned GMRES solver from [63], running on an NVIDIA Tesla T10P GPU, achieved a 20x acceleration versus one core of an Intel Harpertown CPU clocked at 3GHz. However, this speedup is not representative of a real-world scenario where a more efficient, full IC preconditioner, would be running on the CPU. It is commonly known that the use of a Block-Jacobi preconditioner results in a very large number of solver iterations because of the large quantity of information that is dropped from the preconditioner.

Another way of extracting parallelism from factorization based preconditioners while discarding less information is by using a multi-coloring approach. The main idea here is the fact that there are no dependencies between elements of the same color, hence they can be processed in parallel. As long as there are many elements in each color, which is equivalent to having a small number of colors, there will be enough parallelism to exploit and the GPU performance will be good. The number of colors can be reduced by dropping elements, which again brings us to the trade-off between the quality of the preconditioner and the amount of available parallelism. The implementation of Multi-color SSOR, ILU and IC preconditioners on the GPU is reported in [38], where they choose a hybrid approach where the computation of the triangular solves is done on the GPU only if it is faster than on the CPU. Performance results reported in this work show that overall, the IC-preconditioned CG solver running on an NVIDIA Tesla C1060 GPU is only 2x-3x faster than one core of an Intel Xeon E5504 Processor CPU at 2GHz. The reason for this is that while SPMV and BLAS1 operations are indeed accelerated by a factor of 10x, the acceleration of the triangular solves is much smaller, at about 2x, which according to Amdahl's law results in a very lower value for the overall acceleration.

Recently, with NVIDIA adding support for sparse triangular solves in their CUSPARSE library (starting with CUDA 5.0), the implementations of ILU and IC type preconditioners has become much easier. The implementation of IC and ILU preconditioned CG and BiCGStab solvers, respectively, are described in details in [46] while a fully functional preconditioned CG solver can be downloaded from NVIDIA's website. The performance results are consistent with previous reports, with the preconditioned CG solver running on an NVIDIA Tesla C2050 being, on average, only 2x faster than an Intel Core i7 950 CPU at 3.07GHz and running Intel MKL on all cores. We again note the large difference in GPU performance between the un-preconditioned and preconditioned CG solvers.

The quality of a preconditioner can sometimes be improved by applying it not directly but as a polynomial. Such a GPU-accelerated preconditioner was reported in [63] where their Block-Jacobi type preconditioner running on the GPU was used as the underlying preconditioner. Another accelerated polynomial preconditioner, this type of Chebyshev type, was reported in [30].

Multigrid type preconditioners make another class of powerful preconditioners, used mostly in fields like CFD rather than FEM-SA. Geometric multigrid (GMG) is easier to implement and more efficient to execute, however its applicability to FEM-SA is limited to the cases when the FEM mesh has been obtained via refinement from a coarser starting mesh. Such a GPU accelerated GMG, based almost entirely on the SPMV kernel, was reported in [22]. As expected, the resulting speedup is consistent with the speedup of the SPMV operation when executed on the GPU. Using a NVIDIA GeForce GTX 285 GPU, for large problem sizes, they obtain 5-10x acceleration as compared to an Intel Core i7 920 CPU at 2.66GHz and using all four cores. A similar approach to GMG, but this time using stronger smoothers like multi-colored Gauss-Seidel, multi-colored ILU and and Factorized Sparse Approximative Inverse is reported in [28]. These smoothers can also be directly applied as preconditioners to a CG solver. As opposed to GMG, algebraic multigrid (AMG) is not restricted to cases where refinement operations are applied. While no GPU-accelerated AMG applied to FEM-SA problems has been reported so far, there are reports on such AMG implementations in fields like CFD or electromagnetics [27,47,31]. For a comparison on how these fare against modern multi-core CPUs we refer to [62]. Regardless of the type of multigrid scheme that is being used, the time spent in the smoother represents a significant portion from the total solution time. For highly parallel architectures like the GPU the smoother can be accelerated by reducing the number of synchronization points, which occur at every smoother iteration in the case of Jacobi or Gauss-Seidel type smoothers. This is possible using an asynchronous method as the one proposed in [2].

Yet another class of preconditioners are Sparse Approximative Inverse (SPAI) type. SPAI type preconditioners approximate the inverse of a matrix and are computed by solving a least-square problem for each row in the matrix. As such, their computation exposes a large degree of parallelism that can be exploited by GPUs, as reported in [17,55]. In [17] the performance of the GPU-accelerated SPAI preconditioner running on a NVIDIA Tesla M2070 GPU was proved equivalent with the one of ParaSails, one of the fastest SPAI packages, running on 16 AMD Opteron 252 CPUs at 2.6GHz.

GPU-accelerated preconditioners are now available in a variety of solver packages. Factorization type preconditioners are offered by open source packages like CUSPARSE, ViennaCL, PETSc (via CUSPARSE) and LAtoolbox and commercial ones like Acceleware and CULA Sparse. Polynomial preconditioners are offered by CUSP and Acceleware while AMG is provided by CUSP, ViennaCL, PETSc, LAMA and Acceleware.

4.4 Matrix solvers - Sparse Direct solvers

Sparse direct solvers approach the matrix directly, via factorization, while taking into account the sparsity of the matrix. Most sparse direct solvers fall into two classes: multi-frontal and super-nodal. Both types reorder and group matrix entries into dense blocks, which can then be factorized using dense factorization methods. Such methods, due to good data locality, are known to perform very efficiently on many-core architectures like the GPU, with performance close to machine peak. Since solving the matrix is done by algebraic manipulation rather then using an iterative algorithm, sparse direct solvers do not need preconditioners. Hence, the difficulty in porting them to GPUs lays in efficiently scheduling and performing the sparse factorizations on the GPU. The main challenge here is that only blocks with the size in a certain range can be factorized on the GPU. For very small blocks, the overhead of copying the data to the GPU and back cancels any speedup that might be obtained there while too large blocks might not fit in the memory of the GPU.

The first multi-frontal sparse direct solver on GPU was reported in [40] where it was concluded that porting only the matrix multiplication (the most time-consuming kernel) to the GPU, is not enough to get a speedup over a CPU implementation. Good management of the data movement is required for an efficient implementation. Such improved solvers were later reported in [32,19], with [32] reporting good speedup for a series of structural analysis problems, that is 4x with mixed precision and 2.9x for double precision while using a NVIDIA Tesla C1060 GPU as compared to two Intel Xeon 5335 CPUs at 2Ghz (8 cores in total). Unfortunately, as proved by [61], the performance of the solver is highly dependent on the matrix to be solved, that is of the distribution of the sizes of the fronts to be factorized and the dependencies between these fronts. In their benchmarks, they obtain a speedup of only 3x for an NVIDIA Tesla C1060 GPU as compared to a single core of a quad-core Intel Nehalem CPU.

The first report on a GPU-accelerated super-nodal type sparse direct solver is [56]. In their work, the authors enhance the parallel direct solver Pardiso by replacing the most time consuming routines in the solver, that is, matrix multiplication, LU decomposition and triangular solves, with GPU counterparts based on the NVIDIA CUBLAS library. Moreover, this switch is done only if the number of requiring floating operation exceeds a certain threshold. The obtained speedup is however rather limited, with one NVIDIA GeForce 8800 GTX obtaining a 2-3x acceleration as compared to an Intel Pentium 4 CPU at 3.4GHz. Very recently, a similar result was reported in [34], this time using a sophisticated Directed Acyclic Graph (DAG) based scheduler and modern GPU hardware. These results show that the acceleration brought by adding GPUs is quickly diminished when increasing the number of cores, with the added performance being roughly equivalent to that of 4 CPU cores.

From a software point of view, there are currently two package (both commercial), which provide GPU-accelerated multi-frontal type direct solvers: MatrixPro and BCSLIB-EXT. Two other packages (open source), provide GPU-accelerated super nodal type solvers: CHOLMOD and PASTIX.

5 GPU accelerated FEM-SA software

In this section we summarize the software in the field of FEM-SA that is currently available and that either employs GPU acceleration or can be used to add GPU acceleration support to applications.

Currently, most of the major ISV packages for FEM-SA provide the user with GPU-acceleration capabilities. All these packages accelerate only the linear equation solver and perform the rest of the tasks using the CPU. Moreover, most packages use a hybrid approach, where only tasks which are large enough to amortize the cost of data transfer are offloaded to the GPU. A list of such software, together with the type of solvers they employ, is given in Table 3. For each package, speedup results that where reported, mostly by the developers themselves in conference presentations, are provided as well. It is easy to observe that only the speedup for the CG solver inside RADIOSS gets close to the one order of magnitude that we are used to seeing in the context of GPUs. In the case of direct solvers, the actual speedup is much lower, with usually less than 2x being obtained against a multi-socket CPU node.

We provide a list of available GPU-accelerated matrix solver packages, both free and commercial, in Table 4. These can be used to either write a new, accelerated FEM-SA solver package or, most likely, to accelerate an existing one. Note that most of the packages matrix solvers offer only CUDA support, the only exceptions being ViennaCL, LAMA and Acceleware, which support both CUDA and OpenCL.

6 Conclusions

GPU-accelerated computing has come a long way since five years ago when CUDA was first introduced. Both hardware and software ecosystems are now mature, with NVIDIA and CUDA having a large head start as compared to AMD and OpenCL. In particular, NVIDIA GPUs are by far the preferred choice for linear equation solvers and FEM-SA in general, with almost no performance results having been reported for AMD GPUs and only 3 of the 11 GPU accelerated matrix solver packages that we have found so far providing support for OpenCL.

We have found some reports on work done in accelerating the pre-processing stage of FEM-SA, like 2D mesh generation and shape simplification. The acceleration of 3D mesh generation, one of the most time consuming steps in FEM-SA, seems to be still far away, with no such results being reported so far. GPU acceleration is currently limited to 2D mesh generation and shape simplification as the mesh level. However, no software providing such features is available so far.

Significant speedups have been achieved while accelerating the creation and assembly of the global stiffness matrix. However, to the best of our knowledge, no ISV FEM-SA solver currently employs such technology, as for most analyses the overall time spent in this stage is much shorter than the time spent in the matrix solver. As expected, most of the work so far has been concentrated on the acceleration of the linear equation solvers, both iterative and sparse direct.

For iterative solvers, very significant speedups can be achieved by offloading SPMV and vector operations to the GPU. Indeed, for an un-preconditioned or diagonally-preconditioned CG solver, one GPU can provide performance equivalent with that of 20 CPU cores. This speedup drops almost one order of magnitude when more powerful preconditioners, like IC/SSOR are applied, hence powerful preconditioners suitable for the GPU are still an open area of research. In many cases, a "brute force" use of the GPU, with simple preconditioners like Jacobi or Block-Jacobi may provide the best performance. On the software side, there are currently several solver packages, both open source and commercial, that provide GPU accelerated Krylov solvers and preconditioners.

GPU accelerated direct solvers of multi-frontal type have been adopted by the majority of FEM-SA ISV vendors. Limited but significant speedups have been achieved in this way, with the addition of one or two GPUs to multi-core node reducing the overall time to solution to about half. Software packages that provide sparse direct solvers of both multi-frontal type (cur-

Table 3 List of GPU accelerated FEM-SA ISV packages. The first two columns provide the name of the solver and the type of linear equation solver that is employed. The third column provides reported speedups.

Package name	Solver	Reported speedup
ABAQUS	Multifrontal	2x for 1 GPU vs 4 cores [13]
		1.4-1.5x for 1 GPU/node vs 12 cores [13]
		1.7-2.4x for 2 GPU/node vs 12 cores [13]
RADIOSS	Multifrontal	1.8 - 2.5x for 1 GPU vs 4 cores [37]
	CG	6.5-7.5x for 2 GPU/node vs 12 cores [37]
		8.8-13x for 4 GPU/node vs 12 cores [37]
MSC Nastran	Multifrontal	1.55-2x for 2 GPUs vs 8 cores [39]
ANSYS Mechanical	Multifrontal	1.7-2.1x for 1 GPU vs 8 cores [52]
		1.4-1.7x for 1 GPU vs 16 core [52]
LS-DYNA	Multifrontal	1.06-3.3x for 1 GPU vs 8 cores [26]

Table 4 List of GPU-accelerated matrix solver packages. The first two columns show the name of package and the license it is distributed under. "Open Source" stands for a non-standard open source License. The following two columns show the types of solvers provided, either iterative (CG) or sparse direct (S.D.). The next six column show the availability of preconditioners, with BJ standing for Block-Jacobi. For each solver or preconditioner, the presence of a feature is denoted by a ○ mark while its absence is shown as an empty space

Package	License	CG	S.D.	ILU	IC	BJ	SPAI	POLY	AMG
Acceleware[1]	Commercial	○		○			○	○	○
MatrixPro[43]	Commercial		○						
BCSLIB-EXT[3]	Commercial		○						
CULA Sparse[14]	Free academic use	○		○		○			
PETSc[50]	Open Source	○		○			○		○
CUSP[15]	Apache 2.0	○					○	○	○
CUSPARSE	Free			○	○				
ViennaCL[60]	MIT	○		○	○				○
PASTIX[49]	CeCILL-C		○	○					
CHOLMOD[12]	GPL/LGPL		○						
LAMA[35]	MIT	○							○
LAToolbox[36]	Open Source	○		○			○		

rently all commercial) and super-nodal type (all open-source) are also available.

References

1. Acceleware: http://www.acceleware.com/matrix-solvers
2. Anzt, H., Tomov, S., Gates, M., Dongarra, J., Heuveline, V.: Block-asynchronous multigrid smoothers for GPU-accelerated systems. Procedia Computer Science **9**, 7 – 16 (2012)
3. BCSLIB-EXT: http://www.aanalytics.com/products.htm
4. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 18:1–18:11. ACM (2009)
5. Bolz, J., Farmer, I., Grinspun, E., Schrooder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: ACM SIGGRAPH 2003 Papers, pp. 917–924. ACM (2003)
6. Botsch, M., Bommes, D., Vogel, C., Kobbelt, L.: GPU-based tolerance volumes for mesh processing. In: Proceedings of the 12th Pacific Conference on Computer Graphics and Applications, pp. 237–243 (2004)
7. Buatois, L., Caumon, G., Lévy, B.: Concurrent number cruncher: An efficient sparse linear solver on the GPU. High Performance Computing and Communications pp. 358–371 (2007)
8. Cecka, C., Lew, A., Darve, E.: Assembly of finite element methods on graphics processors. International Journal for Numerical Methods in Engineering **85**(5), 640–669 (2011)
9. Cevahir, A., Nukada, A., Matsuoka, S.: Fast conjugate gradients with multiple GPUs. Computational Science–ICCS 2009 pp. 893–903 (2009)
10. Cevahir, A., Nukada, A., Matsuoka, S.: High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. Computer Science-Research and Development **25**(1), 83–91 (2010)
11. Choi, J., Singh, A., Vuduc, R.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing, pp. 115–126. ACM (2010)
12. CHOLMOD: http://www.cise.ufl.edu/research/sparse/cholmod/
13. Crivelli, L., Dunbar, M.: Evolving use of GPU for Dassault Systemes simulation products. In: GPU Technology Conference (GTC 2012) (2012)
14. CULA Sparse: http://www.culatools.com/sparse/
15. CUSP: http://code.google.com/p/cusp-library/
16. DeCoro, C., Tatarchuk, N.: Real-time mesh simplification using the GPU. In: Proceedings of the 2007 symposium on Interactive 3D graphics and games, pp. 161–166 (2007)
17. Dehnavi, M.M., Fernandez, D., Gaudiot, J.L., Giannacopoulos, D.: Parallel sparse approximate inverse precon-

ditioning on graphic processing units. IEEE Transactions on Parallel and Distributed Systems **99**, 1 (2012)
18. Filipovic, J., Peterlik, I., Fousek, J.: GPU acceleration of equations assembly in finite elements method - preliminary results. In: Symposium on Application Accelerators in HPC (SAAHPC) (2009)
19. George, T., Saxena, V., Gupta, A., Singh, A., Choudhury, A.: Multifrontal factorization of sparse SPD matrices on GPUs. In: IEEE International Parallel & Distributed Processing Symposium (IPDPS), pp. 372–383 (2011)
20. Georgescu, S., Chow, P.: GPU accelerated CAE using open solvers and the cloud. SIGARCH Comput. Archit. News **39**(4), 14–19 (2011)
21. Georgescu, S., Okuda, H.: Conjugate gradients on multiple GPUs. International Journal for Numerical Methods in Fluids **64**, 1254–1273 (2010)
22. Geveler, M., Ribbrock, D., Gdeke, D., Zajac, P., Turek, S.: Efficient finite element geometric multigrid solvers for unstructured grids on GPUs. In: Proceedings of the The Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering (PARENG 2011) (2011)
23. Göddeke, D., Strzodka, R., Turek, S.: Accelerating double precision FEM simulations with GPUs. In: F. Hülsemann, M. Kowarschik, U.a. Rüde (eds.) 18th Symposium Simulationstechnique, Frontiers in Simulation, pp. 139–144. SCS Publishing House e.V. (2005). ASIM 2005
24. Göddeke, D., Strzodka, R., Turek, S.a.: Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations. International Journal of Parallel, Emergent and Distributed Systems **22**(4), 221–256 (2007)
25. Göddeke, D., Strzodka, R.a.: Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations (part 2: Double precision gpus). Tech. rep., Fakultät für Mathematik, TU Dortmund (2008). Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 370
26. Göhner, U.: Usage of GPU in LS-DYNA. LS-DYNA Forum (2012)
27. Haase, G., Liebmann, M., Douglas, C., Plank, G.: A parallel algebraic multigrid solver on graphics processing units. High Performance Computing and Applications pp. 38–47 (2010)
28. Heuveline, V., Lukarski, D., Trost, N., Weiss, J.P.: Parallel smoothers for matrix-based geometric multigrid methods on locally refined meshes using multicore CPUs and GPUs. In: R. Keller, D. Kramer, J.P. Weiss (eds.) Facing the Multicore-Challenge II, pp. 158–171. Springer-Verlag (2012)
29. Hjelmervik, J., Léon, J.: GPU-accelerated shape simplification for mechanical-based applications. In: Shape Modeling and Applications, 2007. SMI'07. IEEE International Conference on, pp. 91–102. IEEE (2007)
30. Kamiabad, A.: Implementing a preconditioned iterative linear solver using massively parallel graphics processing units. Master's thesis, University of Toronto (2011)
31. Kraus, J., Fster, M.: Efficient AMG on heterogeneous systems. In: R. Keller, D. Kramer, J.P. Weiss (eds.) Facing the Multicore - Challenge II, *Lecture Notes in Computer Science*, vol. 7174, pp. 133–146. Springer Berlin Heidelberg (2012)
32. Krawezik, G., Poole, G.: Accelerating the ANSYS direct sparse solver with GPUs. In: 2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09) (2009)
33. Krüger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans. Graph. **22**, 908–916 (2003)
34. Lacoste, X., Ramet, P., Faverge, M., Ichitaro, Y., Dongarra, J., et al.: Sparse direct solvers with accelerators over DAG runtimes. Tech. Rep. 7972, INRIA (2012)
35. LAMA: http://www.libama.org
36. LAToolbox from HiFlow: http://www.hiflow3.org
37. Lequiniou, E., Zhou, H.: Speedup Altair RADIOSS solvers using NVIDIA GPU. In: GPU Technology Conference (GTC 2012) (2012)
38. Li, R., Saad, Y.: GPU-accelerated preconditioned iterative linear solvers. Tech. rep., University of Minnesota (2010)
39. Liao, C.: MSC Nastran sparse direct solvers for Tesla GPUs. In: GPU Technology Conference (GTC 2012) (2012)
40. Lucas, R., Wagenbreth, G., Tran, J., Davis, D.: Multifrontal computations on GPUs. Tech. rep., Unpublished ISI White Paper (2007)
41. Luitjens, J., Williams, A., Heroux, M.: Optimizing miniFE an implicit finite element application on GPUs. In: GPU Technology Conference (GTC 2012) (2012)
42. Maciol, P., Plaszewski, P., Banas, K.: 3D finite element numerical integration on GPUs. Procedia Computer Science **1**(1), 1087–1094 (2010)
43. MatrixPro-GSS: http://www.matrixprosoftware.com/
44. Minden, V., Smith, B., Knepley, M.: Preliminary Implementation of PETSc Using GPUs. In: Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering (2010)
45. MiniFE: https://software.sandia.gov/mantevo/download.html
46. Naumov, M.: Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS. Tech. rep., Technical Report and White Paper (2011)
47. Neic, A., Liebmann, M., Haase, G.: Algebraic multigrid solver on clusters of CPUs and GPUs. Applied Parallel and Scientific Computing pp. 389–398 (2012)
48. NVIDIA: NVIDIA CUDA programming guide 5.0 (2012)
49. PASTIX: http://pastix.gforge.inria.fr
50. PETSc: http://www.mcs.anl.gov/petsc/
51. Płaszewski, P., Macioł, P., Banaś, K.: Finite element numerical integration on GPUs. Parallel Processing and Applied Mathematics pp. 411–420 (2010)
52. Posey, S., Courteille, F.: GPU progress in sparse matrix solvers for applications in computational mechanics. In: European Seminar on Computing (ESCO12) (2012)
53. Qi, M., Cao, T.T., Tan, T.S.: Computing 2D constrained Delaunay triangulation using the GPU. In: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '12, pp. 39–46. ACM, New York, NY, USA (2012)
54. Rong, G., Tan, T., Cao, T., et al.: Computing two-dimensional Delaunay triangulation using graphics hardware. In: Proceedings of the 2008 symposium on Interactive 3D graphics and games, pp. 89–97. ACM (2008)
55. Sawyer, W., Vanini, C., Fourestey, G., Popescu, R.: SPAI preconditioners for HPC applications. PAMM **12**(1), 651–652 (2012)
56. Schenk, O., Christen, M., Burkhart, H.: Algorithmic performance studies on graphics processing units. Journal of Parallel and Distributed Computing **68**(10), 1360–1369 (2008)
57. Shontz, S.M., Nistor, D.M.: Cpu-gpu algorithms for triangular surface mesh simplification. In: X. Jiao, J.C.

Weill (eds.) Proceedings of the 21st International Meshing Roundtable, pp. 475–492. Springer Berlin Heidelberg (2013)
58. The Khronos Group: OpenCL specification 1.2 (2011)
59. Verschoor, M., Jalba, A.C.: Analysis and performance estimation of the conjugate gradient method on multiple GPUs. Parallel Computing **38**, 552 – 575 (2012)
60. ViennaCL: http://viennacl.sourceforge.net/
61. Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M., Shringarpure, A.: On the limits of GPU acceleration. In: Proceedings of the 2nd USENIX conference on Hot topics in parallelism, p. 13 (2010)
62. Wagner, M., Rupp, K., Weinbub, J.: A comparison of algebraic multigrid preconditioners using graphics processing units and multi-core central processing units. In: Proceedings of the 2012 Symposium on High Performance Computing, HPC '12, pp. 2:1–2:8 (2012)
63. Wang, M., Klie, H., Parashar, M., Sudan, H.: Solving sparse linear systems on NVIDIA Tesla GPUs. Computational Science–ICCS 2009 pp. 864–873 (2009)
64. Weber, D., Bender, J., Schnoes, M., Stork, A., Fellner, D.: Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. Computer Graphics Forum **32**(1), 16–26 (2013)