Arch. Mech., **63**, 2, pp. 137–161, Warszawa 2011

GPU-based simulation of 3D blood flow in abdominal aorta using OpenFOAM

Z. MALECHA[1)], Ł. MIROSŁAW[2),3)], T. TOMCZAK[4)], Z. KOZA[5)], M. MATYKA[5)], W. TARNAWSKI[2),6)], D. SZCZERBA[7)]

[1)] *Faculty of Power and Mechanical Engineering*
Wrocław University of Technology, Poland
e-mail: ziemowit.malecha@pwr.wroc.pl

[2)] *Vratis Ltd., Klecińska 125, Wrocław, Poland*
e-mail: lukasz.miroslaw@vratis.com

[3)] *Institute of Informatics, Wrocław University of Technology, Poland*

[4)] *Institute of Computer Engineering, Control and Robotics*
Wrocław University of Technology, 50-370 Wrocław, Poland
e-mail: tadeusz.tomczak@pwr.wroc.pl

[5)] *Institute of Theoretical Physics, University of Wrocław*
50-204 Wrocław, Poland
e-mail: zkoza@ift.uni.wroc.pl, maq@ift.uni.wroc.pl

[6)] *Chair of Systems and Computer Networks*
Wrocław University of Technology
50-370 Wrocław, Poland,
e-mail: wojciech.tarnawski@vratis.com

[7)] *ITIS Foundation, Switzerland*
e-mail: dominik@itis.ethz.ch

The simulation of blood flow in the cardiac system has the potential to become an attractive diagnostic tool for many cardiovascular diseases, such as in the case of aneurysm. This potential could be reached if the simulations were to be completed in hours rather than days and without resorting to the use of expensive supercomputers. Therefore we have investigated a possibility of accelerating medical computational fluid dynamics (CFD) simulations using graphics processing units (GPUs). Our results for the 3D blood flow in the human abdominal aorta show that by transferring only a part of the computations (linear system solvers) to the GPU, it is possible to make the typical CFD simulations three to four times faster depending on the CFD model being used. Since these simulations were performed on widely available GPUs that had been designed as mass-market PC extension cards, our results suggest that porting larger parts of CFD to GPUs could really bring the technology into hospitals.[*)]

Key words: GPU, CUDA, CFD, computational fluid dynamics, aortic abdominal bifurcation, biomedical simulations, hardware acceleration, parallel computing, general-purpose computation on graphics hardware, GPGPU.

[*)] The paper was presented at 19th Polish National Fluid Dynamics Conference (KKMP), Poznań, 5–9.09.2010.

1. Introduction

Computational fluid dynamics (CFD) simulations are receiving more and more attention from the medical sector due to their ability to mimic effects that directly or indirectly impact the diagnosis. Recent advances in CFD have shown that simulations of the blood flow and pressure field in the organ of interest achieve the point where practical conclusions can be derived, even for a given patient. Simulations of the cerebrospinal fluid flow [1], aortic bifurcation [2], mechano-chemical model of a solid tumor [3], mechanism and localization of wall failure during abdominal aortic aneurysm formation [4], and vascular system simulations in arbitrary anatomies [5], are just a few examples of the current academic research. CFD computations also have the potential to become an attractive option for medical diagnosis and interventional planning; for example, oscillatory flow disturbances may influence the formation of an aortic aneurysm [4], and information about flow conditions in different parts of the aorta may help doctors identify high-risk cases that require surgical treatment. The blood flow in arteries is also interesting in itself for its oscillatory and even turbulent nature.

The aortic bifurcation is an example of a case where the arterial flow is characterized in terms of complex geometry, with pressure fluctuations and shear stress varying both in time and space, and is therefore the most vulnerable to injuries [4]. Medical image acquisition systems such as angiography, computed tomography and magnetic resonance imaging (MRI), allow the construction of patient-specific, 3D models of the geometry of organs and vessels. Doppler ultrasonography and phase-contrast MRI (PC-MRI) provide partial flow information that can be used to construct the boundary conditions. Before the simulation is launched, the computational mesh needs to be generated to cover the complex geometry and the partial differential equations governing the blood velocity and

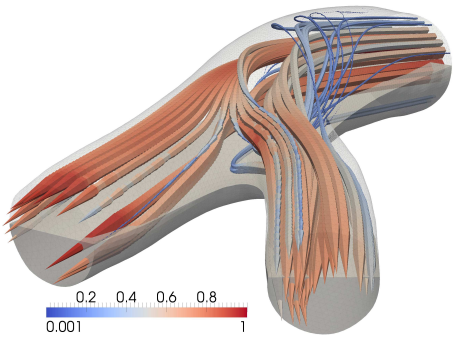

Fig. 1. The steady-state solution to the fluid flow through abdominal aorta. The figure shows selected streamlines coloured by the magnitude of the velocity vectors. The solution was obtained with the GPU-accelerated Semi-Implicit Method for Pressure Linked Equations (SIMPLE) using OpenFOAM [6] and SpeedIT [8] libraries.

pressure need to be carefully discretized to avoid numerical instabilities. Next, the numerical solutions must be stable in time, and be of sufficient quality to rely on in medical diagnosis. Finally, it must be possible to complete the simulations in hours rather than days or weeks.

To reduce computational time, a significant effort is being made to parallelize the medical CFD simulations using state-of-the art numerical methods [2]. Still, the simulations are time-consuming and their direct applicability in medical treatment is highly limited. Therefore in this paper we examine the possibility of accelerating a part of the simulations in a special, massively parallel hardware – graphics processing units.

Modern graphics processing units (GPUs) are many-core processors which, aside from their main application in computer graphics, can also be used in general purpose computations. In contrast to central processing units (CPUs), which currently contain up to eight general-purpose cores and are connected to the main memory through sophisticated, hierarchical caches, GPUs contain *hundreds* of rather simple computational cores connected to a very wide and highly clocked external memory bus. These design features result in computational power and memory bandwidth exceeding that of the fastest multi-core CPUs by almost an order of magnitude. However, this tremendous potential can only be exploited in computations which can be parallelized into at least tens of thousands of threads, with a very regular memory access pattern and small on-chip memory footprint. Fortunately, CFD algorithms contain many steps which satisfy these conditions, and so their acceleration in GPUs is certainly worth trying.

In this paper we report the results of our attempt to accelerate a well-defined and self-contained part of CFD simulations in a GPU. To this end we chose to accelerate two iterative solvers of large systems of sparse linear equations: Conjugate Gradients (CG) and Bi-Conjugate Gradients Stabilized (BiCGStab). These two methods are standard building blocks of CFD simulations based on the Finite Difference Method (FDM), Finite Element Method (FEM) or Finite Volume Method (FVM). As these solvers spent most of their time working out sparse matrix-vector products (SmVP), we performed tests of our GPU SmVP implementation and compared its efficiency with that offered by a CPU implementation from the Intel Math Kernel Library. Next, we linked our GPU-accelerated iterative solvers with OpenFOAM (Open Source Field Operation and Manipulation) [6], an FVM-based, open-source C++ library for solving partial differential equations (PDEs) in arbitrary 3D geometries. Then we used this to solve the flow problem in the human abdominal aorta with three flow models of increasing complexity: potential, steady and unsteady, each solved with a different computational method, respectively: direct, SIMPLE (Semi-Implicit Method for Pressure Linked Equations), and PISO (Pressure Implicit Splitting Operator) [9–13].

We found that although our GPU implementation of CG and BiCGStab solvers used rather simple preconditioners, whereas the original OpenFOAM CPU implementation uses state-of-the art preconditioners, using the GPU-accelerated iterative solvers alone resulted in the four-fold acceleration of the direct CFD solver and in the three-fold acceleration of the SIMPLE method. These results are very promising, as they suggest that porting larger parts of CFD simulations to GPUs could lead to significant acceleration in computation and bring the CFD technology into hospitals as a routine diagnostic method.

The structure of the paper is as follows: the remaining part of the current section contains a basic introduction to using GPUs as general-purpose computing devices (Sec. 1.1), a brief review of recent research on accelerating CFD computations with GPUs (Sec. 1.2), the presentation of iterative linear solvers and their role in CFD simulations (Sec. 1.3), and some details of our GPU implementation (Sec. 1.4). Section 2 presents results of general performance tests of our GPU code. Our results for the human abdominal aorta are reported in Sec. 3. Finally, Sec. 4 presents conclusions and outlook.

1.1. GPUs as general-purpose computing devices

Graphics processing units, or GPUs, were invented in the late 1990s as fixed-pipeline application-specific integrated circuits (ASICs) designed to accelerate computations necessary in 3D real-time graphics. The very idea of building a hardware graphics accelerator came from the observation that rendering a 3D scene requires a well-defined sequence of computational steps (the so-called graphics pipeline), which need to be applied to large streams of easily separable data, e.g. vertex coordinate and color descriptors, a task that in principle could and should be parallelized to the extent unattainable for standard CPUs. Since then, every two or three years, GPU designers have released new and revolutionary architecture which not only allows the users to enjoy increasingly life-like graphics effects, but which also renders the devices more flexible and easier to program. Although the first programmable GPUs appeared in the early 2000s, this feature was hardly noticed outside graphics engineering, for it required the mastering of special-purpose assembly languages and any general-purpose computation had to be expressed in terms of graphics primitives. The situation changed completely in 2006, when NVIDIA released its first fully programmable GPU processor, GT80, together with all the software tools sufficient to write GPU programs in a popular, high-level computer language (C or C++). This new programming environment is called CUDA [14]; ATI Stream is a similar product for AMD ATI graphics cards [15].

Typical examples of modern GPUs are the GF100 and RV790 processors. The former has up to 512 computational cores which can deliver ≈ 1 TFLOPS of com-

putational power for single precision floating point numbers and ≈ 0.5 TFLOPS for double precision floating point numbers. It can also access the GPU memory at 177 GB/s. The latter includes 800 cores, which can deliver 1.4 TFLOPS in single precision and can access the main memory at 125 GB/s. Note that the fastest eight-core CPUs can deliver about 0.15 TFLOPS in single precision and can access their main memory at about 25 GB/s.

Although GPUs apparently surpass CPUs in computational power, neither end-users nor programmers are rushing to replace their CPUs with GPUs, and this is for a good reason. The tremendous computational power of GPUs comes with a price. Firstly, GPUs are designed as co-processors to CPUs and cannot run programs by itself nor communicate with any external device other than the host CPU. The launching of GPU programs ("kernels"), as well as GPU memory management, can only be performed by the CPU. Secondly, GPU cores are arranged in a multi-level hierarchical structure. Essentially, all cores must execute the same program and all cores on the lowest hierarchical level must execute the same instruction or stay idle. Thirdly, the GPU memory has a very special organization which works optimally if the data are accessed in order, as if in streams of consecutive items. If different cores want to simultaneously access memory addresses located in different 256-byte long memory chunks, the main memory bandwidth drops by an order of magnitude. Moreover, to hide large DRAM memory latencies, GPUs can launch in hardware tens or even hundreds of threads per each physical computational unit, all resident in the on-chip memory which implies that the number of registers available to each thread is quite limited, a phenomenon known as "register pressure". For this reason, GPU kernels tend to be rather small programs launched from the CPU in sequence, and the efficiency of on-chip memory utilization is of highest priority. Fourthly, the PCI-Express bus connecting GPUs to the CPUs is relatively slow (up to 8 GB/s), much slower than the internal buses in CPUs and GPUs. Lastly, even though GPUs can be programmed in slightly extended high-level languages, they cannot run routines written in these languages for CPUs. Moreover, a vast majority of existing C or C++ libraries are targeted at either serial processors or parallel systems adhering to a completely different parallelization model and hence their porting to massively parallel GPUs can be a very nontrivial, time-consuming, error-prone task. The number of production-quality libraries for GPUs is thus very limited. This means that practically all numerical software for GPUs must be written from scratch.

Taking these restrictions into account, one concludes that efficient GPU acceleration is possible only for problems with large, homogeneous data sets that can be easily processed with at least tens of thousands of independent, identical threads sharing the data only occasionally. Effective GPU-accelerated programs should also minimize data transfers between the CPU and the GPU and between

GPU cores and the main GPU memory, so that the cost of these transfers could be compensated by a large amount of computations performed in the GPU. It is also clear that the acceleration of CFD simulations, which are usually based on complex CPU codes developed in decades by large groups of highly skilled professionals, must be carried out in small steps targeting rather small, self-contained areas, critical to the overall performance of CFD programs. In accordance with this observation, in this paper we restrict GPU acceleration to sparse linear system solvers and investigate how this can impact the performance of three basic OpenFOAM CFD solvers.

1.2. Applications of GPUs in CFD simulations

Since the data sets used in CFD problems are typically very large and easily separable into small chunks that can be processed in parallel by individual computing threads, CFD belongs to the area where the GPU architecture can be expected to show its advantages. This is confirmed by the available research on this subject.

In 2008 TÖLKE and KRAFCZYK [16] reported that a GPU-accelerated implementation of the Lattice-Boltzmann solver applied to the problem of a moving sphere in a cylindrical pipe at various Reynolds numbers was ≈ 100 times faster than a similar implementation run on the Intel Xeon (3.4 GHz) CPU. They also found that their implementation achieved as much as 44% of the peak GPU computational power and 61% of its peak memory bandwidth. The success of the Lattice-Boltzmann approach is closely related to the fact that it uses regular lattices for which strict memory access requirements of GPU processors are easier to satisfy than in methods based on an irregular computational mesh (e.g. FEM).

In [17], GPUs were used to simulate the flow over a hypersonic vehicle. To this end, the compressible Euler equations were solved with a multi-grid, vertex-based, finite-difference method. Depending on the complexity of the geometry, the NVIDIA 8800GTX GPU was demonstrated to be between 20 and 40 times faster than the Intel Core 2 Duo CPU.

In [18], the Boussinesq approximation of the Navier–Stokes equations was implemented on the GT 200 GPU processor. The maximum problem size ($384 \times 384 \times 192$) was limited by the GPU memory size (4 GB). The performance acceleration against an eight-core Intel Xeon E5420 2.5 GHz system ranged from 2 for small problems to 8.5 for the largest system.

In [19, 20], a solver for the Navier–Stokes equations with up to four GPUs was proposed. Its performance was compared against two CPU-based systems: one with 2 Intel Core 2 Duo (E8400) 3.0 GHz CPUs and the other one with 4 AMD dual-core Opteron 2.4 GHz (8216) processors. A single GPU implementation was

12.6 times faster than the implementation of a single core Intel CPU. Adding more GPUs yielded an almost linear performance growth – a 4 GPU system was 3 times faster than a 1 GPU system and about 100 times faster than a single 4-core Opteron.

An interesting report on multi-GPU clusters recently appeared in [21]. Several mixed MPI-CUDA implementations were tested to investigate the best strategy of developing an efficient and scalable method of accelerating incompressible flow computations on large, mixed CPU-GPU clusters. The tests were performed on a 64-node cluster with 128 GPUs, each with 240 computational units. In this heterogeneous configuration, it was possible to obtain a sustainable performance of 2.4 TeraFLOPS.

In [22], a possibility of GPU acceleration of a Navier–Stokes solver for a stationary laminar flow was investigated. The analysis was carried out on a 4-node cluster, each node equipped with a dual core AMD Opteron X2 Santa Rosa 1.8 GHz with 6.4 GB/s memory bandwidth and an NVIDIA GeForce 8800 GTX GPU with 86.4 GB/s memory bandwidth. The application of GPU coprocessors resulted in a rather small performance gain (160%), which was caused by two factors. Firstly, the GT 8800 processor cannot handle double precision numbers. Secondly, a complex algorithm used in that study required intensive communication between GPU and CPU memories.

The main conclusion emerging from these results is that modern GPU processors allow for up to, approximately, a tenfold performance increase in typical CFD simulations against eight-core CPUs. Methods based on regular lattices, e.g. the lattice Boltzmann method, allow for even greater acceleration. An additional advantage of GPU acceleration is its good scalability with the number of GPUs in use.

1.3. Iterative linear solvers

The most popular numerical methods of solving flow equations consist of three main steps. Firstly, continuous time and space are approximated by finite sets of discrete points (mesh generation). In the second step, the partial differential equations governing the flow are discretized on the mesh points leading to a large set of linear or non-linear algebraic equations. Finally, algebraic equations are solved using iterative or direct methods.

The Navier–Stokes equations are non-linear, but a common technique is to linearize them during the discretization step, which leads to a set of linear algebraic equations with solution-dependent coefficients [12, 23, 24]. The main reason for using linearization methods is the fact that linear algebraic equations can be solved by using very efficient methods of computational linear algebra.

From this description, it is clear that linear solvers are the first candidate for GPU acceleration in CFD programs. Linear systems found in CFD problems have a few common features. The number of unknowns in these equations is proportional to the number of mesh vertices and is very large, often exceeding 10^6. Each equation includes only variables from neighboring vertices, and hence the system is sparse. The computational meshes are usually irregular, so the matrix of the linear equations is not only sparse, but also unstructured.

Thus, our problem is efficiently using a GPU to solve a system of linear equations of a general form

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1.1}$$

where $\mathbf{A}$ is a huge, unstructured, sparse, real matrix, and $\mathbf{x}$, $\mathbf{b}$ are real vectors. A number of special techniques for such systems has been developed. Among them, Krylov subspace iterative methods have become a *de facto* standard for large linear problems. A thorough review of these methods can be found in [25], and many technical details of their computer implementations are discussed in [26]. We chose to accelerate two popular Krylov subspace methods: Conjugate Gradient (CG) and Biconjugate Gradient Stabilized Method (BiCGStab). Below, we present only those of their properties which are relevant to our present work.

All Krylov subspace methods are based on the observation that the product of a sparse matrix $\mathbf{A}$ with any vector $\mathbf{x}$ is a relatively cheap operation that can be completed in a time proportional to the number of nonzero elements of $\mathbf{A}$. Thus, it is not difficult to construct a sequence of vectors $\mathbf{x}, \mathbf{A}\mathbf{x}, \mathbf{A}^2\mathbf{x}, \ldots$. By taking their linear combinations it is then easy to compute any member of the so-called Krylov subspace

$$\mathcal{K}_r(\mathbf{A}, \mathbf{x}) = \operatorname{span}\{\mathbf{x}, \mathbf{A}\mathbf{x}, \mathbf{A}^2\mathbf{x}, \ldots, \mathbf{A}^{r-1}\mathbf{x}\} \tag{1.2}$$

spanned by these vectors. A sequence of approximate solutions $\mathbf{x}_r = \mathbf{x}_0 + \mathbf{w}_r$ to Eq. (1.1) is then computed, where $\mathbf{x}_0$ is the initial guess (often chosen to be the zero vector) and $\mathbf{w}_r$ is a vector from the Krylov subspace $\mathcal{K}_r(\mathbf{A}, \mathbf{A}\mathbf{x}_0 - \mathbf{b})$. There are many possible algorithms of generating "the best" sequence $\mathbf{w}_r$ for which the iterates $\mathbf{x}_r$ will quickly converge to the exact solution. These methods often use some additional information on $\mathbf{A}$, such as whether it is symmetric and whether it is positive definite, etc.

The Conjugate Gradient (CG) method is applicable only to systems with a symmetrical, positive definite matrix $\mathbf{A}$. It involves only one matrix-vector multiplication per iteration, and requires storage for just a few temporary vectors rather than the whole sequence $\{\mathbf{x}_r\}$. Systems with general nonsingular sparse matrices can be solved using the Biconjugate Gradient Stabilized Method (BiCGStab). This method requires two matrix-vector multiplications per iteration and, like CG, storage for a limited number of vectors.

Since all iterative methods aim at finding an approximate rather than exact solution, they usually involve a second matrix $\mathbf{M}$, called a preconditioner, the role of which is to transform the original problem into one with improved convergence properties. This idea is based on an observation that equation

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \tag{1.3}$$

is mathematically equivalent to (1.1), but for a well-chosen $\mathbf{M}$, the transformed equation can be solved more quickly. A good preconditioner should satisfy two conditions. Firstly, it should resemble the original matrix $\mathbf{A}$, so that $\mathbf{M}^{-1}\mathbf{A}$ would approximate the identity matrix. Secondly, it should be possible to solve $\mathbf{M}\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ far more efficiently than $\mathbf{A}\mathbf{x} = \mathbf{b}$ (which is necessary to work out expressions like $\mathbf{M}^{-1}\mathbf{b}$). Although constructing and applying preconditioners usually implies a substantial computational overhead, a well-chosen preconditioner can greatly improve the convergence rate of iterative methods, which fully compensates the overhead. Actually, in many practical cases, iterative methods fail to converge without a preconditioner.

There are only a few theoretical results concerning the rate of convergence of the Krylov subspace methods. For example, assuming the exact computer arithmetic, the upper bound for the number of iterations to reach a given relative error in the solution of (1.3) using the CG method is proportional to $\sqrt{\kappa_2}$, where κ_2 is the spectral condition number of $\mathbf{M}^{-1}\mathbf{A}$. Thus, the role of a good preconditioner is to reduce the spectral condition number of the coefficient matrix, which in turn not only speeds up the convergence, but also reduces the round-off errors. An example of a preconditioner is the Jacobi preconditioner defined as the diagonal part of $\mathbf{A}$, $\mathbf{M} = \mathrm{diag}(\mathbf{A})$.

1.4. GPU implementation

GPU vendors offer several programming frameworks for general-purpose GPU computing. Currently, the most advanced of them are NVIDIA CUDA (proprietary, free) and OpenCL (open standard). We chose CUDA [14], as it appears to be the most mature solution for GPU programming, offering such programming tools as a debugger and profiler. An important and unique feature of CUDA is its full support for C++ templates, which are widely used in our code.

Using CUDA, we ported to GPU the CG and BiCGStab iterative solvers, as well as the Jacobi (diagonal) preconditioner. Both solvers can be used in single and double floating point precision and can be used with or without the preconditioner. It is also possible to use them to solve systems of linear equations in complex arithmetics. All matrix-vector and vector-vector operations are performed in the GPU. Implementation of the SmVP operation is based on the

vector and scalar CSR kernels with data buffering through a texture cache proposed in [27]. The vector-vector scalar product is calculated using the NVIDIA CUBLAS library, and the remaining kernels are custom codes.

The GPU-accelarated linear solvers were then integrated with OpenFOAM by subclassing their `lduMatrix::solver` class. Whenever the CG or BiCGStab solver was about to be called by OpenFOAM to solve (1.1), we converted the computer representation of $\mathbf{A}$ from the OpenFOAM format to the CSR [25, 26] format, transferred it together with vectors $\mathbf{x}$ and $\mathbf{b}$ to the GPU memory, called our GPU solver, and copied the result, i.e. $\mathbf{x}$, back to the CPU memory. The matrix format conversion is carried out in the CPU, which imposes some computational overhead. We chose the CSR format because of its simplicity and generality, which minimized the format conversion overhead.

2. Performance tests

2.1. Hardware and software specification

All performance tests were done on a desktop PC computer with a four-core Intel(R) Core 2 Quad CPU Q8400 clocked at 2.66 GHz, connected to 4 GB of a dual channel DDR2 memory clocked at 800 MHz and offering the peak bandwidth of 12.8 GB/s. The GPU device was an EVGA GeForce GTX 275 graphics card with a GT 200 processor clocked at 633 MHz (240 CUDA cores), connected to 1792 MB of DDR3 448bit memory. In this configuration, the measured peak GPU bandwidth was 103 GB/s and the theoretical peak GPU computational performance was $\approx$ 0.9 TFLOPS (single precision) and $\approx$ 76 GFLOPS (double precision). The measured peak bandwidth of the GPU-CPU bus (PCI-Express x16 gen 2) was 3.1 GB/s for non-pageable memory and 2 GB/s for pageable memory. All bandwidth measurements were done with *bandwidthTest* utility from CUDA SDK. The operating system was an Ubuntu 9.04 Desktop x86-64 with CUDA 2.3 and NVIDIA driver v. 190.53.

2.2. Test procedure

Our GPU-accelerated linear solvers were tested in two scenarios. In the first scenario, we chose a set of test matrices and for each solver, we measured its acceleration as compared to the corresponding CPU implementation. Next, the solvers were integrated with OpenFOAM to determine how much they could accelerate solutions of typical medical CFD problems.

The test matrices, representing typical matrices found in 2D and 3D problems, are listed in Table 1. These are unstructured, real and sparse matrices of medium to large size, with various algebraic properties, including symmetric positive definite, symmetric indefinite, and non-symmetric matrices. They

Table. 1. The basic parameters of the test matrices from [28] used in the GPU performance measurements. n_{nz} denotes the number of matrix nonzero elements, m denotes the number of matrix rows, and the ratio n_{nz}/m gives the average number of nonzero elements per matrix row.

Matrix name	n_{nz}	m	n_{nz}/m
af_shell10	52 672 325	1 508 065	34.9
apache1	542 184	80 800	6.7
audikw_1	77 651 847	943 695	82.3
F1	26 837 113	343 791	78.1
Freescale1	18 920 347	3 428 755	5.5
hood	10 768 436	220 542	48.8
nd12k	14 220 946	36 000	395.0
nd24k	28 715 634	72 000	398.8

were selected from the University of Florida Sparse Matrix Collection [28], and originated from various applications, including CFD, circuit simulations and structural analysis. The number of their nonzero elements vary from 542 184 to 77 651 847, the number of rows range from 36 000 to 3 428 755, and the average number of nonzero elements per row lie between 5.5 and 398.8. They require from 7 MB to 0.91 GB of storage and hence can be processed in most modern graphics cards (usually equipped with 1 GB of RAM). Note that only 6 (out of 2272) of the largest matrices listed in [28] would require more than 1 GB of storage.

The computational performance of CPU and GPU implementations was measured in GFLOPS (10^9 floating-point operations per second) and calculated as the ratio of the required number of floating point operations to the time taken by the computations. In the calculations of the number of floating point operations (computational complexity), each arithmetic operation carried on source operands (elements of vectors or matrices) was counted as one floating-point operation. In this model, the number of floating point operations for the dot product of two m-element vectors was readily found to be equal to $2m - 1$. A similar formula for the computational complexity of sparse matrix-dense vector product reads [8]

$$N_{\mathrm{FLOP}} = 2n_{\mathrm{nz}} - m \tag{2.1}$$

where m is the number of rows in $\mathbf{A}$, and n_{nz} denotes the number of its non-zero elements. Next, the computational complexity of iterative solvers was computed as the sum of computational complexities of each of its elementary internal operations.

2.3. Iterative linear solvers

The computational performance of iterative solvers depends mainly on the efficiency of the sparse matrix-vector product (SmVP) routine and in the case of our GPU implementation, it was found to lie between 3.2 and 15.5 GFLOPS for a single precision and between 2.3 and 12.0 GFLOPS for double precision floating-point arithmetic. As seen in Fig. 2, comparison with a state-of-the art CPU implementation from the Intel Math Kernel Library run on a 4-core Intel CPU showed that the GPU-based version is between 7.3 and 14.5 times faster for single precision, and between 5.1 and 15.9 times faster for double precision, depending on the matrix structure.

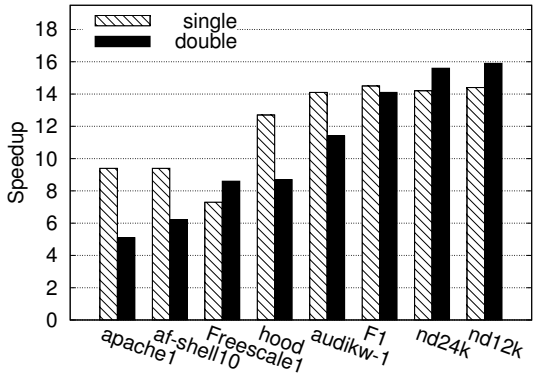

Fig. 2. Acceleration of the SmVP routines for the test matrices from Table 1 and two floating-point precisions, single and double. CPU-GPU data transfer overhead is not taken into account.

Next, the computational performance of linear solvers was tested. For each combination of the floating-point precision, preconditioner, and test matrix, we ran the CG and BiCGStab solvers on both hardware platforms (CPU-only and GPU-accelarated CPU). Note that although it is impossible to solve Eq. (1.1) using the CG method for some of the test matrices listed in Table 1, we used them in the tests to examine the impact of their structure on the solver computational performance.

The peak computational performance of our linear solvers was found to vary between 2.3 and 15.2 GFLOPS for single precision and between 1.5 and 11.5 GFLOPS for double precision. These results are similar to the performance of the SmVP routine, which takes up most of the solver run time.

The relation between the performance of the BiCGStab solver and the number of its internal iterations is shown in Fig. 3a for the two matrices with the highest and lowest performances. The CPU-GPU memory transfer overhead was not taken into account in this figure. Notice that after a few iterations, the computational performance stabilized at the maximum level. An initial, small

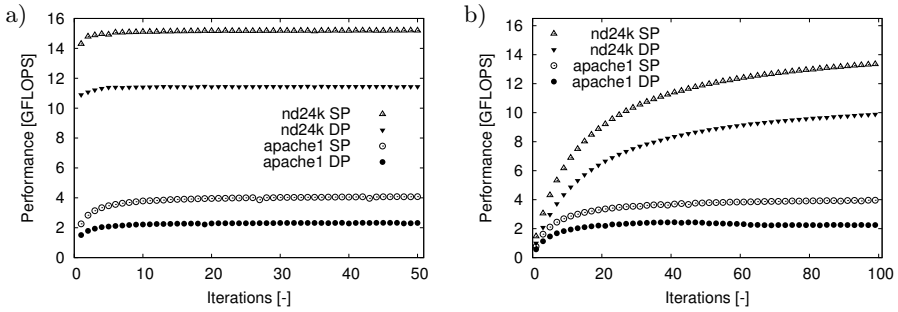

Fig. 3. GPU-accelerated BiCGStab solver performance as a function of the number of solver internal iterations a) without and b) including the CPU-GPU data transfer overhead.

performance drop was caused by the time necessary for the solver's internal data allocation and initialization.

Figure 3b presents the performance of the BiCGStab solver as a function of the number of solver iterations, with the data transfer overhead taken into account. This overhead is responsible for a lower performance if the number of iterations is small. As the number of iteration increases, the performance of the solver converges asymptotically to the maximum value.

To analyze the cost of using the diagonal preconditioner, the time overhead introduced by this preconditioner was defined as

$$\frac{t_{\text{diag}}}{t_{\text{none}}} - 1, \tag{2.2}$$

where t_{diag} is the time taken by the diagonally preconditioned CG solver, and t_{none} is the time taken by this solver without being preconditioned. The results are shown in Fig. 4a. The maximum performance deceleration caused by the diagonal preconditioner may be as high as approximately 40% for a small number

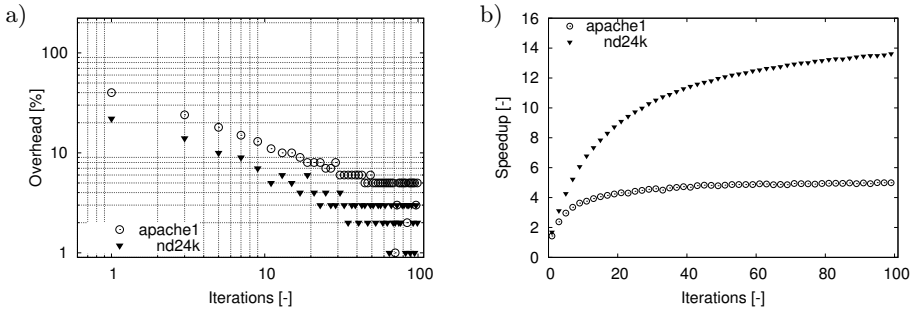

Fig. 4. a) Time overhead of the diagonal preconditioner for the single precision CG solver, computed from Eq. (2.2), b) time ratios of the double-precision, diagonal-preconditioned, GPU-accelerated BiCGStab solver to the corresponding CPU implementation. In b) the CPU-GPU data transfer overhead is taken into account.

of iterations, but as the number of iteration increases, the diagonal preconditioner overhead drops below 10% for all test matrices. The results for the BiCGStab solver are similar (data not shown).

In order to compare our GPU-accelerated solvers with corresponding CPU implementations, we developed our own highly optimized, multi-threaded BiCGStab solver based on the SmVP routine from the Intel Math Kernel Library and a BiCGStab algorithm from [26]. Results of the performance comparison are presented in Fig. 4b and include the data transfer overhead. The GPU-accelerated solver turned out to be always faster than its CPU equivalent running on a 4-core CPU.

Finally, we compared the times needed to solve three systems of linear equations. In all cases, the solvers were run with the same initial conditions and required error tolerance. The results are shown in Table 2. The acceleration was computed as the ratio of the CPU to GPU solver run times. The data transfer overhead was included in the times for the GPU-accelerated solver. In all the cases, the GPU-accelerated solvers turned out to be significantly faster than a highly optimized CPU implementation run on a 4-core processor.

Table 2. Comparison of double precision GPU and CPU linear solver performances.

Matrix name	Number of iterations		Residuum $[\times 10^{-11}]$		Time [s]		**Speedup**
	CPU	GPU	CPU	GPU	CPU	GPU	
apache1	2014	2157	2.88	5.17	18.37	3.51	5.24
F1	4396	5516	2.88	9.97	1339.59	87.40	15.33
hood	6165	9811	7.67	9.72	686.86	86.99	7.90

3. Flow in abdominal aorta

CFD flow problems can be tackled using various methods which, aside from linear solvers, require many more components. One can expect that the more complex an algorithm is, the smaller the benefits will be from accelerating linear solvers only. Moreover, if only a part of essential algorithms has been implemented on the GPU, the overall computational performance is likely to be affected by low-bandwidth communication between the GPU and its host CPU. To investigate these problems, we decided to solve the flow in the abdominal aorta using three different types of flow models: potential, steady and unsteady. Each was solved with a different CFD method: direct, SIMPLE, and PISO [9–13], respectively. To this end, three OpenFOAM flow solvers were used: potential-

FOAM, simpleFOAM and pisoFOAM. They were modified and recompiled to adjust to our needs.

The flow geometry of the abdominal aorta used in our tests is shown in Fig. 5. It was generated from *Virtual Family* models and originated from high resolution MRI scans (0.5–0.9 mm pixel size, 1–2 mm slice thickness) of healthy volunteers [29]. To obtain a convenient mathematical description of the aorta surface, the geometry was expressed in the STL format which is commonly used to describe unstructured, triangulated surfaces. This data was then used to generate a three-dimensional, computational mesh. In this study, an unstructured tetrahedral mesh was used.

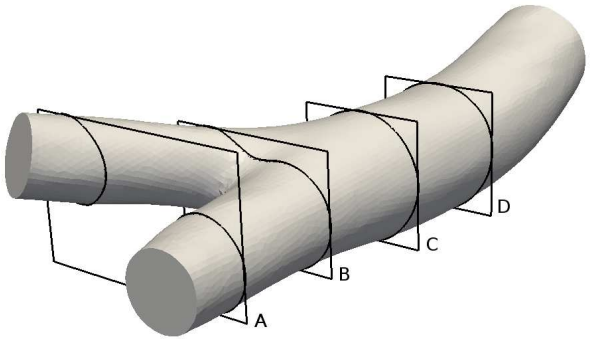

FIG. 5. Geometry of the human abdominal aorta used in the numerical calculations. A, B, C and D are cross-sections of the aorta employed in Fig. 10.

3.1. Acceleration measurement

Generally, the acceleration of a GPU-accelerated CFD solver is defined as the ratio of the original (CPU-only) to the GPU-accelerated OpenFOAM solver runtimes, including the time spent on reading the data from the hard disk. However, evaluation of the acceleration is complicated by the fact that OpenFOAM implements various preconditioners, some of which use sequential algorithms that cannot be efficiently ported to a massively parallel device such as the GPU. Therefore, we investigated two accelerations. In the first one, both GPU and CPU implementations of a particular OpenFOAM solver were forced to use the same preconditioner – the diagonal preconditioner. This test was called "diagonal vs. diagonal". In the second test, both implementations were allowed to use "the best" preconditioners for a given platform. Thus, in this test, the GPU-accelerated implementation used the diagonal preconditioner, whereas the original CPU-only OpenFOAM implementation used more efficient, state-of-the-art preconditioners: either the diagonal incomplete-LU (DILU) or the diagonal incomplete-Cholesky (DIC) preconditioner. Additionally, in the PISO method, the Geometric Agglomerated Algebraic Multigrid (GAMG) solver was used in many steps of the algorithm. This test was called "diagonal vs. other".

3.2. Simple potential flow

Our first test focused on a very simplified flow in the abdominal aorta using PotentialFOAM solver [6]. The PotentialFOAM is the simplest solver among OpenFOAM's applications. It assumes that a flow is irrational, incompressible, inviscid and steady, and is governed by the mass continuity and pressure equations [7]:

$$\nabla \cdot \mathbf{U} = 0, \tag{3.1}$$

$$\nabla^2 p = 0, \tag{3.2}$$

where $\mathbf{U}$ is the velocity vector and p denotes the pressure. The name of the solver and its description given by its authors can be misleading because a velocity potential equation is not solved here. Our main goal with using the PotentialFOAM solver was to show the acceleration of the calculations for problems based on Laplace's or Poisson's equations. Since the above approximation is very simplified, it cannot be used for serious clinical applications. Nevertheless, it can be useful as a utility to provide the initial velocity field for more realistic flow models [7].

The acceleration of the GPU-accelerated potentialFOAM solver is presented in the first bar-graph of Fig. 6. In this case, we can observe that the GPU implementation was approximately 4 times faster than its CPU-only counterpart with a recommended preconditioner. When both CFD solvers were forced to use the same preconditioner, the acceleration increased to ≈ 6 times. Such acceleration

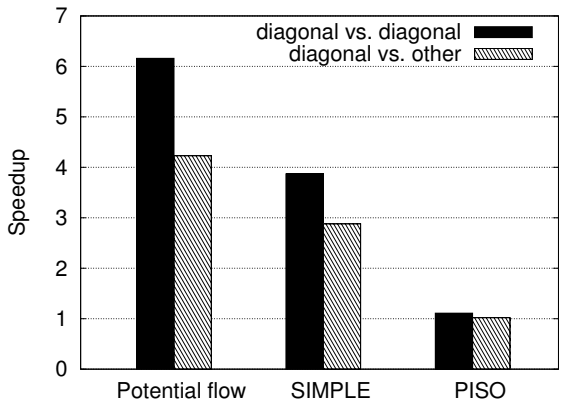

Fig. 6. Acceleration of GPU-accelerated OpenFOAM solvers against the corresponding original CPU implementations with various preconditioners of linear solvers. Dark bars show the acceleration of diagonally preconditioned GPU-accelerated solvers over the CPU implmentations with recommended preconditioners – GAMG for the PISO, and DILU/DIC for the simpleFOAM and potentialFOAM solvers.

was possible because the PotentialFOAM solver was based on a single Poisson equation, which was approximated by a system of linear equations with a constant coefficient matrix $\mathbf{A}$. Hence, all calculations could be carried out on the GPU device.

3.3. Steady incompressible flow

Next, we employed the SIMPLE method to solve the steady and incompressible Navier–Stokes equations

$$\nabla \cdot (\mathbf{UU}) = -\nabla p + \nu \nabla^2 \mathbf{U}, \tag{3.3}$$

where ν is the kinematic viscosity. Here and in the next example, blood was modeled as a Newtonian fluid with a constant density $\rho = 1020$ kg/m^3 and dynamic viscosity $\eta = \nu\rho = 0.003$ kg/ms. In this equation, both velocity $\mathbf{U}$ and pressure p are unknown, so we had to perform outer iterations to calculate pressure and velocity fields, verifying Eqs. (3.1) and (3.3), which is a characteristic feature of the so-called *projection methods* [11, 12].

Another difficulty in solving the Navier–Stokes equations is the nonlinear term $\nabla \cdot (\mathbf{UU})$. In OpenFOAM, this term is replaced with a linear one, $\nabla \cdot (\phi \mathbf{U})$, where ϕ is the velocity flux, which is calculated using the value of $\mathbf{U}$ from the previous iteration.

Figure 1 shows the steady-state solution to the fluid flow through the abdominal aorta with selected streamlines colored by the magnitude of the velocity vectors. At the inlet, a fully developed flow was imposed as the boundary condition. This boundary condition assumes that the velocity profile at the inlet has a parabolic shape with velocity vanishing at the walls and maximum value 1.3 [m/s] in the middle. This value corresponds to the maximum blood velocity in the entrance to the abdominal aorta [2]. To determine the developed-flow condition, we solved an additional Poisson's equation $\nabla^2 U_0(x, y, z) = 1$ exclusively at the inlet patch with the zero Dirichlet boundary condition. The resulting function $U_0(x, y, z)$ was then rescaled to the desired value and prescribed to the normal component of the inlet boundary velocity. A no-slip boundary condition was applied to the walls of the aorta, which were assumed to be rigid, whereas the zero velocity gradient condition was used at the outlets.

As seen in the second bar-graph of Fig. 6, the GPU acceleration for the SIMPLE method was found to be equal to roughly four or three, depending on the linear preconditioner employed in the CPU implementation. These values are a bit smaller than in the case of the potential flow. The main reason for this is the fact that during the outer iteration, it was necessary to update the coefficient matrix used by the linear solver in each iteration. This required frequent data transfers between the GPU and the CPU, which is an inefficient operation.

3.4. Unsteady flow

None of the methods described above is capable of revealing all details of actual flows in the abdominal aorta, mainly because the inflow to the real aorta is pulsatory and the equations used to determine such a flow have to be time-dependent. Thus, we arrive at the full Navier–Stokes equations:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U}\mathbf{U}) = -\nabla p + \nu \nabla^2 \mathbf{U} \tag{3.4}$$

with the time-dependent, fully developed, inlet velocity, boundary condition:

$$U_{\text{in}} = U_0(x, y, z)\,(\sin(2\pi\omega t) + 1) \tag{3.5}$$

where $U_0(x, y, z)$ is the function which is parabolic in shape, calculated in the same way as in the previous section, but with a maximum value of 0.65 m/s. The frequency corresponding to a heart rate of 80 beats per minute [2] is $\omega = 1.333$ Hz. The cross-section of the inlet velocity boundary condition described by 3.5 can be seen in Fig. 7. As before, the no-slip boundary condition was applied to the walls of the aorta, which were assumed to be rigid, whereas the zero velocity gradient condition was used at the outlets.

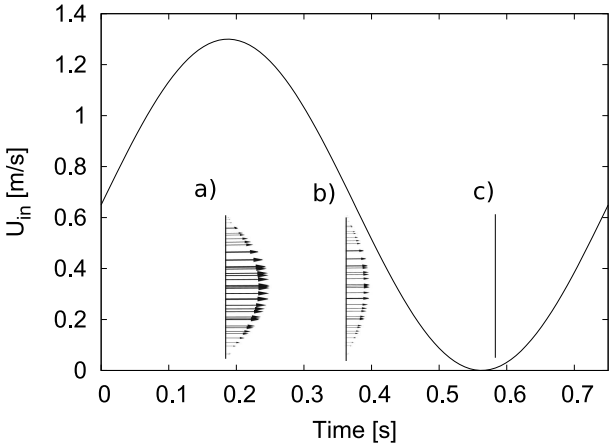

Fig. 7. Time-dependent inlet boundary condition expressed by Eq. (3.5). Arrows represent linear cross-sections from two-dimensional profiles of velocity imposed as the boundary condition at the inlet to the aorta. The profiles represent the developed boundary condition (parabolic profile) for three selected time-steps: $t = 0.18$, 0.37, and 0.58 [s].

In reality, the minimum inlet velocity was measured as slightly negative, revealing a short period of reverse flow. Our boundary condition (3.5) assumes the minimum velocity to vanish, and in this respect slightly differs from the boundary condition proposed in [2].

Equations (3.4) were solved with the PISO method, another example of a projection method which is used, more often, for time-dependent problems other than SIMPLE. In the PISO algorithm, to enforce the continuity equation (3.1), an additional pressure equation must be solved and the velocity field is then corrected [11, 12].

3.5. Results

The unsteady simulations of the flow in the abdominal aorta reveals some interesting features. Figures 8 and 9 show the flow for different times t, corresponding to different phases of the inlet velocity boundary condition (cf. Fig. 7). At $t = 0.18$ s, the inlet boundary velocity is in the increasing phase and is just before it reaches maximum value. At $t = 0.37$ s, the inlet velocity is in the decreasing phase and for $t = 0.58$ s, it is just before it reaches minimum value. After reaching the minimum value, the flow starts a new cycle. Figure 8 shows selected three-dimensional streamlines in which different colors correspond to different magnitudes of the velocity, whereas Fig. 9 represents the velocity magnitude in the same horizontal cross-section as in the Fig. 8.

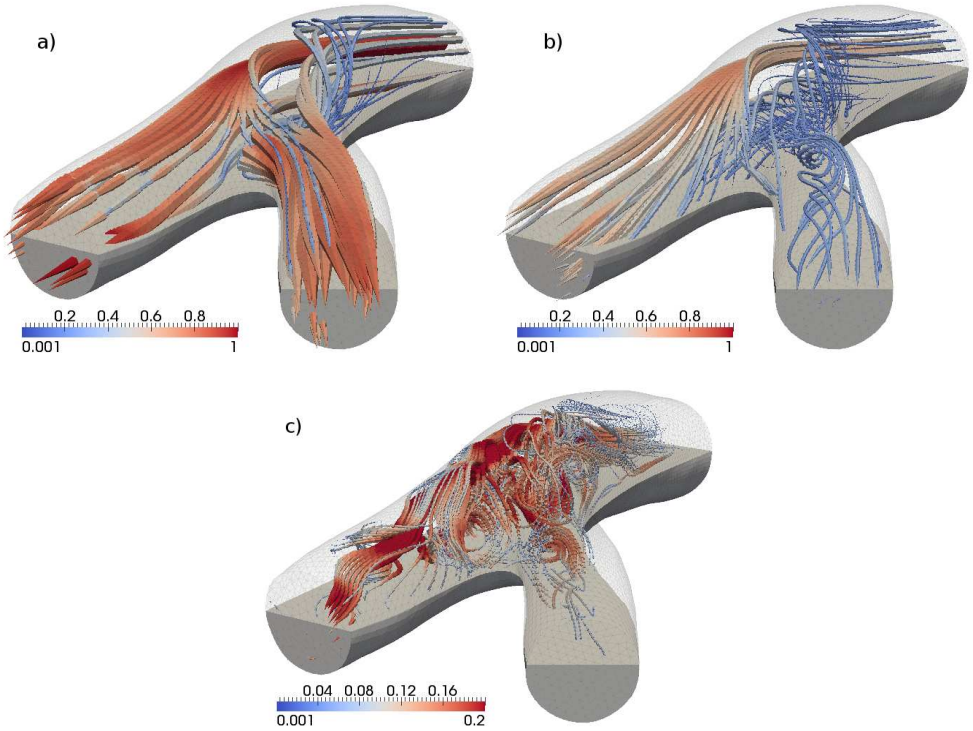

FIG. 8. Flow in abdominal aorta for times: a) $t = 0.18$ s, b) $t = 0.37$ s, c) $t = 0.58$ s. Selected streamlines are shown, with color and intensity representing the velocity magnitude.

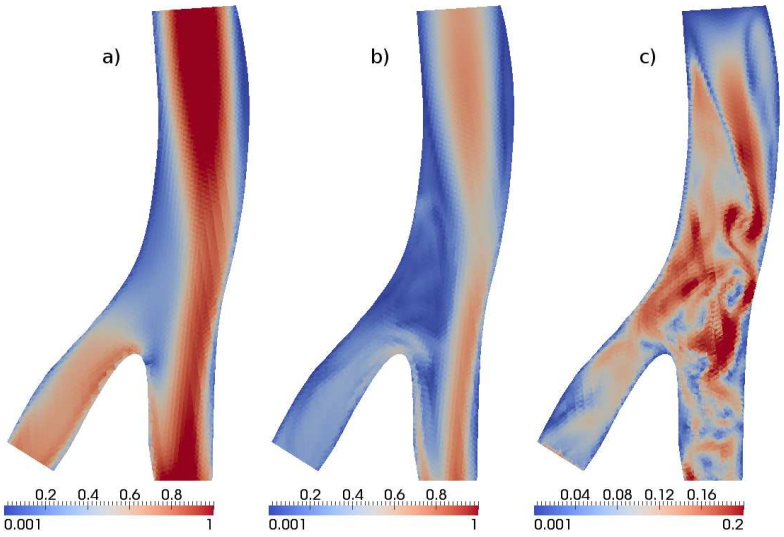

FIG. 9. A cross-section of the abdominal aorta, with the magnitude of the velocity field at times: a) $t = 0.18$ s, b) $t = 0.37$ s, c) $t = 0.58$ s. The cross-section is the same as in Fig. 8.

Analyzing the flow field in time, one can observe alternate periods of ordered and very irregular flow patterns. This is caused by the fact that fluid dynamics is strongly affected by the state of the inlet velocity. In the increasing phase of the inlet velocity, the flow becomes gradually more ordered, and finally (for the maximum inlet velocity) approaches a state similar to the one obtained in the steady solution (compare Figs. 1 and 8a). In Fig. 9a, just before the exit from the aorta, one can observe a characteristic region where the velocity is much slower than in the surrounding fluid. In this region, when the inlet velocity has just started to decrease after reaching its maximum value, a recirculation zone manifests itself, see Figs. 8b and 9b.

During the decreasing phase of the inlet velocity, the flow becomes gradually more complicated and finally reaches a very complex state, see pairs of Figs. 8b, 9b and 8c, 9c. There is no longer one dominant recirculation zone but new vortex structures appear successively with strong mixing characteristics.

Figure 10 shows the magnitude of the velocity field for different times and different cross-sections of the aorta. Each cross-section is marked with letters A, B, C and D respectively, and corresponds to the indicators marked in Fig. 5.

By analyzing the flow patterns in the different cross-sections of the aorta, we can confirm our previous observations. However, we can also observe some new, interesting features. For time $t = 0.18$ s and cross-section D, we can see that the flow is distributed almost uniformly across the section. But when we move closer to the bifurcation region, the main flow shifts from the center of the aorta to one of the sides, as seen in cross-section C. Consequently, we can observe that aorta

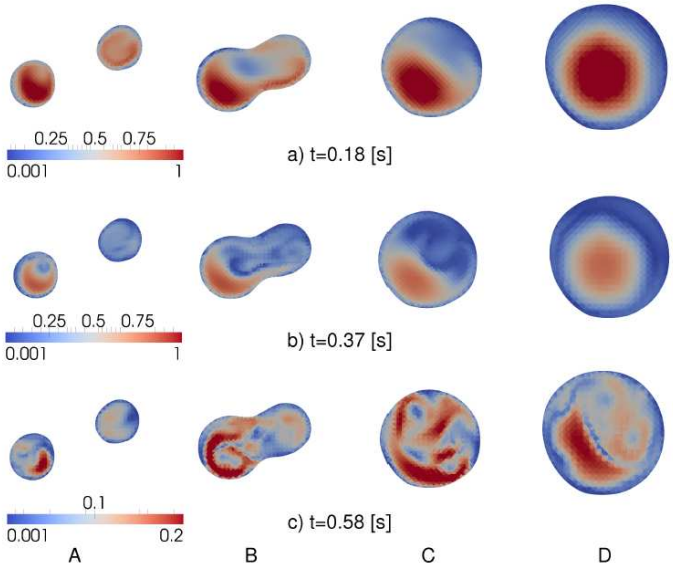

Fig. 10. Magnitudes of the velocity field for different times and cross-sections of the abdominal aorta. The cross-sections are marked as A, B, C, D respectively and correspond to the indicators in Fig. 5.

velocities are higher in one branch than in another, as shown in cross-sections B and A. For time $t = 0.37$ s, the inlet velocity is in the decreasing phase and the flow field becomes more complex, but the flow patterns in the cross-sections are similar to those obtained at $t = 0.18$ s.

For times $t = 0.18$ s and $t = 0.37$ s, we can observe that the regions of low and high velocities are separated and focused in different parts of the aorta. But at $t = 0.58$ s, when the inlet velocity is in the decreasing phase and just before it reaches minimum value, we can observe that regions of low and high velocity penetrate each other, causing those regions to form very complex shapes. It confirms again that the flow in the abdominal aorta has very strong mixing features.

The results for the GPU acceleration are presented in Fig. 6 and show that the acceleration of the PISO algorithm is hardly noticeable. This is a result of the fact that OpenFOAM implements the PISO method using the Geometric Agglomerated Algebraic Multigrid (GAMG) method, which was not ported to the GPU.

4. Conclusions

Recent advancements in computer fluid dynamics (CFD), combined with the availability of high-resolution magnetic resonance images, suggest that the

simulation of blood flow in the cardiac system has the potential to become an attractive diagnostic tool for many cardiovascular diseases. The main problem that still needs to be solved is that such simulations are time-consuming, even if they are carried out on supercomputers. Therefore, we have investigated a possibility of accelerating medical computational fluid dynamics (CFD) simulations using graphics processing units (GPUs), a promising emerging hardware and software platform whose computational power, already counted in teraflops, place it among supercomputers of only five years ago. To this end, we have implemented two GPU-accelerated iterative solvers for large systems of sparse linear equations: the Preconditioned Conjugate Gradient (CG) method and the Preconditioned BiConjugate Gradient Stabilized (BiCGStab) method. The first of them can be used for computational problems with a symmetric, positive definite coefficient matrix, while the second one can be used for problems with arbitrary non-singular matrices. Together, they cover a large set of problems appearing in Computational Fluid Dynamics and other numerical problems.

We have tested these solvers on matrices from the Florida Matrix Collection. The SmVP performance tests yielded a GPU acceleration factor that was between 3.5 and 14.5 for single precision and between 4.5 and 15.9 for double precision, as compared to the same algorithm using the Intel MKL library and run on a modern 4-core CPU. These results show that GPUs are a very attractive supplement to traditional, CPU-based computational environments.

To test the performance of the GPU-accelerated linear solvers and their applicability to more demanding problems, we have compiled them into a stand-alone library (SpeedIT), integrated that with the OpenFOAM library, and then used it to solve a realistic 3D flow in the bifurcation of the abdominal aorta. We investigated different types of flow models as well as different methods of solution. Our investigations revealed that acceleration is limited by the amount of data transfers between the CPU and the GPU, relative to the amount of computations done in the GPU. The acceleration was quite good for relatively simple CFD methods. For example, potential flow calculations were approximately four to six times faster, depending on the preconditioner used in the CPU implementation. This was possible because all calculations were done on the GPU. We also found satisfactory acceleration for steady incompressible flows. In this case, the Navier–Stokes equations were solved using the steady SIMPLE method. Despite the computational algorithm of the SIMPLE method being much more complicated than the one for potential flows, we were still able to gain acceleration, of nearly four times, for the calculations.

Depending on the problem being considered, both potential flow or steady flow solvers may be sufficient. In these cases, it is possible to accelerate linear solvers only, and still achieve satisfactory acceleration. However, for more de-

manding problems, such as in transient flow solvers, this simple strategy is no longer sufficient, since when we linked our GPU-accelerated linear solvers with the transient PISO method, the acceleration was hardly noticeable. This leads to the conclusion that in order to obtain reasonably good acceleration for time-dependent problems, the whole computational algorithm must be ported to the GPU. The first step towards reaching this goal could be the generalization of our solvers for handling systems of linear equations of the form $\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}$, i.e. with the coefficient matrix $\mathbf{A}$, depending on the solution, which is our next goal.

Another aim of our work was the investigation of the flow in the human abdominal aorta, based on real three-dimensional geometry. By using the PISO method for transient incompressible flows, we were able to observe several interesting phenomena. We observed that the flow pattern had been strongly affected by the state of the inlet velocity. When the inlet velocity was at an increasing phase, the flow had gradually become more ordered and finally reached a state that was similar to the one obtained with the steady SIMPLE solver. When the inlet velocity was at the decreasing phase, the flow had become more complex and chaotic. Shortly after passing the maximum inlet velocity value, a relatively large recirculation zone emerged. With decreasing velocity, we were able to observe several different vortex structures. This indicated a strong mixing nature for the flow. Our next step in more realistic blood flow simulations will be to consider the blood as a non-Newtonian fluid, and to model the interaction between the blood and the vein's wall.

References

1. V. Kurtcuoglu *et al.*, *Computational investigation of subject-specific cerebrospinal fluid flow in the third ventricle and aqueduct of sylvius*, Journal of Biomechanics, **40**, 6, 1235–1245, 2007.

2. T. Gohil *et al.*, *Simulation of Oscillatory Flow in an Aortic Bifurcation Using FVM and FEM: a Comparative Study of Implementation Strategies*, International Journal for Numerical Methods in Fluids, 2010.

3. S. Hirsch *et al.*, *A mechano-chemical model of a solid tumor for therapy outcome predictions* [in:] Computational Science – ICCS 2009, Lecture Notes in Computer Science, **5544/2009**, 715–724, Springer Berlin/Heidelberg, 2009.

4. D. Szczerba *et al.*, *Mechanism and localization of wall failure during abdominal aortic aneurysm formation* [in:] ISBMS '08: Proceedings of the 4th international symposium on Biomedical Simulation, 119–126, Berlin, Heidelberg: Springer-Verlag, 2008.

5. D. Szczerba, G. Szekely, *Simulating Vascular Systems in Arbitrary Anatomies*, [in:] Medical Image Computing and Computer-Assisted Intervention, Springer, 2005.

6. *OpenFOAM. The Open Source CFD Toolbox. User Guide*, Free Software Foundation, Inc., 2009.

7. *OpenFOAM. The Open Source CFD Toolbox. Programmer's Guide*, Free Software Foundation, Inc., 2009.

8. L. MIROSŁAW *et al.*, *Integration of GPU-Accelerated Linear Solvers with OpenFoam and Their Application to CFD*, submitted to The Computer Journal, 2010.

9. H. OERTEL, ED., *Prandtl's Essentials of Fluid Mechanics*, Springer-Verlag, 2000.

10. R. ISSA, *Solution of implicitly discretized fluid flow equations by operator-splitting*, J. Comput. Phys. **62**, 40–65, 1986.

11. T. CHUNG, *Computational fluid dynamics*, Cambridge University Press, 2002.

12. J. FERZIGER, M. PERIĆ, *Computational methods for fluid dynamics*, Springer Berlin, 1999.

13. H. VERSTEEG, W. MALALSEKERA, *An introduction to computational fluid dynamics*, Longman Scientific & Technical, 1995.

14. *NVIDIA CUDA Programming Guide Version 3.0*, NVIDIA, 2010. [on-line:] http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.

15. *ATI Stream Technology*, http://www.amd.com/us/products/technologies/stream-technology/Pages/stream-technology.aspx, 2009.

16. J. TÖLKE, M. KRAFCZYK, *TeraFLOP computing on a desktop PC with GPUs for 3D CFD*, International Journal of Computational Fluid Dynamics, **22**, 7, 443–456, 2008.

17. E. ELSEN, P. LEGRESLEY, E. DARVE, *Large calculation of the flow over a hypersonic vehicle using a GPU*, J. Comput. Phys., **227**, 24, 10 148–10 161, 2008.

18. J.M. COHEN, M.J. MOLEMAKER, *A fast double precision CFD code using CUDA*, [in:] 21st International Conference on Parallel Computational Fluid Dynamics (ParCFD2009), 2009.

19. J.C. THIBAULT, I. SENOCAK, *CUDA Implementation of a Navier–Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows*, [in:] 47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition, American Institute of Aeronautics and Astronautics, Inc., 2009.

20. I. SENOCAK, J. THIBAULT, M. CAYLOR, *Rapid-Response Urban CFD Simulations Using a GPU Computing Paradigm on Desktop Supercomputers*, [in:] Eighth Symposium on the Urban Environment, 2009.

21. D.A. JACOBSEN, J.C. THIBAULT, I. SENOCAK, *An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters*, [in:] 48th AIAA Aerospace Sciences Meeting, American Institute of Aeronautics and Astronautics, Inc., 2010.

22. D. GÖDDEKE *et al.*, *GPU acceleration of an unmodified parallel finite element Navier-Stokes solver*, [in:] W.W. SMARI, J.P. MCINTIRE [Eds.], High Performance Computing & Simulation 2009, 12–21, 2009.

23. C. FLETCHER, *Computational Techniques for Fluid Dynamics 2*, Springer-Verlag, 1991.

24. K. HOFFMANN, S. CHIANG, *Computational Fluid dynamics, Vol. II*, Engineering Education System, 2000.

25. Y. SAAD, *Iterative Methods for Sparse Linear Systems*, SIAM, 2003.

26. R. BARRETT *et al.*, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, Philadelphia, PA: SIAM, 1994.

27. N. BELL, M. GARLAND, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, [in:] SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 1–11, New York, NY, USA, ACM, 2009.

28. T.A. DAVIS, Y. HU, *The University of Florida Sparse Matrix Collection*, submitted to ACM Transactions on Mathematical Software.

29. A. CHRIST *et al.*, *The virtual family-development of surface-based anatomical models of two adults and two children for dosimetric simulations*, Physics in Medicine and Biology, **55**, 2, N23, 2010.

Received June 14, 2010; revised version March 17, 2011.