Paper 6

©Civil-Comp Press, 2012
Proceedings of the Eighth International Conference
on Engineering Computational Technology,
B.H.V. Topping, (Editor),
Civil-Comp Press, Stirlingshire, Scotland

Application of GPU-Based Computing to Large Scale Finite Element Analysis of Three-Dimensional Structures

A. Akbariyeh[1], T.J. Carrigan[2], B.H. Dennis[1]
W.S. Chan[1], B.P. Wang[1] and K.L. Lawrence[1]
[1] University of Texas at Arlington, United States of America
[2] Pointwise Inc., Fort Worth, Texas, United States of America

Abstract

Finite element analysis for stress and deformation prediction has become routine in many industries. However, the analysis of complex three-dimensional geometries composed of millions of degrees of freedom is beyond the computing capacity of the typical desktop computer. Recent advances in commodity graphics processing units (GPUs) have opened a new route for computing large scale finite element solutions on the desktop computer in a timely fashion. The current generation of advanced GPU hardware is equipped with hundreds of streaming processors and offers the potential of teraflops performance. It is no surprise that this hardware is becoming increasingly popular for general scientific computing because of their high performance to cost ratio. However, the peak performance of the GPU is difficult to obtain and requires careful consideration regarding algorithms and implementation. In the case of finite element analysis, the most time consuming step is solving the resulting linear system of equations. This system is typically large, sparse, and unstructured. Often these systems are solved iteratively with a domain decomposition technique used to distribute the computational load among parallel processors. In this paper the benefits of using GPUs to improve the performance of the iterative sparse matrix solver for a finite element program is explored. We focus specifically on hexahedral elements for linear elasticity with trilinear basis functions for displacement. This does not require domain decomposition, so it is simpler than the corresponding implementation for distributed memory parallel computers. In addition, the performance of the GPU implementation is compared with the corresponding serial version run on a conventional processor for various mesh sizes and sparse matrix storage schemes.

Keywords: finite element, graphics processor, parallel processing, sparse matrix.

1 Introduction

The past decade has witnessed a rapid growth in multi-core shared memory architectures for consumer level computing. The trend appears to move away from increase clock speed and towards increasing the number of processing units on a single processing chip. The video graphics hardware manufacturers have been the leaders in this approach, offering commodity video cards that may contain hundreds of processing units. They have successfully produced specialized hardware that accelerates typical graphics processing operations such as shading, texturing, and coordinate transformations. These operations can often be done in parallel and hence increasing the processing units increases the rendering speed. Often this specialized hardware, known as a graphics processing unit (GPU) can perform these operations thousands of times faster than the best general purpose computational processing unit (CPU). Theoretical peak speed for some of these GPUs exceed 1 teraflop/s, which has given them very impressive price/performance ratios.

Recently, there has been growing interest in applying GPUs to general purpose scientific computing. Of course, the best performance will be obtained for algorithms that can be mapped to basic operations used in computer graphics. However, an algorithm that is parallel in nature can potentially experience significant speedups on a GPU compared to a CPU, if carefully implemented. Some researchers have implemented finite difference and even computational fluid dynamics solvers on GPUs with some success [1, 2, 3].

In this paper, we report on a GPU implementation for a 3-D linear elastic finite element solver for 8-node elements. We compare the speed for the code on a single CPU with the speed on a GPU. In our approach, a fully assembled linear system of equations is stored in a sparse matrix format and solved with an iterative solver with a simple diagonal preconditioner. The iterative solver is parallel in nature as it depends on matrix vector multiplications and is a good candidate for acceleration with the GPU. One important aspect of this work is the recognition that the stiffness matrix is a sparse collection of smaller dense block matrices. We exploit this pattern to reduce the number of calls to global memory on the GPU and thus maximize our utilization of the hardware. Sparse matrix block format requires special matrix-vector operations that were implemented in NVIDIA CUDA C [4] and represent an extension of the codes already available from NVIDIA for sparse matrix operations. We show examples of method applied to 3-D linear elastic analysis for various mesh sizes.

2 GPU hardware

Understanding the parallel architecture of the GPU hardware is essential for writing efficient and compatible software code. In this section we briefly introduce the current NVIDIA GPU architecture compute capability 2.x to those who are not exposed to the matter.

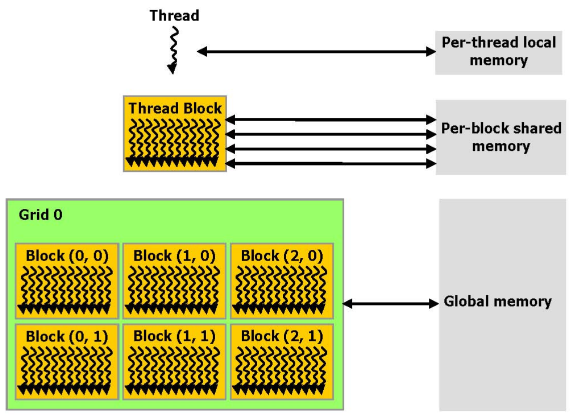

Figure 1: Illustrating the thread-memory interaction on GPU [4]

CUDA C is the primary language designed for doing scientific computing on NVIDIA GPUs. CUDA is essentially a set of language extensions for C. The parallel part of the program is often referred to as the kernel. The kernel is a C function that, when called, is executed N times in parallel by N different CUDA threads in a hierarchical fashion [4]. Each thread has a unique thread ID and they are organized in groups called thread block and blocks are launched in a grid of blocks. A GPU program still runs on the CPU and during execution, data is transferred between GPU global memory and system memory. The CPU side is called the host, and the GPU side is called the device. The GPU hardware consists of an array of Streaming Multiprocessors (SMs). The scheduling of a grid-block on these SMs is done internally based on available execution capacity. The scheduler distributes thread blocks among available SMs. The CUDA uses SIMT(Single-Instruction, Multiple-Thread) architecture to leverage instruction-level parallelism. A multiprocessor breaks the given block into fixed-size groups of threads called warps[1], and it executes the stream of warps in a SIMT fashion.

Similar to kernel execution, GPU memory space is also hierarchical. Global memory, shared memory and registers are the main device memory types. Each thread has private registers. Each thread block has a shared memory visible to all members. All threads have access to the same global memory [4]. Figure 1 shows the accessibility of different memory spaces during execution of a single grid of blocks on one GPU. Global memory has the highest memory access latency and the registers have the lowest.

3 Compressed sparse row block matrix storage format

Choosing the right storage format for sparse matrices has the most impact factor on performance of matrix vector operations. We altered the well known compressed

[1]32 for compute capability 2.0

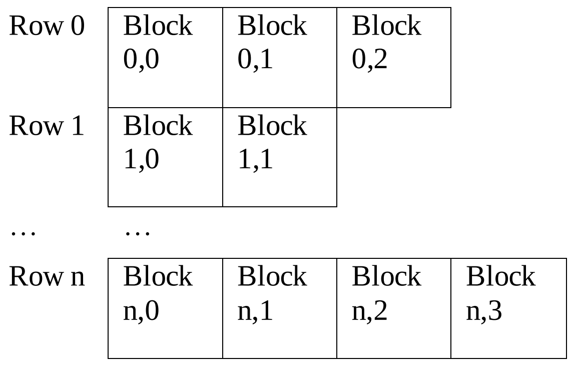

Figure 2: Compress Sparse Row Block storage scheme

sparse row storage format to achieve higher performance for multiple degrees of freedom (DOF) per node problems. In the regular Compressed Sparse Row(CSR), the sparse matrix is represented by three vectors; $values$, $rows$ and $columns$ [5]. In a single DOF problem when we generate local stiffness matrix, coupling between nodes is represented by one number. However in multi-DOF problems, this coupling happens to be a square matrix, or block. The idea behind compressed sparse row block is to store data in a way that reduces memory accesses in a matrix vector multiplication operation. When stored blockwise, indexing occurs block by block so data placement or retrieval is increased according to the blocksize.

The CSRB storage format is very similar to Compressed Sparse Row storage. It consists of three vectors. The first vector $vals$ contains all non-zero values in the sparse matrix. The next vector $cols$ stores the column indices of non-zero blocks of each row. The last vector $rows$ stores pointers to the beginning of each row of blocks. Fig. 2 represents the schematic of CSRB matrix storage format.

4 Matrix-vector multiplication

Matrix-vector multiplication ($Ax = y$) is the most computationally intensive operation used by iterative methods when solving sparse linear systems with simple preconditioners. For example, the Jacobi preconditioned conjugate gradient (PCG) method has one MVM operation per iteration and it is the most time consuming part of the solution. It is a logical choice to focus on improving the performance of this operation. There are several ways to exploit this naturally parallel operation. One option is to assign a parallel thread to each entry in the matrix. However, this approach requires additional techniques to avoid a race condition when the products are summed. To avoid the race condition, we assign the multiplication of each row to one thread. Note that this approach does not require the addition step of mesh partitioning or domain decomposition used in other parallel algorithms.

In the case of a 3-DOF per node problem, CSR MVM has to read the same data from vector x for each DOF of each node, in three consecutive rows. In other hand our Compressed Sparse Row Block(CSRB) multiplication kernel handles all DOF of each node at once and therefore it avoids redundant memory accesses to vector x.

Parallel MVM kernels are bandwidth limited. Hence any reduction in memory access results in higher performance. The benefit of CSRB is that it only stores row and column pointers of blocks which save memory storage but the key point is the better performance in the matrix vector multiplication due to elimination of redundant memory accesses to vector x and less memory reads to resolve row and column indices.

We developed a MVM parallel GPU kernel as well as a serial CPU version specially to handle CSRB MVM operations. The GPU MVM kernel assigns one node to one parallel thread to do the multiplication and returns the results for all DOF associated with that node. As mentioned before, CSRB eliminates redundant memory access to the vector x and resolves row and column indices with less memory reads. Another way to interpret this is that the CSRB GPU kernel has higher instruction intensity compared to a CSR kernel. The ratio between instructions and memory read in CSRB is higher than the regular CSR.

The parallel GPU MVM kernel tends to diverge in execution due to variable number of non-zeros per row. This means in each thread block, all threads have to wait for the longest row to finish multiplying. One way to get around the thread block divergence is to schedule the nodes with the same row length together. But scheduling nodes in a scattered pattern would have a penalty in accessing vector x and is explained in the next paragraph.

When developing GPU kernels for bandwidth limited problems, global coalesced memory access is the most important factor to consider. Coalesced memory access simply means there is no gap or stride between requested data. Typically in parallel codes, multiple threads make requests to access different memory locations at the same time. If these requests happen to point to a series of memory locations, memory access will be so called coalesced. In our CUDA code, we have to access global memory three times during execution. The first access is reading stiffness matrix data. We addressed this by reordering and padding our CSRB data structure to assure coalesced global memory access. The second global memory access is reading vector x. Each parallel thread has to deal with multiple blocks and has to read 3 values from vector x per block. In general there is no guarantee to have coalesced reads from vector x for each row. But normally the mesh generators use re-numbering techniques to reduce stiffness matrix bandwidth. In CSRB case when we execute multiple threads which are accessing memory locations that are close to each other, the Fermi GPU will automatically line them up and manage the memory access the best possible way. The conclusion is memory access to vector x could be considered partially coalesced as long as we keep the neighboring nodes together or in other words keep the bandwidth as low as possible. And finally the third global memory access is writing results back to vector y. Because each block multiplies consecutive rows at once, writing to vector y in global memory is fully coalesced.

5 Preconditioned conjugate gradient method

The Conjugate gradient algorithm is one of the best known iterative solvers for sparse symmetric positive definite linear systems. We have implemented preconditioned conjugate gradient (PCG) as our iterative solver. In our implementation of PCG method, we have eliminated one inner product per iteration simply by reordering steps in the algorithm. We use Jacobi preconditioning for simplicity [5]. An additional inner product operation is used to calculate the norm of the residual to gauge solution convergence.

Algorithm 1:Original PCG algorithm [5]

```
Compute $r_0 = b - Ax_0,\ z_0 = M^{-1}r_0$, and $p_0 = z_0$
For $j = 0, 1, ...$, until convergenc Do
    $\alpha_j = (r_j, z_j)/(Ap_j, p_j)$
    $x_{j+1} = x_j + \alpha_j p_j$
    $r_{j+1} = r_j - \alpha_j p_j$
    $z_{j+1} = M^{-1}r_{j+1}$
    $\beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$
    $p_{j+1} = z_{j+1} + \beta_j p_j$
    $\alpha_i = (r_i, r_o)/(Ap_i, r_o)$
EndDo
```

Algorithm 2:Modified Implementation of PCG Algorithm for GPU

```
$r = Ax$
$r = b - r$
$z = M^{-1}r$
$p = z$
For $j = 0, 1, ...$ until convergence Do
    If $j = 0$ Then
        $temp_1 = (r, z)$
    Else
        $temp_1 = (r, z)$
        $\beta = temp_1/temp_0$
        $p = z + \beta p$
    EndIf
    $temp_2 = A\,p$
    $\alpha_j = temp_1/temp_2$
    $x = x + \alpha\,p$
    $r = r\ - \alpha\,temp_2$
    $z = M^{-1}r$
EndDo
```

For the GPU implementation of the PCG iterative solver, memory transfers between host and device should be minimized. Simply electing to utilize the GPU only for MVM operations, results in excessive number of memory transfers and hence poor solver performance. But if we also do the inner product operations on GPU, there is no need to transfer vectors back and forth between host and device. Other researchers found that for conjugate gradient solver, increased performance could be realized by utilizing GPU to compute inner products as well as matrix-vector products [6]. However, inner products are not entirely parallel as the products of the vector elements must be summed. Some parallelism can be introduced in the summation by using a reduction operation, however parallel efficiency is low unless the vector is very long. So, minimizing the number of inner products will help to improve overall parallel efficiency for our GPU kernel. For the inner products that could not be elimated, we use standard NVIDIA cuBLAS library to carry out our inner products as well as level 1 BLAS operations [7].

6 Results

In this section we apply the GPU to the solution of systems of equations resulting from finite element discretizations. The cases considered here are composed of 8-node hexahedral elements for linear elastic problems. The local element stiffness matrices are formed in the conventional way [8]. The local stiffness matrices are assembled into a global matrix with CSRB format and then boundary conditions are enforced. The aforementioned steps are completed on the CPU of the host computer while the solution of the system of equations is performed on the GPU using the PCG method described in previous sections. Before executing the PCG function, the global stiffness matrix and force vector are copied to the global memory on the GPU. Timing functions are used to measure time required to perform a single matrix vector product and the time to perform a single PCG iteration. Similar timing functions were used in the serial version of the code running on the host CPU.

We then define the speedup, which is a measure of performance of the GPU relative to the CPU, with the formula below.

$$speedup = \frac{cpu\_execution\_time}{gpu\_execution\_time} \tag{1}$$

For the serial timing, double precision computations using a C code were run using a single core on a quad-core Intel i5 2.66GHz processor with 4GB of system memory. GPU computations were done in double precision utilizing a CUDA C code. An NVIDIA Quadro 5000 GPU with 352 cores and 2.5GB of memory was used as the GPU.

The speedup is used to assess the GPU performance for two different geometry cases. In the first case, we consider a cantilever beam that is fixed at one end and has a uniform shear load applied at the other. This problem has a known exact solution and can be meshed with different resolutions quite easily. An example mesh is shown in Figure 3.

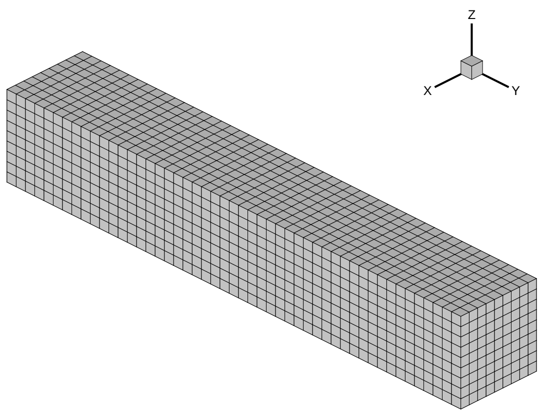

Figure 3: Mesh for cantilever beam

Grid	Total DOF	Nodes	Elements	Non-Zeros
1	3000	1000	624	19942
2	30000	10000	9019	233632
3	500,400	166,800	152,627	4,246,528
4	1,081,344	360,448	332,661	9,228,544
5	1,520,640	506,880	485,100	13,292,020
6	2,010,624	670,208	642,033	17,586,400

Table 1: Meshes used for cantilever beam case

The speedup is compared for the six different mesh sizes given in Table 1. The sizes range from 3000 to more than 2 million DOF.

The speedup for the PCG solver for different grid sizes is shown in Figure 4. Note that in this case, both the GPU and CPU code use full matrix CSRB, although the global stiffness matrix is symmetric. Figure 5 compares the GPU speedup for the case where more conventional symmetric CSRB storage is used on the CPU side.

In the development of CUDA kernel for MVM operation, we found using full-matrix storage gave better memory access and performance compared to symmetric matrix storage. Although it is not necessary to do the same on the CPU side, we included Fig. 4 to show the actual speedup using the same algorithm. Using symmetric CSRB storage format on the CPU, results in faster execution of the PCG algorithm comparing to full-matrix CSRB storage, therefor we added Fig. 5 to show the minimum performance gain of the GPU PCG implementation.

The PCG results demonstrate the ability of the GPU CUDA code to significantly outperform the serially executed C code in both cases. For grid sizes above 1 million DOF, the CUDA implementation of the PCG algorithm shows the maximum and fairly

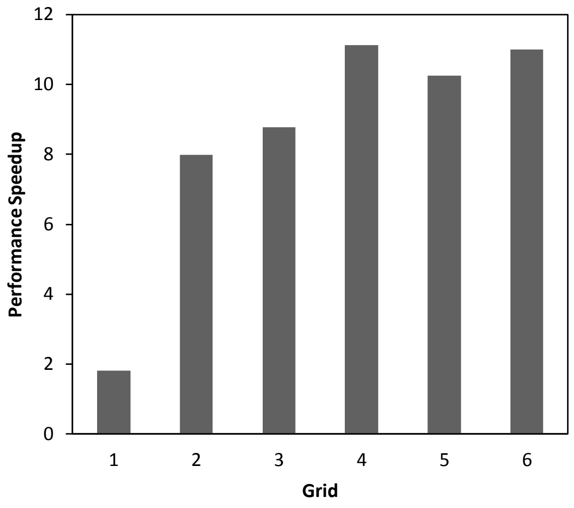

Figure 4: Speedup for PCG on GPU using full-matrix CSRB on both GPU and CPU for different mesh sizes

constant speedup in performance. Beyond this critical mesh size, the memory bandwidth becomes the performance limiting factor. The small variation in performance gain for different grid sizes is normal due to parallel nature of the code. A parallel kernel call on the GPU schedules a grid of blocks of threads [9]. A speedup peak appears if the problem size scales perfectly to the number of target GPU's multiprocessors.

We conclude the discussion of this case by considering the impact of CSRB verses non-block CSR on MVM operations. In this case, MVM speedup is define as

$$speedup = \frac{CSR\_gpu\_execution\_time}{CSRB\_gpu\_execution\_time}, \tag{2}$$

which effectively compares CSRB performance relative to CSR. Results are shown in 6. For larger meshes, the CSRB results in MVM operations that are five times faster than CSR MVM operatations, which is most likely due to more efficient memory access. Here we can clearly see the choice of CSRB is essential for getting maximum performance from the GPU hardware when solving matrices resulting from finite element discretizations of multidimensional problems.

In the final geometry case, we chose an object that is more representative of an applied finite element model. A model of a steel bracket, shown in Figure 7, was used. As a preprocessing step in the finite element process, the computational domain defined by the solid bracket was discretized. The commercial grid generator Pointwise V17 was used to generate a grid composed of 688,190 hexahedral elements and 837,042 nodes. In order to reduce the overall cell count, only half of the bracket was discretized using multiblock grid generation techniques. The resulting multiblock structured grid topology coupled with Pointwise's elliptic PDE methods allowed for precise control of both local and global grid quality.

These elliptic methods were used to iteratively solve Poisson's equation using boundary control functions to improve boundary orthogonality and interior control

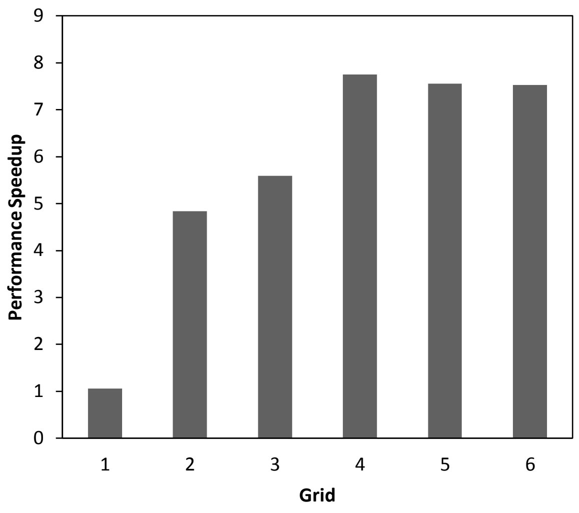

Figure 5: Speedup for PCG on GPU using full-matrix CSRB on GPU and symmetric CSR on CPU for different mesh sizes

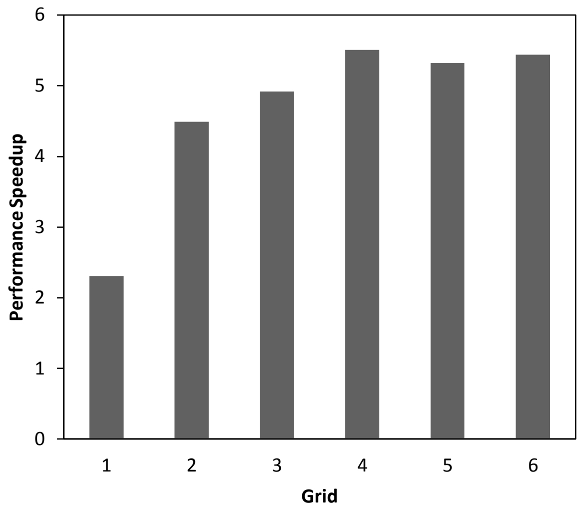

Figure 6: Speedup of MVM for CSRB vs CSR on GPU for different mesh sizes

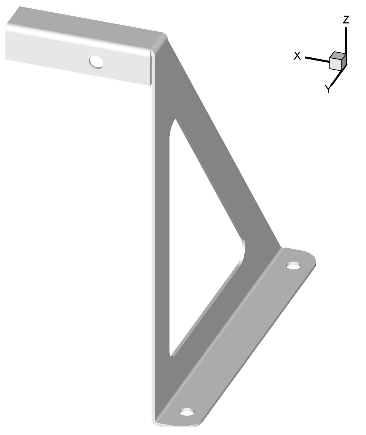

Figure 7: Solid model of a bracket

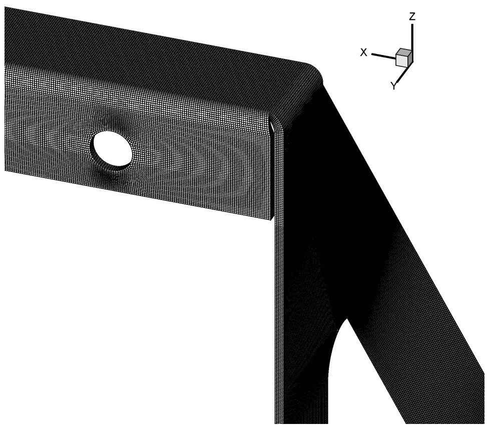

Figure 8: View of the mesh for the bracket

functions to improve grid point clustering and overall smoothness. Additionally, by allowing inner block boundaries and grid lines to float while enforcing boundary orthogonality, improved element quality was realized. A closeup view of the mesh is shown in Figure 8.

For this case, the GPU PCG solver required 60 ms/iteration while the CPU PCG solver required 337 ms/iteration, which represents a speedup of 5.6. This speedup is slightly lower than the beam case with grid 6. We attribute this difference to the more irregular structure of the sparse matrix in the bracket case, which effects the efficiency of memory access.

7 Conclusion

This paper demonstrates the feasibility of using consumer GPUs to solve large sparse systems arising from 3-D finite element discretizations of multidimensional equations. The approach exploits the sparse block structure of the global matrix to achieve speedups exceeding ten compared to a serial code executed on a CPU. The speedup improves up until one million DOF, after which it becomes nearly constant. A speedup of ten was achieved for a beam, while a speedup of 5.6 was achieved for a bracket. Our results show the potential of using GPUs to simulate larger and more complex phenomena on a single workstation without the need for mesh partitioning or domain decomposition.

Acknowledgment

We thank the NVIDIA Corporation for providing the Quadro 5000 GPU used in this research.

References

[1] A. Corrigan, F. Camelli, R. Lohner, J. Wallin, "Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware", in *19th AIAA Computational Fluid Dynamics*, number 2009-4001. San Antonio, Texas, June 2009.

[2] J. Cohen, M. Molemaker, "A Fast Double Precision CFD Code Using CUDA", in *Proceedings of Parallel Computational Fluid Dynamics 2009*, May 2009.

[3] D. Jespersen, "Acceleration of a CFD Code with a GPU", Technical Report NAS-09-003, NAS, November 2009.

[4] *CUDA C Programming Guide Version 4.0*, NVidia, 2011.

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, MA, 1996.

[6] J. Bolz, I. Farmer, E. Grinspun, P. Schroder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid", *ACM Transactions on Graphics (TOG)*, 22 (3): 917–924, July 2003.

[7] "CUDA Toolkit 4.0 CUBLAS Library", Technical Report PG-05326-040-v01, NVIDIA Corporation, April 2011.

[8] T.J.R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Dover Publications, Inc., New York, 2000.

[9] J. Sanders, E. Kandrot, *CUDA by Example An Introduction to General-Purpose GPU Programming*, Addison-Wesley, Upper Saddle River, NJ, 2011.