7 GPU-Based High Performance Computing: Employing Massively Parallel Processors for Speeding-Up Compute Intensive Algorithms

Constantin Suciu, Lucian Itu, Cosmin Nita, Anamaria Vizitiu, Iulian Stroia, Laszlo Lazăr, Alina Gîrbea, Ulrich Foerster, and Viorel Mihalef

Abstract

One-dimensional blood flow models have been used extensively for hemodynamic computations in the human arterial circulation. In this chapter we introduce a high performance computing solution based Graphics Processing Units (GPU). Novel GPU only and hybrid CPU-GPU solutions are proposed and evaluated. Physiologically sound periodic (structured tree) and non-periodic (windkessel) boundary conditions are considered, in combination with both elastic and viscoelastic arterial wall laws, and different second-order accurate numerical solutions schemes. Both the GPU only and the hybrid solutions lead to significantly smaller execution times.

Parts of Sect. 7.2 have been published before in the paper 'Graphics Processing Unit Accelerated One-Dimensional Blood Flow Computation in the Human Arterial Tree', International Journal on Numerical Methods in Biomedical Engineering, Vol. 29, December, 2013, pp. 1428–1455.

C. Suciu (✉) • L. Itu • C. Nita • A. Vizitiu • I. Stroia • A. Gîrbea
Corporate Technology, Siemens SRL, B-dul Eroilor nr. 3A, Brasov 500007, Romania

Automation and Information Technology, Transilvania University of Brasov, Mihai Viteazu nr. 5, Brasov 5000174, Romania
e-mail: constantin.suciu@siemens.com

L. Lazăr
Corporate Technology, Siemens SRL, B-dul Eroilor nr. 3A, Brasov 500007, Romania

U. Foerster
Corporate Technology, Siemens AG, Otto-Hahn-Ring 6, Munich 81739, Germany

Harz University of Applied Sciences, Friedrichstrasse 57-59, Wernigerode 38855, Germany

V. Mihalef
Medical Imaging Technologies, Siemens Healthcare, 755 College Road, Princeton, NJ 08540, USA

L.M. Itu et al. (eds.), *Patient-specific Hemodynamic Computations: Application to Personalized Diagnosis of Cardiovascular Pathologies*,
DOI 10.1007/978-3-319-56853-9_7

Moreover, we introduce GPU-based computationally efficient implementations for the voxelization of a mesh, for the solution of large linear system of equations using both the preconditioned conjugate gradient method, and for random forest based classification. The GPU based implementations can lead to significant execution time improvements for all these applications. This feature is based on research, and is not commercially available. Due to regulatory reasons its future availability cannot be guaranteed.

7.1 Introduction

Graphics Processing Units (GPUs) are dedicated processors, designed originally as graphic accelerators. Since CUDA (Compute Unified Device Architecture) was introduced in 2006 by NVIDIA as a graphic application programming interface (API), the GPU has been used increasingly in various areas of scientific computations due to its superior parallel performance and energy efficiency, leading to the definition of a new concept: general-purpose computing on graphics processing units (GPGPU) (Ryoo et al. 2008; Zou et al. 2009). GPGPU pipelines initially developed due to the speed-up requirements of scientific computing applications. Due to its very efficient performance-cost ratio, and its widespread availability, the GPU is currently the most used massively parallel processor.

Multiple graphics cards may be used in one computer, or GPU clusters may be employed. Nevertheless even a single CPU-GPU framework typically outperforms multiple CPUs due to the specialization of each processor (Mittal and Vetter 2015).

The GPU is a stream processor, developed for performing a large number of floating point operations (FLOPS) in parallel, by using a large number of processing units. Recent GPUs deliver the same performance as that of a cluster at 10% of the cost. The most important development has been the introduction of a programming interface for GPUs, which has transformed them into General Purpose GPUs (GPGPU) (Owens et al. 2008). The possibility of running jobs, which typically require clusters, on a single computer equipped with one or more GPUs, has drawn the attention of researchers from various research areas.

Since graphics applications require significant computing power, the GPU, as opposed to the CPU, uses the majority of its transistors for data processing and not for data cache or for execution control. This characteristic coincides with the requirements of many scientific computing applications (Chen et al. 2009). Furthermore, the GPU is well suited for parallel computations. Its stream processors work independently and concurrently at high frequencies when a program is being executed, diminishing thus significantly the need for complex execution control.

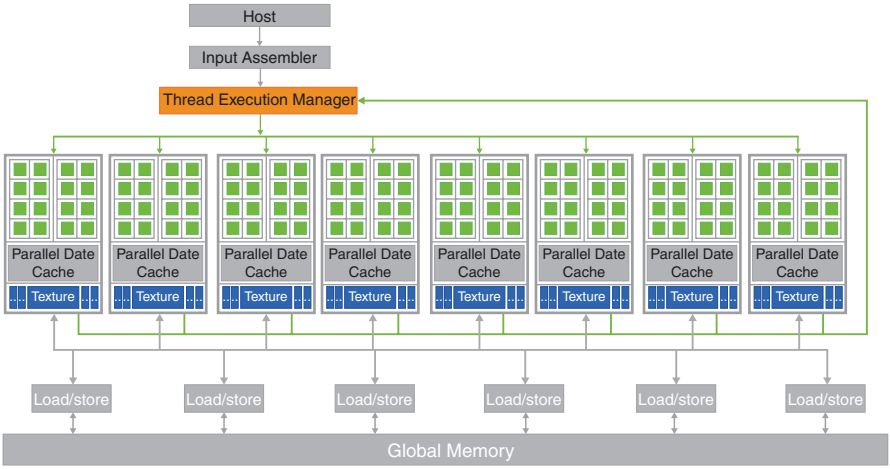

Fig. 7.1 Architecture of a GPU programmed through CUDA (Kirk and Hwu 2010)

Figure 7.1 depicts the architecture of a GPU which can be programmed through CUDA. It is composed of several streaming multiprocessors, which can execute in parallel a large number of threads. In this example, two multiprocessors form a structural block (this specific organization varies from one GPU generation to the next). Furthermore, each multiprocessor has a number of processing units which have a common control logic and instruction cache.

The G80 GPU, which was introduced together with the CUDA language, had a memory bandwidth of 86 GB/s a CPU-GPU bandwidth of 8 GB/s. Furthermore the G80 has 128 processing units (16 multiprocessors, each one with 8 processing units). The peak performance was over 500 gigaflops. Current GPUs have over 1000 processing units and exceed several teraflops in terms of processing performance.

Each processing unit can run thousands of threads in a single application. A typical GPU-based application executes 5000–15,000 threads. The G80 allows the simultaneous execution of up to 768 threads per streaming multiprocessor, and over 12,000 threads for the entire GPU. The GT200 allows the simultaneous execution of up to 1024 threads per streaming multiprocessor, and over 30,000 threads for the entire GPU. The maximum level of parallelism at hardware level has increased with the release of the Fermi, Kepler and Maxwell architectures.

In the CUDA architecture, the host and the CUDA device have separate memory spaces: CUDA devices are separate hardware components with their own DRAM. To run a kernel on the device, the programmer needs to allocate memory on the device and transfer the data of interest from the host to the device. Similarly, once the execution on the GPU has finalized, the results need to be transferred back to the CPU, and the device memory that is no longer required has to be deallocated.

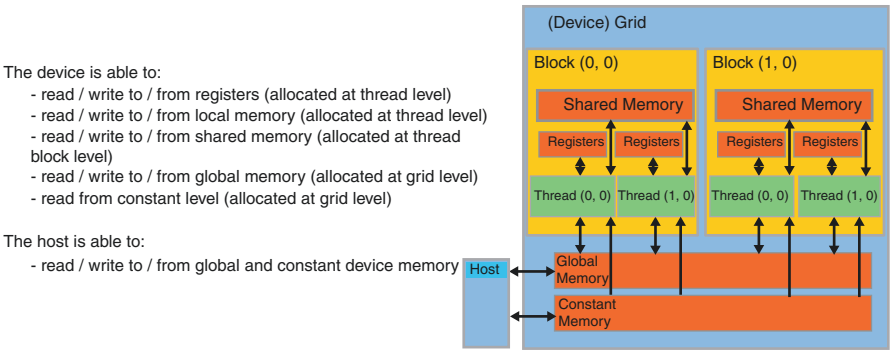

Fig. 7.2 CUDA memory model (Kirk and Hwu 2010)

Figure 7.2 depicts an overview of the CUDA memory model. On the lower side of the figure one can observe the global and the constant memory. These are the two types of memory that can be accessed by the host in both directions (the constant memory can only be read by the CUDA device).

At thread block level, shared memory is available, which can be accessed by all threads of the same block. Typically, shared memory is employed to reduce the number of redundant accesses of the global memory. Unlike the global memory, shared memory can be accessed with low latency for reading and writing data.

Furthermore, each thread has access to a set of localized registers. These are high speed memory locations, but the number of such locations is limited.

GPUs are currently extensively being used in biomedical engineering (for fast diagnosis, simulation of surgical procedures and prediction, etc.). Image-guided therapy systems, which are gaining more traction in clinical treatment and interventions applications, are based on programs with a large level of parallelism (Shams et al. 2010). Thus, radiology departments may profit in future from GPU-based high performance computing, at reasonable costs. GPUs have been used in research activities related to the cardiovascular system: modelling and simulation of the heart (offline—to reduce the simulation time), or for the real-time visualization of 3D/4D sequences acquired through MRI or CT. Sato et al. have described the acceleration of simulations performed for understanding the propagation of electrical waves in the cardiac tissue, and have compared CPU and GPU based execution times (Sato et al. 2009). The approach based on a single GPU has shown to be 20–40 times than a single CPU based implementation. Furthermore, an interactive simulation for cardiac interventions has been proposed, which also takes into account the collision between the catheter and the inner wall of the heart (Yu et al. 2010). The GPU-based implementation is used for cardiac modelling, visualization and interactive simulation. The simulation of the cardiovascular system can also be performed with the Lattice Boltzmann method which is readily parallelizable on GPUs (Nita et al. 2013).

Tanno et al. have introduced a GPU-based implementation of the artificial compressibility method (Tanno et al. 2011). A speed-up of approx. 8× was achieved, when compared to the CPU based implementation.

Moreover, a multi-GPU simulator based on the three-dimensional Navier-Stokes equations has been introduced, which can simulate the interaction between different fluids, and which is based on a level-set approach (Zaspel and Griebel 2013). High-order finite differences schemes and the projection method were employed for the discretization in space and time. The speed-up, when compared to a CPU-based implementation, is of around 3×, while the energy consumption is approx. two times smaller.

Another research study has analyzed the GPU-based speed-up potential for a hemodynamic simulation performed for the abdominal aorta (Malecha et al. 2011). Only a part of the code has been transferred to the GPU (the solution of the linear system of equations), and a speed-up of 3–4× was obtained.

In a different approach, blood was modeled as a mix of plasma and red blood cells, and hemodynamic simulations were run (Rahimian et al. 2010). Up to 260 millions deformable red blood cells were used, and the code was developed so as to support parallelism at all levels, including internodal parallelism, intranodal parallelism through shared memory, data parallelism (vectorization), as well as a very large number of threads for efficient GPU-based execution. The most complex simulation employed 256 CPU-GPU systems, and 0.7 petaflops were achieved.

In conclusion, GPU-based approaches are already employed widely in many compute-intensive research areas, like:

- Computational fluid dynamics (Itu et al. 2013a, b)
- Machine learning (Garcia et al. 2008)
- Molecular modeling (Hasan Khondker et al. 2014)
- Astrophysics (Klages et al. 2015)
- Bioinformatics (Schatz et al. 2007; Manavski and Valle 2008)
- Medical imaging (Shams et al. 2010; Shen et al. 2009)
- Lattice Boltzmann methods (Nita et al. 2013)
- Monte Carlo simulations (Alerstam et al. 2008)
- Weather forecasting (Schalkwijk et al. 2015)
- Fuzzy logic (Cococcioni et al. 2011)

The first generation of CUDA enabled GPUs was limited to single precision floating point computations, and hence could not be used efficiently for applications with high accuracy requirements, e.g. CFD. Double precision computations were then made available in the second generation of CUDA enabled GPUs (Tesla architecture). Furthermore, the Fermi and Kepler architectures increased the double precision performance. Hence, GPUs are used nowadays widely for the acceleration of scientific computations.

The GPU is viewed as a compute device which is able to run a very high number of threads in parallel inside a kernel (a function, written in C language, which is executed on the GPU and launched by the CPU). The threads of a kernel are organized at three levels: blocks of threads are organized in a three dimensional grid at the top level, threads are organized in three-dimensional blocks at the middle level, and, at the lowest levels, threads are grouped into warps (groups of 32 threads formed by linearization of the three-dimensional block structure along the x, y and z axes respectively).

The concepts and information presented in this chapter are based on research and are not commercially available. Due to regulatory reasons its future availability cannot be guaranteed.

7.2 GPU Accelerated One-Dimensional Blood Flow Computation in the Human Arterial Tree

In previous chapters we have focused on computational approaches for modeling the flow of blood in the human cardiovascular system.

For three-dimensional blood flow models, due to the extremely high computational requirements, there has been a lot of interest in exploring high performance computing techniques for speeding up the algorithms. Although one-dimensional blood flow models are generally at least two orders of magnitude faster, the requirement of short execution times is still valid. Thus, when blood flow is modeled in patient-specific geometries in a clinical setting, results are required in a timely manner not only to potentially treat the patient faster, but also to perform computations for more patients in a certain amount of time.

As described in previous chapter, it is crucial to match the patient-specific state in a hemodynamic computation. The tuning procedure requires repetitive runs on the same geometry, with different parameter values (e.g. for inlet, outlet or wall boundary conditions (BC)), until the computed and the measured quantities match. This increases the total execution time for a single patient-specific geometry.

Another type of application for which the acceleration of the hemodynamic computations in general, and of the one-dimensional blood flow in particular, is important, has been reported in (Vardoulis et al. 2012): a total of 1000 different hemodynamic cases have been simulated (by varying geometry, heart rate, compliance or resistance) in order to find a correlation between total arterial compliance and aortic pulse wave speed. The speed-up of the execution time can either significantly reduce the time required to perform all computations for a certain correlation result, or it can allow one to perform even more computations in the same amount of time, with different configurations, to obtain better final results.

The high performance computing techniques have focused on the parallelization of the algorithms. For the implementation of the parallel algorithms cluster-based approaches, graphics processing unit (GPU) based approaches, or even a combination

of the two have been explored. Since the execution time of a three-dimensional model is generally at least two orders of magnitude higher than the execution time of a one-dimensional model, most of the parallelization activities have focused on three-dimensional models (Habchia et al. 2003; Tanno et al. 2011; Zaspel and Griebel 2013). Nevertheless, the speed-up requirements mentioned above are equally valid for one-dimensional and three-dimensional models.

A previous research activity focused on the parallelization of the one-dimensional blood flow model, employing a cluster-based approach, has been introduced in (Kumar et al. 2003), with speed-up values of up to 3.5×. On the other hand, with the advent of CUDA (Compute Unified Device Architecture), several researchers have identified the potential of GPUs to accelerate biomedical engineering applications in general (Kirk and Hwu 2010), and computational fluid dynamics (CFD) computations in particular to unprecedented levels (Jiang et al. 2011).

In this section, we focus on the GPU based acceleration of the one-dimensional blood flow model and present two algorithms: a novel Parallel Hybrid CPU-GPU algorithm with Compact Copy operations (PHCGCC) and a Parallel GPU Only (PGO) algorithm (Itu et al. 2013a, b). We use a full body arterial model composed of 51 arteries and the speed-up of the two approaches is evaluated compared to both single-threaded and multi-threaded CPU implementations. The computations are performed using two different second order numerical schemes, with an elastic or viscoelastic wall model, and windkessel or structured tree boundary conditions as representative examples of physiological non-periodic and respectively periodic outlet boundary conditions.

7.2.1 Methods

The one-dimensional blood flow model is derived from the three-dimensional Navier-Stokes equations based on a series of simplifying assumptions (Formaggia et al. 2003). The governing equations ensuring mass and momentum conservation are as follows:

$$\frac{\partial A(x,t)}{\partial t} + \frac{\partial q(x,t)}{\partial x} = 0, \tag{7.1}$$

$$\frac{\partial q(x,t)}{\partial t} + \frac{\partial}{\partial x}\left(\alpha \frac{q^2(x,t)}{A(x,t)}\right) + \frac{A(x,t)}{\rho}\frac{\partial p(x,t)}{\partial x} = K_R \frac{q(x,t)}{A(x,t)}, \tag{7.2}$$

where x denotes the axial location and t denotes the time. $A(x,t)$ is the cross-sectional area, $p(x,t)$ the pressure, $q(x,t)$ the flow rate, and ρ is the density. Coefficients α and K_R account for the momentum-flux correction and viscous losses due to friction respectively. For a parabolic velocity profile, $K_R = -8\pi\nu$ and $\alpha = 4/3$, with ν being the kinematic viscosity of the fluid.

A state equation, which relates the pressure inside the vessel to the cross-sectional area, is used to close the system of equations. When the vessel wall is modeled as a pure elastic material, the following relationship holds:

$$p(x,t) = \Psi_{el}(A) + p_0 = \frac{4}{3}\frac{Eh}{r_0}(x)\left(1 - \sqrt{\frac{A_0(x)}{A(x,t)}}\right) + p_0, \tag{7.3}$$

where E is the Young modulus, h is the wall thickness, r_0 is the initial radius corresponding to the initial pressure p_0, and A_0 is the initial cross-sectional area. The elastic wall properties are estimated using a best fit to experimental data (Olufsen et al. 2000).

Alternatively, a viscoelastic wall model can also be used. To include viscoelasticity, the vessel wall is considered to be a Voigt-type material (Fung 1993), for which the tensile stress depends on both the tensile strain and the time-derivative of the strain (Malossi et al. 2012):

$$p(x,t) = \Psi_{el}(A) + \Psi_{v}(A) + p_0 = \frac{4}{3}\frac{Eh}{r_0}\left(1 - \sqrt{\frac{A_0}{A(x,t)}}\right) + \frac{\gamma_S}{A\sqrt{A}}\frac{\partial A}{\partial t} + p_0, \tag{7.4}$$

where γ_S is the viscoelastic coefficient, defined by:

$$\gamma_S = \frac{T_S \cdot \tan\Phi_S}{4\pi}\frac{hE}{1-\sigma^2}. \tag{7.5}$$

Here T_S is the wave characteristic time (usually taken equal to the systolic period ~0.24 s), Φ_S is the viscoelastic angle (10°), while σ is the Poisson ratio (the material is considered to be incompressible for σ = 0.5). As in the case of elastic modeling, the viscoelastic coefficient is considered to be non-uniform in space (i.e. $\gamma_S = \gamma_S(x)$).

The presence of the viscoelastic component in Eq. (7.4) introduces an additional term in the momentum conservation equation:

$$\frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\alpha\frac{q^2}{A}\right) + \frac{A}{\rho}\frac{\partial \Psi_{el}}{\partial x} + \frac{A}{\rho}\frac{\partial \Psi_{v}}{\partial x} = K_R\frac{q}{A}. \tag{7.6}$$

Spatial and temporal dependencies of the quantities have been omitted for notational clarity. At each bifurcation, the continuity of flow and total pressure is imposed,

$$q_p = \sum_i (q_d)_i, \tag{7.7}$$

$$p_p + \frac{1}{2}\rho\frac{q_p^2}{A_p^2} = (p_d)_i + \frac{1}{2}\rho\frac{(q_d^2)_i}{(A_d^2)_i}, \tag{7.8}$$

where subscript p refers to the parent, while subscript d refers to the daughter vessels.

7.2.1.1 Boundary Conditions

Depending on the availability of in-vivo measurements and the underlying assumptions used in the modeling, researchers typically use one of the following inlet boundary condition: (1) time-varying flow profile, (2) a lumped model of the heart coupled at the inlet (Formaggia et al. 2006), or (3) a non-reflecting boundary condition like a forward running pressure wave (Mynard et al. 2012a, b), (Willemet et al. 2011). A time-varying velocity profile (or flow rate profile) can be consistently determined in a clinical setting, and is often part of the diagnostic workflow (2D/3D Phase-contrast MRI (Magnetic Resonance Imaging), Doppler ultrasound). The parameters of the lumped heart model can be computed based on non-invasively acquired flow rate and pressure values (Senzaki et al. 1996), while the third type of inlet boundary condition is generally not used in patient-specific computations.

Outlet boundary conditions may be classified as either periodic or non-periodic boundary conditions. Whereas periodic boundary conditions can only be used in steady-state computations (e.g. the patient state does not change from one heart cycle to the next—the same inlet flow rate profile is applied for each heart cycle) and require the flow information from the previous heart cycle, non-periodic boundary conditions do not have these restrictions (e.g. they can be used to model the transition from a rest state to an exercise state for a patient).

We consider two physiologically motivated boundary conditions:

1. The three-element windkessel model (WK), as a non-periodic boundary condition (Westerhof et al. 1971):

$$\frac{\partial p}{\partial t} = R_p \frac{\partial q}{\partial t} - \frac{p}{R_d \cdot C} + \frac{q\left(R_p + R_d\right)}{R_d \cdot C}, \tag{7.9}$$

where R_p is the proximal resistance, R_d is the distal resistance and C is the compliance, while p and q refer to the pressure and respectively the flow rate at the inlet of the windkessel model.

2. The structured tree model (ST) (Olufsen et al. 2000), as a periodic boundary condition. The structured tree is a binary, asymmetrical vascular tree computed individually for each outlet, composed of a varying number of vessel generations. It is terminated once the radius decreases below a preset minimum radius and its root impedance, $z(t)$, is computed recursively. The root impedance is applied at the outlet of the proximal domain through a convolution integral:

$$p\left(x,t\right) = \int_{t-T}^{t} q\left(x,\tau\right) z\left(x,t-\tau\right) d\tau, \tag{7.10}$$

where T is the period. To apply a periodic boundary condition, the flow history is stored and a multiply-sum scan operation is performed at each time-step, leading to considerably higher execution times than for a non-periodic boundary condition.

We emphasize the fact that this choice is not mandatory, the windkessel model can also be applied as a periodic boundary condition, and, as recently described, even the structured tree boundary condition can be applied as a non-periodic boundary condition (Cousins et al. 2013).

7.2.1.2 Numerical Solution of the One-Dimensional Blood Flow Model

When an elastic wall model is used, Eqs. (7.1)–(7.3) represents a hyperbolic system of equations. When a viscoelastic wall model is used, the hyperbolic nature of the equations is lost due to the additional term in the pressure-area relationship. The approaches for the numerical solution of the one-dimensional equations can be divided into two main categories:

1. Methods which do not exploit the original hyperbolic nature of the equations: discontinuous finite element Galerkin method with stabilization terms (Raghu et al. 2011); implicit finite difference/spectral element method where the non-linear terms are solved iteratively at each time-step using the Newton method (Reymond et al. 2011), (Bessems et al. 2008), etc.;
2. Methods which exploit the hyperbolic nature of the equations in case an elastic wall law is used (Olufsen et al. 2000), (Mynard and Nithiarasu 2008), or which recover the original hyperbolic nature of the equations in case a viscoelastic wall law is used, by employing an operator-splitting scheme for the momentum equation. This method has been originally proposed in Formaggia et al. (2003) for a single vessel and subsequently used in Passerini (2009) and Alastruey et al. (2011).

Implicit methods, although solvable with larger time-steps, are slower since they require the solution of a system of equations at each time step and, additionally, require the application of the Newton method for the non-linear terms (Reymond et al. 2011). On the other hand, the methods that exploit the hyperbolic nature of the equations are explicit. They are computationally faster, in spite of the time-step limitation imposed by the CFL condition (named after Courant, Friedrich and Lewy (Courant et al. 1928)).

In addition to the fact that the explicit methods are faster in a sequential implementation, their explicit nature also enables them to be parallelized, and thus be implemented on a cluster (Kumar et al. 2003), or on a GPU.

The explicit methods are based either on a first order method (the method of characteristics), or on a second order method (two-step Lax Wendroff (Olufsen et al. 2000), or expansion in Taylor series (Mynard and Nithiarasu 2008)). Due to their higher accuracy, second-order methods are preferred and have been used for the current study. The method of characteristics is used at the inflow, bifurcation and outflow points.

7.2.1.3 Numerical Solution of the Elastic One-Dimensional Model

First, Eqs. (7.1) and (7.2) are written in conservation form:

$$\frac{\partial \mathbf{U}}{\partial t}+\frac{\partial \mathbf{R}}{\partial x}=\mathbf{S}, \tag{7.11}$$

$$\mathbf{U}=\begin{pmatrix} A \\ q \end{pmatrix} \quad \mathbf{R}=\begin{pmatrix} R_1 \\ R_2 \end{pmatrix}=\begin{pmatrix} q \\ \alpha \dfrac{q^2}{A}+B \end{pmatrix} \quad \mathbf{S}=\begin{pmatrix} S_1 \\ S_2 \end{pmatrix}=\begin{pmatrix} 0 \\ -K_R \dfrac{q}{A}+\dfrac{\partial B}{\partial r_0}\dfrac{dr_0}{dx} \end{pmatrix}, \tag{7.12}$$

$$B\left(r_0(x),p(x,t)\right)=\frac{1}{\rho}\int_{p_0}^{p(x,t)}\Psi_{el}^{-1}\left(p'\right)dp',$$

where $\boldsymbol{U}$ is the vector of the unknown quantities, $\boldsymbol{R}$ is the flux term, and $\boldsymbol{S}$ is the right hand side (RHS).

The Lax-Wendroff (LW) scheme consists of two main steps:

Step 1. Computation of the half step-points: these values are computed between the grid points, hence there are only interior half-step values:

$$\mathbf{U}_j^{n+1/2}=\frac{\mathbf{U}_{j+1/2}^n+\mathbf{U}_{j-1/2}^n}{2}+\frac{\Delta t}{2}\cdot\left(-\frac{\mathbf{R}_{j+1/2}^n-\mathbf{R}_{j-1/2}^n}{\Delta x}+\frac{\mathbf{S}_{j+1/2}^n+\mathbf{S}_{j-1/2}^n}{2}\right), \tag{7.13}$$

where $j=m\pm 1/2$ and m refers to the grid points;

Step 2. Computation of the full-step-points: this step uses values both from the previous time step and from the half-step points:

$$\mathbf{U}_m^{n+1}=\mathbf{U}_m^n-\frac{\Delta t}{\Delta x}\left(\mathbf{R}_{m+1/2}^{n+1/2}-\mathbf{R}_{m-1/2}^{n+1/2}\right)+\frac{\Delta t}{2}\cdot\left(\mathbf{S}_{m+1/2}^{n+1/2}+\mathbf{S}_{m-1/2}^{n+1/2}\right)\cdot\left(\mathbf{S}_{m+1/2}^{n+1/2}+\mathbf{S}_{m-1/2}^{n+1/2}\right). \tag{7.14}$$

The expansion in Taylor series (TS) scheme consists of a single step:

$$\frac{\mathbf{U}^{n+1}-\mathbf{U}^n}{\Delta t}=\mathbf{S}^n-\frac{\partial \mathbf{R}^n}{\partial x}-\frac{\Delta t}{2}\left[\frac{\partial}{\partial x}\left(\mathbf{R}_{\mathbf{U}}^n\mathbf{S}^n-\mathbf{R}_{\mathbf{U}}^n\frac{\partial \mathbf{R}^n}{\partial x}\right)-\mathbf{S}_{\mathbf{U}}^n\frac{\partial \mathbf{R}^n}{\partial x}-\mathbf{S}_{\mathbf{U}}^n\mathbf{S}^n\right], \tag{7.15}$$

where all the spatial derivatives are discretized using central difference schemes, and:

$$\mathbf{R}_{\mathbf{U}}^n=\frac{\partial \mathbf{R}}{\partial \mathbf{U}};\quad \mathbf{S}_{\mathbf{U}}^n=\frac{\partial \mathbf{S}}{\partial \mathbf{U}}. \tag{7.16}$$

Both numerical schemes require the apriori computation of the flux and RHS terms.

7.2.1.4 Numerical Solution of the Viscoelastic One-Dimensional Model

An operator splitting scheme is employed for the momentum equation in order to recover the hyperbolic nature of the equations. Thus, Eq. (7.6) is rewritten as:

$$\frac{\partial q}{\partial t}+\frac{\partial R_2}{\partial x}-\frac{A}{\rho}\frac{\partial}{\partial x}\left(\gamma\frac{\partial q}{\partial x}\right)=S_2. \tag{7.17}$$

The equation is no longer hyperbolic and cannot be cast into conservative form. The splitting scheme assumes that the contribution of the viscoelastic term is small compared to the contribution of the elastic term. The flow rate is considered to be composed of an elastic and a viscoelastic component ($q=q_e+q_v$), and Eq. (7.17) is split into two equations:

$$\frac{\partial q_e}{\partial t}+\frac{\partial R_2}{\partial x}=S_2. \tag{7.18}$$

$$\frac{\partial q_v}{\partial t}-\frac{A}{\rho}\frac{\partial}{\partial x}\left(\gamma\frac{\partial q}{\partial x}\right)=0. \tag{7.19}$$

Consequently, the numerical solution at each step is composed of two sequential sub-steps:

Step 1. The system composed of Eqs. (7.1) and (7.18) is solved, yielding the quantities $A(x,t)$ and $q_e(x,t)$.
Step 2. Equation (7.19) is solved to obtain $qv(x,t)$ and thus the total flow rate $q(x,t)$:

$$\frac{q_v^{n+1}-q_v^n}{\Delta t}-\frac{A^{n+1}}{\rho}\frac{\partial}{\partial x}\left(\gamma\frac{\partial\left(q_e^{n+1}+q_v^{n+1}\right)}{\partial x}\right)=0. \tag{7.20}$$

Equation (7.20) is discretized using a central difference scheme, leading to a tridiagonal system of equations, which can be readily solved using the Thomas algorithm in a sequential program. For the viscoelastic component of the flow, homogeneous Dirichlet boundary conditions are imposed at the boundaries of each vessel.

Both the LW and the TS schemes can be used to compute $A(x,t)$ and $qe(x,t)$, but because the LW method is composed of two steps, it requires the computation of the viscoelastic correction term twice for each time step. Since this would significantly increase the total execution time, we applied only the TS scheme when a viscoelastic wall law was enforced. The various computational setups, for which different parallelization strategies have been adopted, are displayed in Table 7.1. For performance comparison, we considered both a Single-threaded CPU only (SCO) algorithm and a Multi-threaded CPU only (MCO) algorithm. The MCO algorithm represents a parallel version of the SCO algorithm, implemented using openMP.

Table 7.1 Computational setups for which the speed-up obtained through GPU-based parallel implementations is investigated

Case	Numerical scheme	Wall law	Outlet BC
1	Lax-Wendroff	Elastic	Windkessel
2	Lax-Wendroff	Elastic	Structured tree
3	Taylor series	Elastic	Windkessel
4	Taylor series	Elastic	Structured tree
5	Taylor series	Viscoelastic	Windkessel
6	Taylor series	Viscoelastic	Structured tree

Table 7.2 Execution time and corresponding percentage of total execution time for the computational steps of the numerical solution of the one-dimensional blood flow model

	Case 5		Case 6	
Computational step	Time (s)	Perc. of total time (%)	Time (s)	Perc. of total time (%)
Interior grid points	357.12	46.12	357.67	30.32
Inflow grid point	0.08	0.01	0.14	0.01
Bifurcation grid points	27.66	3.57	27.61	2.34
Outflow grid points	4.96	0.64	401.72	34.06
Viscoelastic comp.	367.43	47.45	367.55	31.16
Other operations	17.12	2.21	24.87	2.11

Before introducing the parallel implementation of the one-dimensional blood flow model, we first analyze the execution time of the SCO algorithm. Table 7.2 displays the execution time of the different parts of the SCO algorithm, for cases 5 and 6 from Table 7.1 (these two cases were chosen because they contain all computational steps and both types of outlet boundary conditions are considered). Execution times are obtained for the arterial tree described in Sect. 7.3 and correspond to the computation for ten heart cycles. When the WK boundary condition is used, approx. 93% of the time is spent on the computation at the interior grid points and on the viscoelastic terms of the flow rate. Since the numerical solution for the interior grid points is explicit, this part can be efficiently parallelized on a manycore architecture (like the one of a GPU device). Though the computation of the viscoelastic terms employs a sequential algorithm, it can also be efficiently parallelized on a manycore architecture, as shown in Zhang et al. (2010). Furthermore, the computation at the bifurcation and outflow grid points is also parallelizable, but due to the low number of grid points of these types, usually below 100 for an arterial tree, the implementation on a manycore architecture is not efficient. Other operations (initialization activities, writing results to files during the last heart cycle, etc.) account for 2.21% of the total execution time and are not parallelizable. As a result, operations which occupy 93.57% of the total execution time for case 5 are efficiently parallelizable. The difference in terms of execution time between case 5 and case 6 is primarily due to the outlet boundary condition, which requires a multiply-sum scan operation at each

time step. Since this operation is efficiently parallelizable on a manycore architecture (Sengupta et al. 2008), the operations which occupy 95.54% of the total execution time for case 6 are efficiently parallelizable.

For the MCO algorithm, the computation on the interior, bifurcation and outflow points, as well as the computation of the viscoelastic component of the flow rate, can be efficiently parallelized since the number of cores is much smaller for a multicore architecture than for a manycore architecture. This is achieved by associating different arterial segments and bifurcation points to distinct cores.

The results in Table 7.2 show that if the LW scheme were used for a viscoelastic wall law (case in which the viscoelastic correction term would be computed twice at each time step), the total execution would increase by 30–50%, depending on the computational setup.

We conclude that the implementation of the numerical solution of the one-dimensional blood flow model is efficiently parallelizable on a manycore architecture (like the one of a GPU device), regardless of the computational setup.

7.2.1.5 Parallelization of the Numerical Solution

We propose a parallel implementation of the one-dimensional blood flow model, based on a GPU device, programmed through CUDA (Kirk and Hwu 2010).

The numerical scheme of the interior grid points of each vessel is efficiently parallelizable and we considered two different implementation approaches:

1. A Parallel Hybrid CPU–GPU (PHCG) algorithm, whereas the unknown quantities at the interior points are computed on the GPU and the inflow/bifurcation/outflow points (called in the following junction points) are computed on the CPU. The advantage is that each device is used for computations for which it is best suited (CPU—sequential, GPU—parallel), but the disadvantage is that memory copies are required at each time step in order to interchange the values near the junction points;
2. A Parallel GPU Only (PGO) algorithm, whereas all grid points are computed on the GPU and the CPU is only used to initialize and to control the execution on the GPU. The advantage is that no memory copies between the CPU and the GPU are required, but the disadvantage is that less parallelizable operations need to be performed on the GPU.

7.2.1.6 Parallel Hybrid CPU-GPU (PHCG) Algorithm

First we refer to the implementation used in case an elastic wall law is applied. Starting from the numerical schemes described in the previous section, we applied the general workflow displayed in Fig. 7.3a.

The CPU is called host, while the GPU is called device. First, the arterial model is initialized (host memory is allocated for each grid point, initial radius, initial cross-sectional area, wall elasticity, derivatives of radius and wall elasticity are computed) and the device memory is allocated and initialized. Next, a while loop is started which advances the entire model in time for a given number of iterations and heart cycles. Inside the while loop, the host and device thread are executed in parallel until a synchronization barrier is reached. During the parallel activities, the

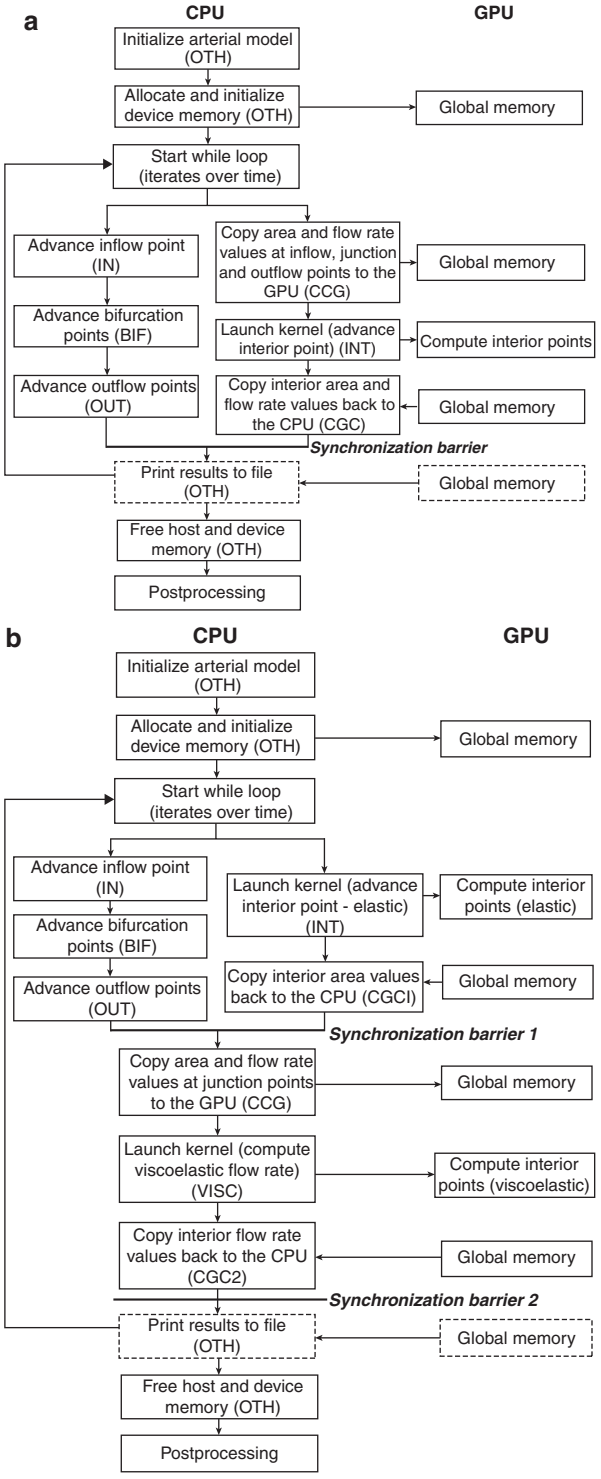

Fig. 7.3 PHCG workflow in case (**a**) an elastic wall law (Itu et al. 2012a, b), or (**b**) a viscoelastic wall law is used. Junction points are solved on the CPU, while interior points are solved on the GPU. Memory copies are required at each iteration in order to exchange the values near and at the junction points (Itu et al. 2013a, b)

CPU computes the new values at the junction points and the device performs the computations for the interior points (Eqs. (7.13), (7.14) for the LW scheme and (7.15) for the TS scheme). Since the device operations are asynchronous, no special approach is required to achieve the task level parallelism between the CPU and the GPU. The computation of the junction points on the CPU is parallelized using openMP for all PHCG implementations. An acronym is displayed in Fig. 7.3 for each operation to easily match the execution times discussed in the next section with the operations (e.g. OTH stands for *Other operations*, which comprise several activities).

To compute the junction points, the host code requires the values at the grid points next to the junction points, from the previous time step. To compute the values at the grid points next to the junction points, the device code requires the values at the junction points, also from the previous time step. Hence, to exchange the values at or next to the next junction points, memory copy operations between the device and the host are performed at the beginning and the end of each iteration. A synchronization barrier is introduced after each iteration to ensure that the copy operations have finished. During the last cycle of the computation, after convergence has been reached, the results are saved to files for visualization or post processing. Since the number of iterations for each heart cycle is very high (18,000 for a grid space of 0.1 cm), the results are saved only after a certain number of iterations (every 20–50 iterations).

To improve the execution time on the GPU, the kernel has been optimized. The specific goal has been to lower the global memory requirement. This approach is necessary to assure efficient kernel performance, even for smaller arterial trees, where parallelism is not pronounced. To reduce global memory operations, memory accesses are coalesced (the global memory accesses performed by threads of the same warp (group of 32 threads) are both sequential and aligned). To obtain aligned memory accesses all global memory arrays have been padded. Furthermore, to avoid redundant accesses performed by different threads, intermediate results are stored in the shared memory of the multiprocessor.

The execution configuration of the kernel which computes the interior grid points is organized as follows: each thread is responsible for one grid point, a block of threads is defined for each arterial segment and both the block and the thread grid are one-dimensional. The numerical solution of a one-dimensional arterial tree, as described in the previous section, is a domain decomposition approach. Hence, data is exchanged between two arterial segments only at the interfaces of the domains. Since for the PHCG algorithm the junction points are solved on the GPU and there is no communication and synchronization requirement between the thread blocks, the association between one thread block and one arterial segment is natural. Furthermore, since parallelism is limited (the number of interior grid points in an arterial tree is usually below 10,000 when a grid space of 0.1 cm is used) and the computational intensity is high (the kernel which computes the interior grid points is limited by the instruction throughput—see Sect. 7.4), an approach for which one thread may compute the unknown quantities of several grid points has not been considered. We split an arterial segment into several domains if the hardware

resources of a streaming multiprocessor were insufficient to run the corresponding thread block (the solution variables at the interfaces between the domains of the same arterial segment were determined by enforcing continuity of flow rate and total pressure).

An important aspect for the PHCG algorithm is the data transfer between host and device. Although, the amount of data to be transferred is low (only the values at or next to the junction points are exchanged), the total execution time required for these data transfers is high. This is due to the high number of copy operations and the fact that the locations to be copied are scattered throughout the memory arrays. Three different approaches, displayed in Fig. 7.4, are evaluated for decreasing the total execution time and have led to three different variants of the PHCG algorithm:

1. PHCG Copy Separately (PHCGCS): each location is copied separately, resulting in a total of eight copy operations for each arterial segment at each iteration. Figure 7.4a displays the locations for which the values are exchanged at each iteration as well as the direction of the copy operations. The arrays displayed in this figure are generic and correspond to either the cross-sectional areas or the flow rates of a single blood vessel;
2. PHCG Copy All (PHCGCA): the entire arrays used for cross-sectional area and for flow rate are transferred at each iteration: four copy operations at each iteration (Fig. 7.4b);
3. PHCG Copy Compact (PHCGCC): additional arrays are allocated for the locations which are copied: four copy operations at each iteration. Figure 7.4c displays the additional arrays which need to be allocated and the locations which read/write to these arrays. For an arterial network, the values of all dependent variables of one type (cross sectional area or flow rate) are stored in a single array.

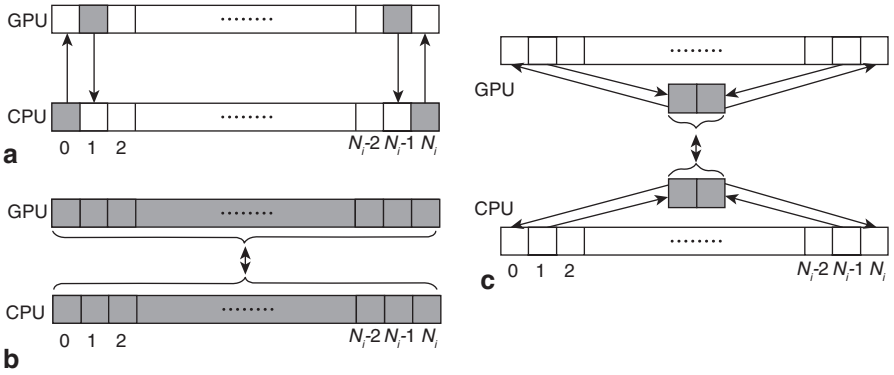

Fig. 7.4. Host ↔ Device Memory copy variants: (**a**) separate copy operation for each location, (**b**) copy entire array, (**c**) copy compact additional arrays (Itu et al. 2013a, b)

The first two memory copy strategies were introduced previously (Itu et al. 2012a, b), while the third one is developed during the current research activity and represents a combination of the first two strategies. The PHCGCS variant minimizes the amount of memory to be copied; the PHCGCA minimizes the number of copy operations, while the PHCGCC variant minimizes both aspects by trading kernel performance for data transfer performance (some threads of the kernel populate the additional arrays displayed in Fig. 7.4c).

Figure 7.5 displays the kernel operations and the shared memory arrays used for the two previously described numerical schemes (LW and TS). Since neighboring threads access the same $q/A/\boldsymbol{R}/\boldsymbol{S}$ values, shared memory is used to avoid redundant global memory reads and redundant computations. The operations of the LW scheme (Fig. 7.5a) are based on Eqs. (7.13) and (7.14) and require only four shared memory arrays (the shared memory is dynamically allocated and the size of the arrays is equal to the number of grid points of the longest vessel). The operations of the TS scheme (Fig. 7.5b) are based on Eq. (7.15) and use 11 shared memory arrays. The shared memory requirement is much higher for the TS scheme since: (1) the computations are performed in a single step (the arrays cannot be reused), and (2)

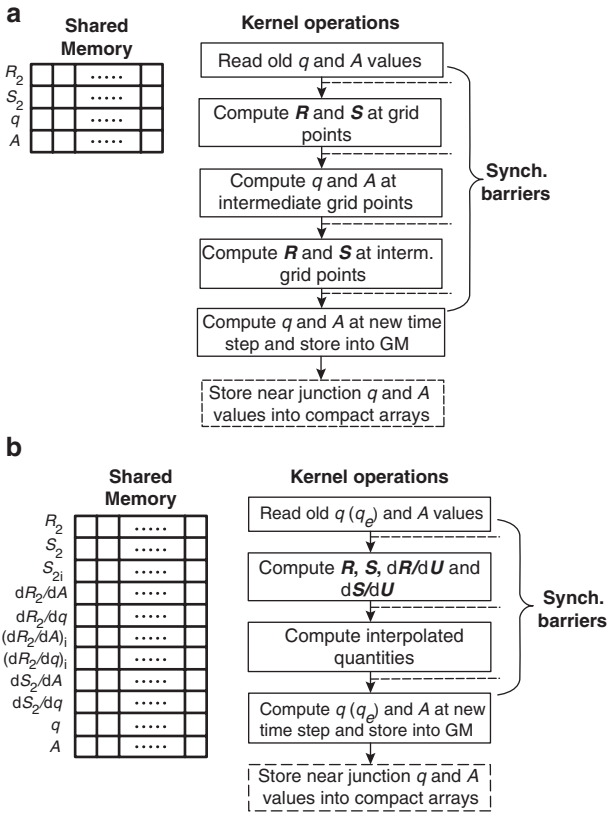

Fig. 7.5 Kernel operations and shared memory arrays used for the computation of new values at the interior grid point using (**a**) the LW scheme (Itu et al. 2012a, b), and (**b**) the TS scheme (Itu et al. 2013a, b)

Eq. (7.15) uses the derivates of $\boldsymbol{R}$ and $\boldsymbol{S}$ with respect to q and A (the quantities terminated with subscript i are computed by interpolation at locations between the grid points). If a viscoelastic wall law is enforced, the kernel displayed in Fig. 7.5b is used to compute the cross-sectional area values and the elastic component of the flow rate. The last operation of each kernel is displayed in a dashed rectangle since it is only performed for the PHCGCC variant of the PHCG algorithm. If the PHCGCC algorithm is used, the values corresponding to the last time step are read either from the regular arrays, or from the compact arrays displayed in Fig. 7.4c during the first operation of each of the two kernels displayed in Fig. 7.5. Synchronization barriers are used between the individual steps if, during the subsequent step, threads access values computed by other threads (these values are typically stored in the shared memory arrays). The synchronization barriers displayed in Fig. 7.5 are inserted at GPU thread block level (using __syncthreads()), while the synchronization barriers displayed in Fig. 7.3 are inserted at CPU level (using cudaDeviceSynchronize()).

The PHCG workflow introduced previously in Itu et al. (2012a, b), and reviewed in Fig. 7.3a cannot be used with a viscoelastic wall law. This is due to the additional steps required by the operator splitting scheme employed for this type of wall law. Consequently, we have introduced a new workflow, as illustrated in Fig. 7.3b. Two different kernels are used: one for the computation of the cross-sectional area and of the elastic flow rate: Eqs. (7.1) and (7.18); and a second one for the computation of the viscoelastic flow rate: Eq. (7.20).

The execution configuration of the first kernel is the same as in the case of an elastic wall law. Host and device instructions are executed in parallel at the beginning of each iteration. After a first synchronization barrier, the values at or next to the junction points are interchanged in order to prepare the computation of the viscoelastic flow rate (in Eq. (7.20) the new values of the cross-sectional area and of the elastic flow rate at all grid points are required), followed by the computation of the viscoelastic and the total flow rate. To solve the tridiagonal system of equations on the device, we employed an optimized CR (Cyclic Reduction)–PCR (Parallel Cyclic Reduction) algorithm (Zhang et al. 2010). Finally, the new flow rate values next to the junction points are copied back to the host and a second synchronization barrier is introduced at the end of the iteration.

For the kernel which computes the viscoelastic flow rate we use an execution configuration with a number of blocks equal to the number of arterial segments. The number of threads of each block is set equal to the smallest power of two value which is higher than the number of grid points in the longest arterial segment. This enables an efficient execution of the CR-PCR algorithm on the GPU.

Figure 7.6 displays the kernel and the shared memory arrays used for the computation of the viscoelastic component of the flow rate and of the total flow rate. First the tridiagonal system is set up (i.e. the coefficients of the three diagonals and of the RHS are computed). The CR-PCR algorithm is composed of three main steps, two forward reduction (CR and PCR, respectively) and one backward substitution (CR) step. Next, the total flow rate is determined and the new flow rate values are stored in the compact arrays if the PHCGCC algorithm is used.

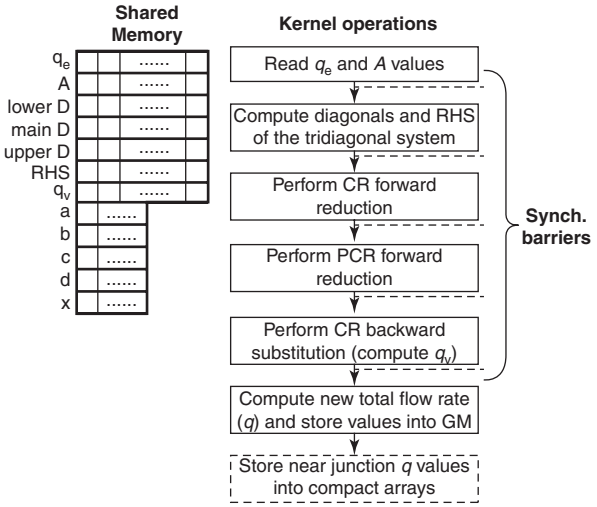

Fig. 7.6 Kernel operations and shared memory arrays used for the computation of the viscoelastic component of the flow rate and of the total flow rate (Itu et al. 2013a, b)

7.2.1.7 Parallel GPU only (PGO) Implementation

The necessity to perform copy operations at each iteration reduces significantly the overall performance of the PHCG algorithm. Hence, we performed an implementation whereas all grid points are computed on the device. This eliminates the memory copies (only the memory copies at the print iterations are required), but also forces the device to perform less parallelizable computations required for the junction points. Another disadvantage of the PGO algorithm, compared to the PHCG algorithm, is that since all operations are performed on the GPU, the task-level parallelism between CPU and GPU is lost. Figure 7.7a displays the workflow for the most complex case, namely when a viscoelastic wall law is used together with the ST boundary condition. A maximum of three kernels are executed at each iteration:

1. Computation of the convolution integral (a multiply-sum scan operation (Sengupta et al. 2008)): Eq. (7.10);
2. Computation of the new cross-sectional area and of the elastic flow rate: Eqs. (7.1) and (7.18);
3. Computation of the viscoelastic flow rate: Eq. (7.20).

The execution configuration of the first kernel is organized as follows: the number of blocks is equal to the number of arterial segments and the number of threads is set to 512. Since the number of time steps per heart cycle (which varies between 8000 and 38,000 for different grid space values) is much higher than the number of threads per block, first each thread performs multiple multiply-sum operations and stores the result in a static shared memory array (composed of 1024 double precision elements). Finally the threads perform a scan operation for the shared memory array and store the result in the global memory. The execution configuration of the other two kernels is the same as the one described in the previous section.

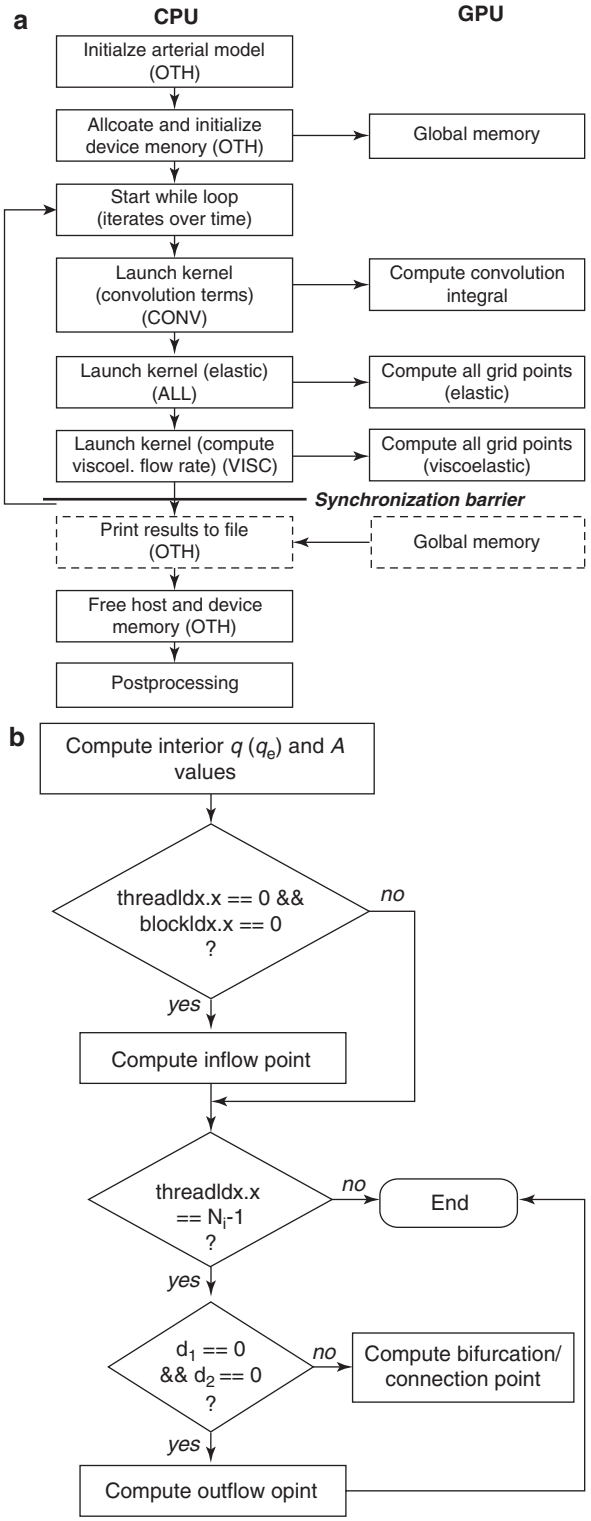

Fig. 7.7 (**a**) Generic GPU workflow when a structured tree boundary condition is used and a viscoelastic wall law is enforced. All of the computations are performed inside GPU kernels and the CPU only coordinates the operations; (**b**) Kernel operations used for the computation of the new values at all grid points (Itu et al. 2013a, b)

If the WK boundary condition is used, the first kernel is not called, and if an elastic wall law is used the third kernel is not called. An acronym is displayed in Fig. 7.7 for each operation to easily match the execution times discussed in the next section with the operations.

Figure 7.7b displays the kernel operations used to compute the new cross-sectional area and flow rate values at all grid points of a vessel segment, with a focus on the junction points. First, the interior points are computed as displayed in Fig. 7.5 (the individual operations have not been detailed). Next, the first thread of the first block solves the inlet point and the last thread of each block solves the outlet or the bifurcation/connection point (a connection point is also a junction point, which is introduced if an arterial segment is split into several domains). Thus, for the junction points, parallelism is only present at block level and not at thread level.

7.2.2 Results

Blood was modeled as an incompressible Newtonian fluid with a density of $\rho = 1.055$ g/cm^3 and a dynamic viscosity of $\mu = \nu \bullet \rho = 0.045$ dynes/cm^{2}s for all the computations.

To compare the performance of the different algorithms (SCO, MCO, three PHCG variants and PGO), the arterial tree detailed in Stergiopulos et al. (1992), and displayed in Fig. 7.8 was used. It is composed of 51 arteries. A time-varying flow rate profile was imposed at the inlet (Olufsen et al. 2000), and for the outlets, the WK and the ST boundary conditions were applied (the parameter values displayed in Table 7.3 were used). The total resistance and the compliance values were set as in Bessems (2008), and the minimum radius used for the generation of the structured tree was tuned ad-hoc so as to obtain a similar total resistance as for the WK outlet boundary condition (total resistance:1.37×10^3 dynes s/cm^5). This aspect allowed us to adequately compare the time-varying flow rate and pressure profiles obtained with the two types of physiologically motivated boundary conditions.

For the ST boundary condition we used an exponential factor equal to 2.7. The constants characterizing the asymmetry of the binary tree were set to 0.908 and 0.578, and the length-to-radius ratio was equal to 50. The elastic properties of the wall were set equal for both the proximal domain and for the structured trees. Together with the minimum radius at which the structured tree is terminated, these parameters determine the compliance of the boundary condition.

The single-threaded CPU algorithm (SCO) was executed on single Intel i7 CPU core with 3.4 GHz, the multi-threaded CPU algorithm (MCO) was executed on an eight-core i7 processor, while for the parallel algorithms (PHCG, PGO) a NVIDIA GPU GTX680 (1536 cores on 8 streaming multiprocessors with 192 cores, 48 kB of shared memory and 64K registers) was used (the GTX680 is based on the Kepler architecture). All computations were performed with double precision floating-point

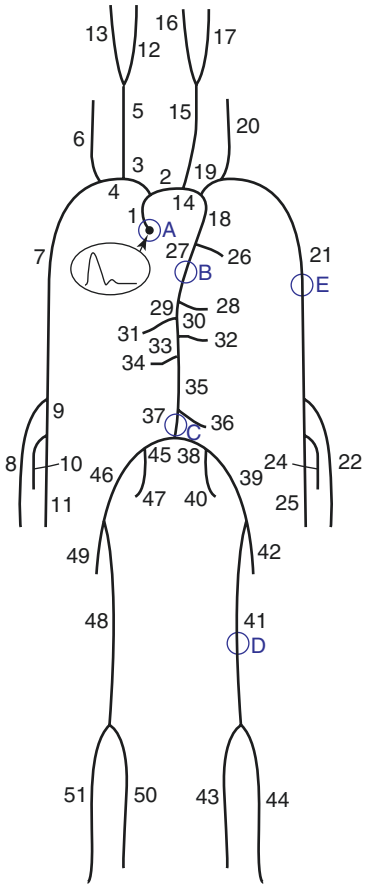

Fig. 7.8 Representation of the 51 main arteries in the human arterial system; the artery numbers of the outlet segments correspond to those displayed in Table 7.3 (Itu et al. 2013a, b).

data structures, since single precision would affect the accuracy of the results, especially at the junction points where the method of characteristics is applied based on the Newton method.

In the following subsections, we discuss the performance of the parallel algorithms and the simulation results obtained under different computational setups.

7.2.2.1 Comparison of Parallel and Sequential Computing and with Different Numerical Schemes

Taking the results determined with the SCO algorithm as reference, we computed the L_2 norms of the absolute differences between the reference numerical solution and the numerical solution obtained with the PHCG and PGO algorithms. All L_2 norm results were smaller than 10^{-13}, i.e. close to the precision of the double-type value in computer data structures (both numerical schemes, LW and TS, were used, but differences were only computed between results obtained with the same numerical scheme).

Table 7.3 Parameters of the outlet vessels used for the Windkessel boundary condition (R_p, R_d, C) and for the structured tree boundary condition (r_{min})

Art. Nr.	r_{top} (cm)	r_{bot} (cm)	Length (cm)	R_p [g/(cm^4 s)]	R_d [g/(cm^4 s)]	C (10^{-6} cm^4 s^2/g)	r_{min} (cm)
6	0.188	0.183	14.8	8.693	28.007	58.7	0.00235
8	0.174	0.142	23.5	17.165	61.434	25.9	0.00182
10	0.091	0.091	7.9	59.782	238.61	6.6	0.0012
11	0.203	0.183	17.1	8.693	28.007	59.0	0.00235
12	0.177	0.083	17.7	76.989	316.51	4.8	0.0011
13	0.177	0.083	17.7	76.989	315.81	4.8	0.0011
16	0.177	0.083	17.7	76.989	316.51	4.8	0.0011
17	0.177	0.083	17.7	76.989	315.81	4.8	0.0011
20	0.188	0.186	14.8	8.339	28.360	58.7	0.0022
22	0.174	0.142	23.5	17.165	61.434	25.9	0.00182
24	0.091	0.091	7.9	59.782	238.61	6.6	0.0012
25	0.203	0.183	17.1	8.693	28.007	59.0	0.00235
26	0.20	0.15	8.0	14.755	51.844	28.3	0.00193
28	0.30	0.30	1.0	2.796	5.504	268.0	0.0039
31	0.435	0.435	5.9	1.313	11.486	431.0	0.00007
32	0.26	0.26	3.2	3.792	9.007	162.0	0.0033
34	0.26	0.26	3.2	3.792	9.007	162.0	0.0033
36	0.16	0.16	5.0	12.378	42.521	34.0	0.00205
40	0.20	0.20	5.0	6.955	21.144	92.6	0.00255
42	0.255	0.186	12.6	8.339	26.560	62.5	0.0024
43	0.247	0.141	32.1	17.506	62.694	30.0	0.00182
44	0.13	0.13	34.3	21.969	80.330	22.1	0.0017
47	0.20	0.20	5.0	6.955	21.144	92.6	0.00255
49	0.255	0.186	12.6	8.339	26.560	62.5	0.0024
50	0.247	0.141	32.1	17.506	62.694	30.0	0.00182
51	0.13	0.13	34.3	21.969	80.330	22.1	0.0017

Furthermore, we computed the L_2 norm of the absolute differences between the numerical solution obtained with the LW scheme and the TS scheme, using the SCO algorithm. The norm results were in the order of 10^{-6} cm^2 for the cross-sectional area and of 10^{-5} ml/s for the flow rate, showing that both numerical schemes lead to the practically same results.

7.2.2.2 Comparison of the Memory Copy Strategies for the PHCG Algorithm

We evaluated the performance of the three memory copy strategies for the PHCG algorithm. Table 7.4 displays the execution times of the GPU operations, corresponding to the computation of one heart cycle with an elastic wall, the LW scheme and WK outlet boundary conditions (this is the computational setup considered in (Itu et al. 2012a, b)). For the PHCGCS algorithm, the kernel execution occupies

Table 7.4 Execution times (s) of the GPU operations obtained for the computation of one heart cycle with the three variants of the PHCG algorithm. The results correspond to a computation with elastic walls, the LW scheme and the WK outlet boundary condition. *Copy H→D* refers to a copy operation between the host (CPU) and the device (GPU), while *Copy D→H* refers to a copy operation in the opposite direction

Operation	PHCGCS	PHCGCA	PHCGCC
Copy H→D	23.7	3.76	0.85
Kernel	1.86	1.86	2.02
Copy D→H	43.1	5.29	0.89

only 2.7% of the total execution time on the GPU, making the application heavily PCI Express Bus limited. Although the amount of data to be transferred is higher, the PHCGCA algorithm represents an improvement, since the number of copy operations is reduced drastically. The best results are obtained with the PHCGCC algorithm, since the amount of data to be transferred is small as in the first case and the number of copy operations is reduced as in the second case. The only drawback is that some of the threads of the kernel need to populate the additional arrays displayed in Fig. 7.4c. This leads to an increase of 8.6% for the kernel execution time, but the increase is easily compensated by the time gained for the memory copies.

7.2.2.3 Comparison of the Performance Obtained with the SCO, MCO, PHCG and PGO Algorithms

Tables 7.5 and 7.6 summarize execution times measured for the six different computational setups displayed in Table 7.1. The execution times correspond to ten heart cycles and the highest speed-up values are displayed in bold. The grid space has been set to 0.1 cm and the time step to $5.55 \cdot 10^{-5}$s. The values are based on literature data and on the CFL-restriction respectively.

The PHCGCA algorithm cannot be applied for all computational setups investigated herein (as described in Sect. 2.3.1, the workflow in Fig. 7.4a cannot be applied for a viscoelastic wall law). The speed-up values in Tables 7.5 and 7.6 are computed based on the execution time of both the SCO and MCO algorithms.

The speed-up values vary between **5.26**× and **8.55**× compared to the SCO algorithm and between **1.84**× and **4.02**× compared to the MCO algorithm. As anticipated, the PHCGCC algorithm outperforms the PHCGCA algorithm for all cases for which the PHCGCA was applied. For an elastic wall law, in case a WK boundary condition is used, the PHCGCC algorithm performs best, while in case the ST boundary condition is used, the PGO algorithm leads to the highest speed-up. For a viscoelastic wall law, the PHCGCC algorithm performs best, regardless of the type of outlet boundary condition. Execution times are higher with a ST boundary condition because of the time spent for the computation of the convolution integral in Eq. (7.10).

For an elastic wall law, with the PHCGCC algorithm, the execution times are comparable for the LW and the TS scheme (for both outlet boundary condition types), with slight advantages for the LW scheme. For the SCO and MCO algorithms, the LW scheme is superior to the TS scheme.

Table 7.5 Execution times and speed-ups obtained for the computation of ten heart cycles with the SCO, MCO, and PGO algorithms. The first four cases correspond to an elastic wall with either the Lax-Wendroff (LW) or the Taylor series (TS) scheme and with a Windkessel (WK) or structured tree (ST) boundary condition. The last two cases correspond to a viscoelastic wall law with the TS scheme and with a WK or ST boundary condition

						PGO		
							Speed-up	
Case	Num. sch.	Wall law	Outlet BC	SCO (s)	MCO (s)	Time (s)	SCO	MCO
1	LW	Elastic	WK	273.4	81.13	101.98	2.68×	0.79×
2	LW	Elastic	ST	673.7	205.38	111.49	**6.04**×	**1.84**×
3	TS	Elastic	WK	396.3	119.37	105.10	3.77×	1.14×
4	TS	Elastic	ST	797.2	233.09	116.56	**6.84**×	**2.00**×
5	TS	Viscoel.	WK	774.4	384.43	235.14	3.29×	1.63×
6	TS	Viscoel.	ST	1179.6	501.08	241.45	4.89×	2.07×

Table 7.6 Execution times and speed-ups obtained for the computation of ten heart cycles with the PHCGC algorithms. The first four cases correspond to an elastic wall with either the Lax-Wendroff (LW) or the Taylor series (TS) scheme and with a Windkessel (WK) or structured tree (ST) boundary condition. The last two cases correspond to a viscoelastic wall law with the TS scheme and with a WK or ST boundary condition.

				PHCGCA			PHCGCC		
					Speed-up			Speed-up	
Case	Num. sch.	Wall law	Outlet BC	Time (s)	SCO	MCO	Time (s)	SCO	MCO
1	LW	Elastic	WK	68.21	4.01×	1.19×	42.4	**6.45**×	**1.91**×
2	LW	Elastic	ST	182.34	3.69×	1.13×	149.92	4.49×	1.37×
3	TS	Elastic	WK	74.81	5.30×	1.59×	46.37	**8.55**×	**2.57**×
4	TS	Elastic	ST	187.27	4.26×	1.24×	151.90	5.25×	1.53×
5	TS	Viscoel.	WK	–	–		95.64	**8.10**×	**4.02**×
6	TS	Viscoel.	ST	–	–		224.38	**5.26**×	**2.23**×

A detailed analysis of the results obtained with the best performing algorithms (PHCGCC and PGO) is presented below. Figure 7.9 displays the percentage of the execution time occupied by each operation identified in the workflows in Fig. 7.3, for cases 1, 2, 5 and 6, computed with the PHCGCC algorithm. Regarding the computations with an elastic wall law, as is displayed in Fig. 7.3a, computations on the host and on the device are performed in parallel. The operations on the device require more time than the host operations if a WK boundary condition is used. Although the copy operations were optimized, they occupy almost half of the total time spent on the GPU. Besides, a considerable time is required for other operations, which include control instructions, data exchange operations between the host arrays and the arrays used for the copy operations, and print operations during the last cycle. If a ST boundary condition is used, the computation of the convolution integral in Eq. (7.10), performed on the host, occupies most of the execution time.

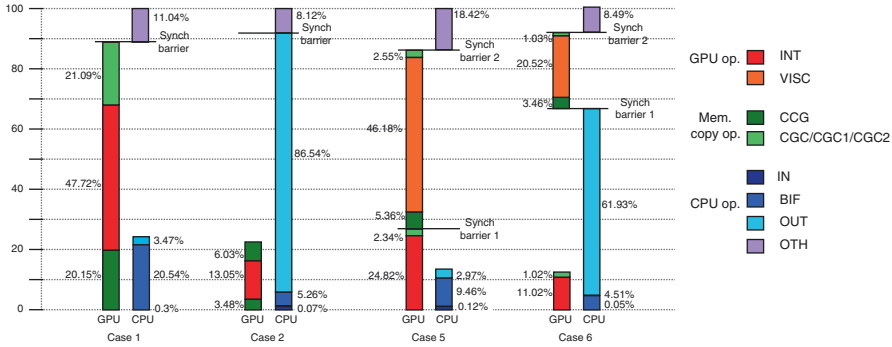

Fig. 7.9 Detailed percentage values of the execution time occupied by the operations identified in the workflow in Fig. 7.3 for the PHCGCC algorithm, for an elastic wall law (cases 1 and 2) and a viscoelastic wall law (cases 5 and 6). Acronyms are detailed in Fig. 7.3 (Itu et al. 2013a, b)

This is the primary reason behind the low speedup achieved with the PHCGCC algorithm and the ST boundary condition. In addition, it also explains the similar speed-up values obtained with the PHCGCC and PHCGCA algorithms.

Regarding the computations with a viscoelastic wall law, as is displayed in Fig. 7.3b, computations on the host and on the device are performed in parallel at the beginning of each iteration, but since the computation of the viscoelastic flow rate requires the values of the elastic flow rate and of the cross-sectional area at the junction points (from the current time step), during the second part of each iteration, only the device performs computations. As for the elastic wall law, in case a ST boundary condition is used, the computation of the convolution integral in Eq. (7.10), performed on the host, occupies most of the execution time.

Figure 7.10 displays the percentage of the execution time occupied by each operation identified in the workflow in Fig. 7.7, for cases 1 and 6, computed with the PGO algorithm. In the first case a single kernel is used, while for case 6 also the convolution integral and the viscoelastic flow rate correction are computed. The computation of the interior and junction points require more execution time for case 6 than for case 1, since, on the one side the TS scheme is used instead of the LW scheme, and on the other side additional operations are performed because of the viscoelastic wall law. Compared to case 6 in Fig. 7.9 (ST boundary condition), the execution time dedicated to the outflow points is reduced significantly since the operations are performed on the device, but because the computation of all grid points requires considerably more time, the total execution time for case 6 is higher with the PGO algorithm than with the PHCGCC algorithm.

Figure 7.11 displays a comparison of the number of heart cycles which can be computed per hour with different algorithms: the SCO and MCO algorithms, the previously introduced PHCGCA algorithm (applied only for non-periodic boundary conditions) and the best performing parallel algorithm for each computational setup as determined in the current study. The four different computational setups have been obtained by combining the different wall laws and outlet boundary conditions and by

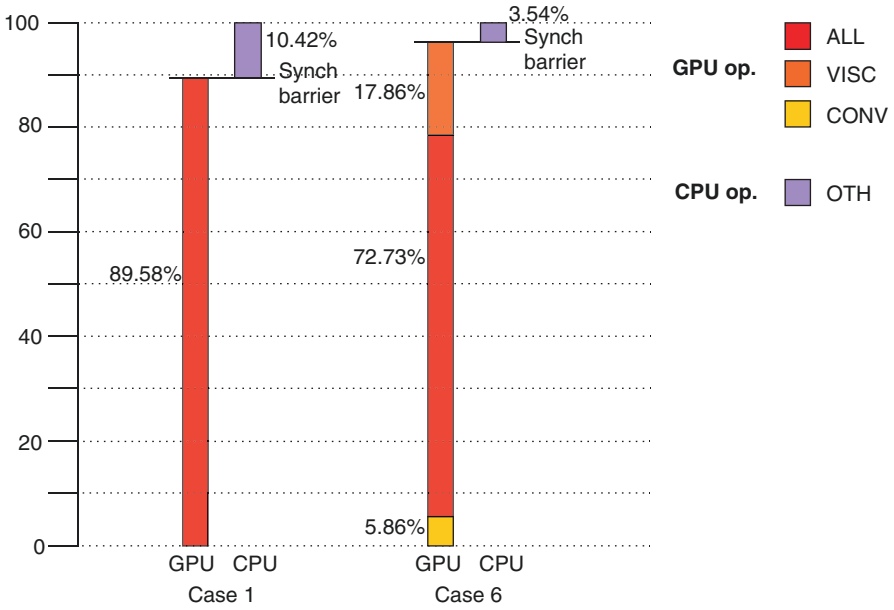

Fig. 7.10 Detailed percentage values of the execution time occupied by the operations identified in the workflow in Fig. 7.7 for the PGO algorithm, for an elastic wall law (case 1) and a viscoelastic wall law (case 6). Acronyms are detailed in Fig. 7.7 (Itu et al. 2013a, b)

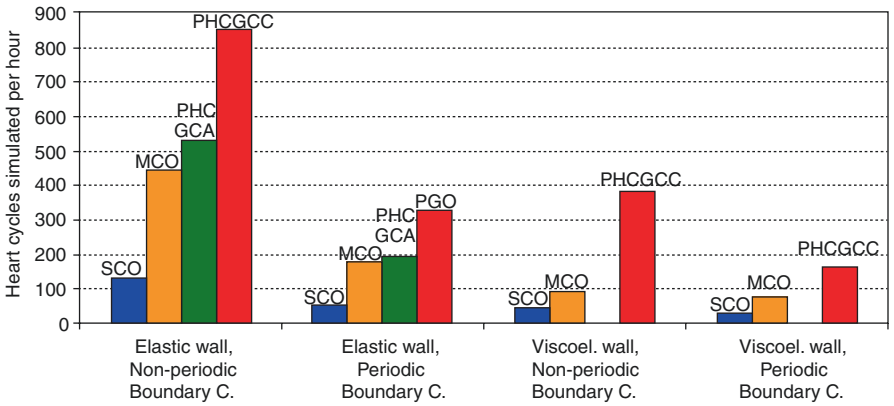

Fig. 7.11 Heart cycles computed per hour for the SCO algorithm, the MCO algorithm and the best performing parallel algorithm for each computational setup (Itu et al. 2013a, b)

choosing the best performing numerical scheme (according to the results in Tables 7.5 and 7.6). The results show that the best performing GPU based algorithms considerably increase the number of heart cycles which can be computed per hour.

To analyze the effect of the simulation parameters on the speed-up factor, we display in Fig. 7.12 the speed-up values obtained for different grid space values: $\Delta x = 0.25$ cm (5486 degrees of freedom (dofs), 8000 time steps per cycle),

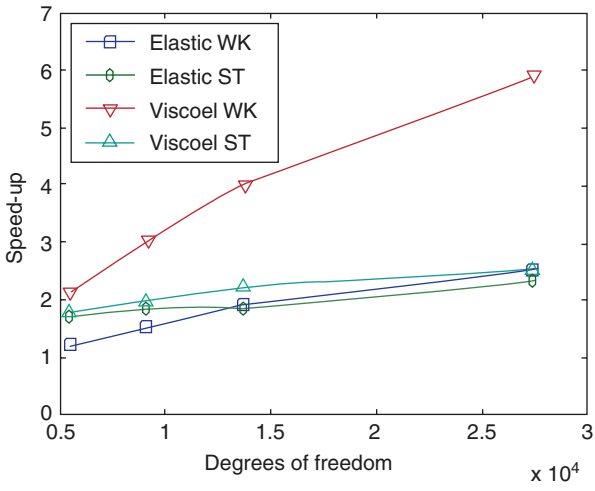

Fig. 7.12 Speed-up values obtained for different grid space configurations for the best performing parallel algorithm compared to the MCO algorithm (Itu et al. 2013a, b)

$\Delta x = 0.15$ cm (9144 dofs, 12,500 time steps per cycle), $\Delta x = 0.1$ cm (13,716 dofs, 18,000 time steps per cycle), and $\Delta x = 0.05$ cm (27,432 dofs, 37,000 time steps per cycle). The displayed values represent the speed-up obtained by the best performing GPU based algorithms compared to the MCO algorithm. The time-step values are chosen to satisfy the CFL condition for each case, and both types of wall laws and outlet boundary conditions are considered. In each case, the numerical scheme and the parallel algorithm applied for the computation correspond to the best speed up value obtained for a grid space of 0.1 cm. Figure 7.12 displays an approximately linear increase of the speed-up value, indicating that the computational power of the GPU is not fully exploited for any of the computational configurations with a grid space higher than 0.05 cm. The increase is moderate for three of the four computational setups and more pronounced in case a viscoelastic wall law is used together with a WK boundary condition. This aspect is given by the fact that the implementation of the viscoelastic wall law is more efficient for the PHCGCC algorithm compared to the MCO algorithm. On the other hand, when a viscoelastic wall law is used together with the ST boundary condition, most of the time is spent for computing the outlet grid points and the difference in execution time for the viscoelastic component becomes less important.

Figure 7.13 displays the time-varying pressure, flow rate and cross-sectional area at the five locations marked with a blue circle in Fig. 7.8.

Each figure contains four plots, which have been obtained with either an elastic or viscoelastic wall and with a WK or ST boundary condition. Since the total resistance introduced by either of the two types of boundary conditions is similar, the average quantities are approximately equal at all locations inside the arterial tree. Referring first to the computations with elastic walls, the pressure values obtained with the ST boundary condition decrease at a later time inside one heart cycle, indicating that the reflected wave arrives later (an aspect which is more pronounced for the proximal parts of the arterial tree). This can be explained as follows: the ST boundary condition simulates the propagation of the waves down to the arteriolar

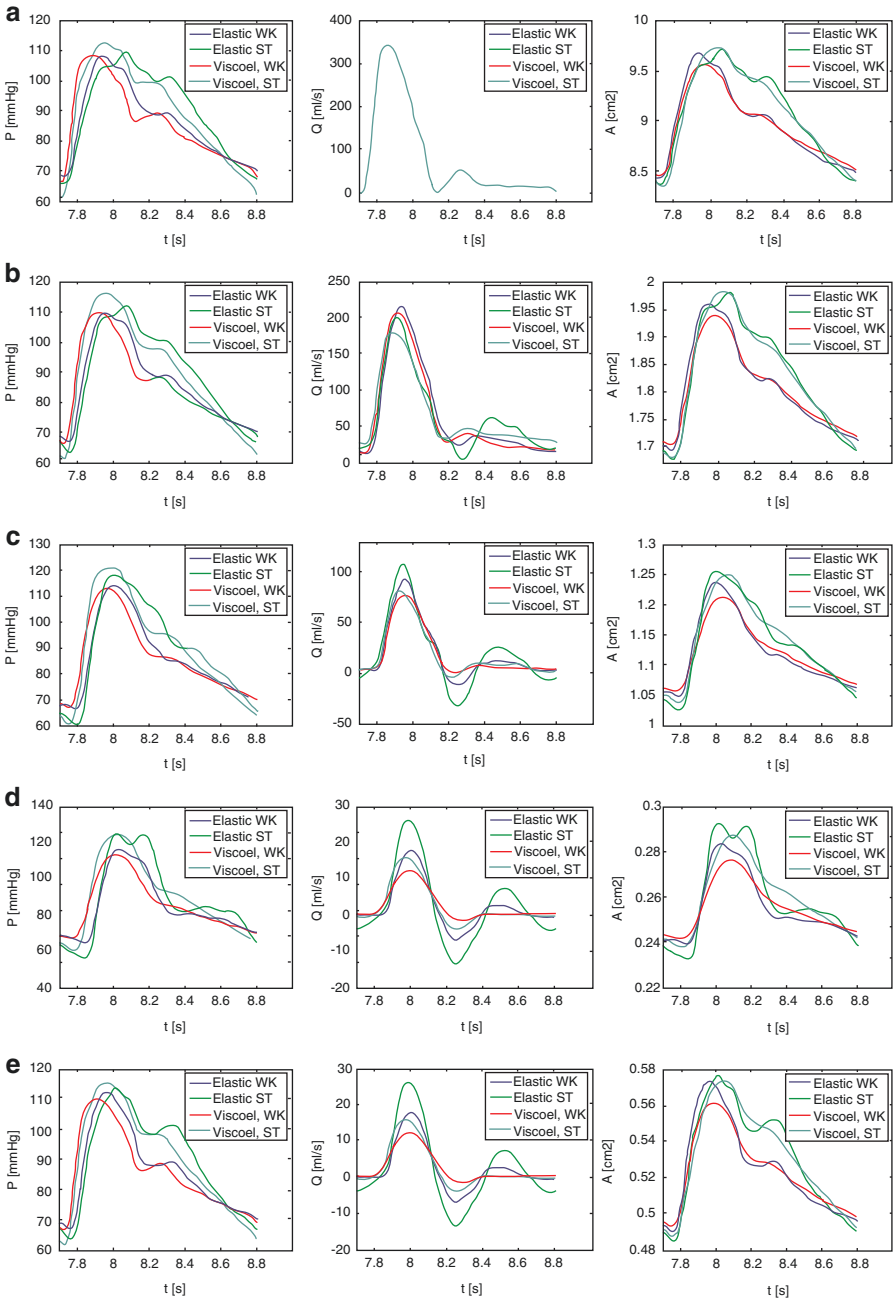

Fig. 7.13 Time-varying pressure, flow rate and cross-sectional area at (**a**) aortic root, (**b**) descending aorta, (**c**) abdominal aorta, (**d**) femoral artery, and (**e**) subclavian artery (corresponding to locations A–E respectively in Fig. 7.6). Four plots are displayed in each figure, which have been obtained with either an elastic or viscoelastic wall and with a WK or ST boundary condition (Itu et al. 2013a, b)

level where the reflections occur primarily, whereas the WK boundary condition, as a lumped model, is not able to capture the wave propagation phenomena in the distal part of the tree and introduces the reflections at the outlet points of the proximal arteries. As a result of the later arriving pressure waves, also the maximum pressure value is reached at a later moment in time. These aspects also lead to higher oscillations inside the flow rate waveforms which are displayed in the second column of Fig. 7.13. Finally, for the cross-sectional area, generally, the variation inside one heart cycle is higher with a ST boundary condition. For the elastic wall, the pressure and the cross-sectional area waveforms are in phase and a more pronounced variation of the area values is reflected by a higher pressure pulse. The higher pressure pulse obtained for the structure tree boundary condition indicates a lower total compliance than the one enforced through the WK boundary condition. We emphasize the fact the compliance of the proximal part of the tree is identical in both cases and the difference in total compliance is given only by the outlet boundary conditions.

When a viscoelastic wall is used, the main difference is that the high-frequency oscillations in the waveforms are reduced. This can be observed in both the pressure and the flow rate waveforms and the phenomenon is more pronounced at the distal locations. These observations are consistent with results reported in literature (Reymond et al. 2011). The introduction of the viscoelastic wall does not change the overall behavior of the WK and ST boundary conditions, the observations mentioned above being still valid, as would be expected. Another important consequence of the introduction of the viscoelastic wall is the fact that pressure and area are no longer in phase, the peak cross-sectional area value being generally obtained at a later moment in time inside one heart cycle.

Furthermore, Fig. 7.14 displays the pressure-area relationships at three different locations. A hysteresis loop can be observed when a viscoelastic wall laws is used, as opposed to the linear variation for an elastic wall law. The area of the hysteresis loop is proportional to the energy dissipation given by the viscoelastic properties of the wall.

7.2.3 Discussion and Conclusions

To test the speed-up potential of novel hybrid CPU-GPU and GPU only based implementations of the one-dimensional blood flow model, we have used a full body arterial model and have applied two physiologically motivated outlet boundary conditions, 3-element windkessel circuits as non-periodic boundary condition and structured trees as periodic boundary condition, and two different types of constitutive wall laws. The speed-up values over a multi-threaded CPU based implementation range from 1.84× to 4.02×, and over a single-threaded CPU based implementation range from 5.26× to 8.55×, thus significantly improving on previously reported parallel implementations and confirming the excellent speed-up potential of the GPU-based implementation.

The results showed that, for an elastic wall, if a non-periodic boundary condition is used, the PHCGCC algorithm performs best, while for a periodic boundary condition, the PGO algorithm performs best. This is motivated by various aspects. First of all, the PHCGCC algorithm decreases the execution time not only through data

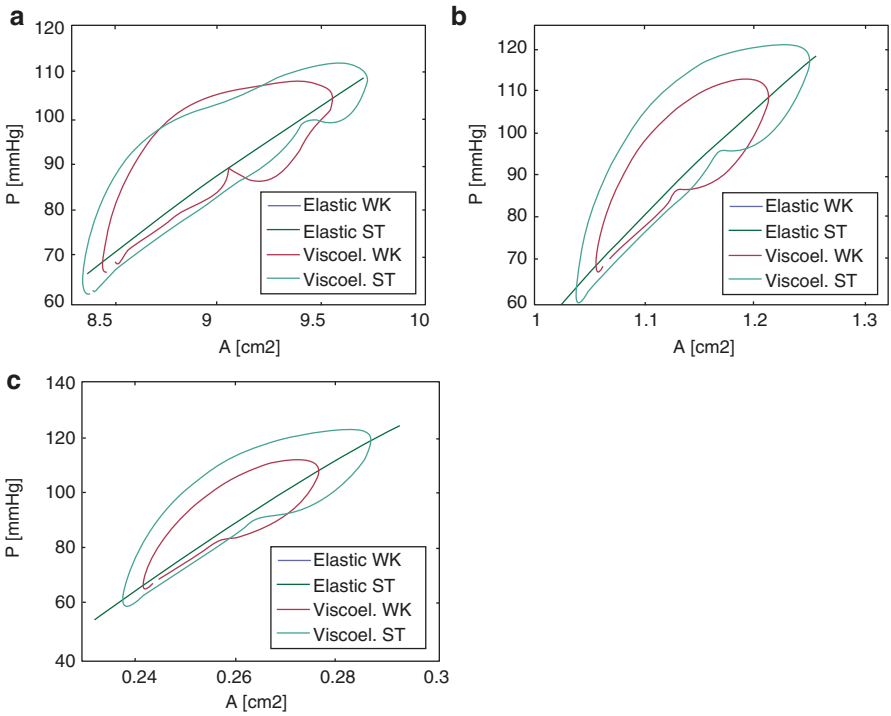

Fig. 7.14 Pressure-area relationships at (**a**) aortic root, (**b**) abdominal aorta, and (**c**) femoral artery (corresponding to locations A, C and respectively D in Fig. 7.8). Four plots are displayed in each figure, which have been obtained with either an elastic or viscoelastic wall and with a WK or ST boundary condition (Itu et al. 2013a, b)

parallelization on the GPU, but also through task level parallelization between CPU and GPU. For periodic boundary conditions, the execution time required specifically for the outlet points is more than one order of magnitude higher than for non-periodic boundary conditions (Fig. 7.9). Since for the PHCGCC algorithm the outlet points are computed on the CPU, the speed-up obtained for a periodic boundary condition is limited. In this case, the PGO algorithm performs better, although it does not employ task level parallelization, because the multiply-sum scan operation required for the convolution integral is more efficient on the GPU (especially when the number of time steps per cycle is high, as is here the case). In case a viscoelastic wall law is used, the PHCGCC algorithm performs best for both types of boundary conditions. Although the PGO reduces the execution time dedicated to the outlet points, it is slower than the PHCGCC algorithm because the computation of all grid points increases significantly due to the viscoelastic wall law.

As has been shown in Fig. 7.12, the speed-up potential of the GPU-based algorithms is even higher when the number of grid points increases. This is given by the fact that, for the standard configuration with $\Delta x = 0.1$ cm, only 6858 threads are generated at the kernel execution, whereas the GPU is able to run grids with tens of thousands of threads. On the other side, even when the number of threads is

decreased 2.5 times ($\Delta x = 0.25$ cm), still significant speed-up values are obtained. This is obtained as a result of the activities performed for the kernel optimization, which optimized the kernel towards high computational intensity and few global memory operations. Whereas the grid space is a crucial factor for the final speed-up value, the number of time steps does not influence the results, since different iterations are executed sequentially both on the CPU and on the GPU.

Furthermore, when the interior points of the arteries are solved on the GPU, the maximum number of grid points for a single segment is limited by the resources of the GPU (number of registers, amount of shared memory and maximum number of threads per block). Consequently, if an arterial vessel contains too many grid points, it is split into separate segments and junction points are introduced which are treated similarly to bifurcation points (conservation of flow rate and pressure). For the arterial tree used herein, this approach has been required only when a grid space with $\Delta x = 0.05$ cm was used and only for the vessels longer than 28.2 cm (the main limiting factor has been the shared memory). Furthermore, we emphasize that a generic arterial model was used for demonstrating the algorithm. For patient-specific models, the geometry will have slight variations in terms of lengths and radiuses. This, however, does not affect the proposed algorithm and the memory requirements.

Next, we focus on a theoretical analysis of the performance gain obtained through the GPU, based on the standard computational setup with $\Delta x = 0.1$ cm. CUDA introduced the single instruction multiple threads (SIMT) architecture whereas an operation can be executed in parallel on p processors. By analogy with a SIMD system (Jordan and Alaghband 2003), p represents in this case the number of cores of the GPU. The total number of interior grid points is:

$$N = \sum_{i=1}^{m} \left(N_i - 2 \right), \tag{7.21}$$

where m is the total number of arteries and N_i is the number of grid points of each artery.

Referring first to the computation of the interior grid points, if n_I different operations are required to compute the new cross-sectional area and flow-rate values at a single interior grid point, the theoretical computing time is:

$$T_p = \frac{n_I \cdot N}{p} \tag{7.22}$$

for the PHCG algorithm and:

$$T_1 = n_I \cdot N \tag{7.23}$$

for the SCO algorithm. Hence, the theoretical parallel speed-up and the efficiency are respectively:

$$S_p = \frac{T_1}{T_p} = p, \qquad E_p = \frac{S_p}{p} = 100\%. \tag{7.24}$$

The performance of GPU kernels is limited by either the global memory bandwidth or by the computational intensity. The kernel which computes the interior grid points has an overall global memory throughput of only 11.3 GB/s, as opposed to the peak theoretical of 192.2 GB/s of the GTX680 card. The instruction throughput on the other side is of 69.91 GFLOPS (Giga Floating Point Operations per Second), with $5.71 \bullet 10^6$ warps executed per second. The technical specifications of the GTX680 only contain the theoretical single precision GFLOP value (3090 GFLOPS), whereas the computations for the one-dimensional model are performed in double precision. As is specified though in, the instruction throughput for double precision computations is significantly lower than the instruction throughput for single precision computations for the GTX680 architecture.

Since for the GTX680 $p=1536$, the ideal execution time of the interior points should be 1/1536 of the sequential execution time. Taking case 1 in Tables 7.5 and 7.6 as a representative case, the execution time of the interior points for the PHCGCC algorithm is of 20.23 s, while for the SCO algorithm it is of 240.45 s. Thus, the speedup is of only 11.88× and not 1536×. The great difference between the theoretical speedup and the empirical speedup is given by (1) not all thread blocks can be executed simultaneously because of the shared memory limitation—occupancy is limited (only one thread block can be executed at a time on each of the 8 streaming multiprocessors of the GTX680, whereas there are 51 thread blocks in total), (2) the necessity to perform synchronization at thread level, (3) the parallelism is limited for a grid space of 0.1 cm, and (4) the number of double precision floating point units is significantly smaller than the number of cores.

Secondly, we refer to the computation of the viscoelastic component of the flow rate. The size of the tridiagonal system varies between the different vessel segments, leading to an average length of 14 cm for the arteries of the tree displayed in Fig. 7.8. Hence the average size of the tridiagonal system is 141. This value is rounded up to a power of 2, leading to a tridiagonal system of size $n_T=256$. For the Thomas algorithm $8nT$ operations are executed, while for the CR-PCR parallel algorithm $17(n_T-m_T)+12m_T\log_2 m_T$ operations are executed, where m_T is the size of the system solved with the PCR algorithm ($m_T=128$ on average). Hence:

$$S_p = p \cdot \frac{8n_T}{17\left(n_T - m_T\right) + 12m_T \log_2 m_T} = 243.33, \qquad E_p = \frac{S_p}{p} = 15.84\%. \quad (7.25)$$

Similarly to the first kernel, the performance of the kernel which computes the viscoelastic component of the flow rate is also limited by the computational intensity. The overall global memory throughput is of only 1.3 GB/s, whereas the instruction throughput is of 77.94 GFLOPS, with $3.12 \bullet 10^6$ warps executed per second.

Taking case 5 in Tables 7.5 and 7.6 as a representative case, the execution time of the viscoelastic flow rate for the PHCGCC/PGO algorithms is of 44.17 s, while for the SCO algorithm it is of 376.49 s. Thus the speed-up is of only 8.52× and not 243.33×. Additionally to the reasons enumerated above (since shared memory usage for this kernel is similar to the first kernel, occupancy is limited: only one thread block can be executed at a time on a streaming multiprocessor), also the

fact that the $17(n_T - m_T) + 12m_T \log_2 mT$ operations cannot be executed in parallel plays an important role (the CR-PCR algorithm is executed on average in $2\log_2 n_T - \log_2 m_T - 1 = 8$ sequential steps).

Thirdly, we refer to the computation of the convolution integral (for the PGO algorithm). Each multiplication/addition pair of this operation is combined into a single floating point operation, leading to a total of n_C operations, where n_C is the number of time steps for one heart cycle. Thus,

$$S_p = p = 336, \qquad E_p = \frac{S_p}{p} = 100\%. \tag{7.26}$$

Unlike the first two kernels, the performance of the kernel which computes the convolution integral is limited by the global memory bandwidth. The overall global memory throughput is of 184 GB/s, which is close to the peak theoretical value, whereas the instruction throughput is of only 21.8 GFLOPS, with $7.98 \bullet 10^6$ warps executed per second.

Taking case 2 in Tables 7.5 and 7.6 as a representative case, the execution time of the convolution integral for the PGO algorithm is of 14.14 s, while for the SCO/PHCGCC algorithms it is of 407.13 s. Thus the speed-up is of only 28.79× and not 1536×. Additionally to reasons (2) and (4) enumerated during the analysis performed for the interior grid points, also the fact that the n_C operations cannot be executed in parallel, and the fact that the kernel is limited by the global memory throughput, play an important role (the optimized scan algorithm uses $\log_2 n_C = 11$ sequential steps). In this case the number of simultaneously active threads on a streaming multiprocessor is close to maximum, leading to an occupancy of 81% (the shared memory does not limit the occupancy since each block requires only 8 kB of shared memory).

As expected, the overall speed-up of the application is lower than the speed-up of the individual components because of the initialization activities, control operations, task-level synchronization, writing of results to files, data exchange operations between the host arrays and the arrays used for the copy operations (PHCGCC algorithm), limited parallelism of junction points operations (PGO algorithm), etc.

Our past work analyzed the speed-up potential only for one computational setup (elastic wall, windkessel outlet boundary condition as non-periodic BC and the Lax-Wendroff numerical scheme) of the six scenarios considered in the present study and only compared with the SCO algorithm (Itu et al. 2012a, b). Additionally, a different GPU (GTX460) was used, and only the PHCG approach was considered, for which, from the three different variants introduced herein, only the PHCGCS and PHCGCA algorithms were investigated. As can be seen in the results section (Tables 7.5 and 7.6 and Fig. 7.11), the third variant (the PHCGCC algorithm), outperforms the previous two algorithms. Additionally, a new PHCG workflow, based on the operator splitting scheme, was developed to be able to employ any of the PHCG variants in case a viscoelastic wall law is applied (Fig. 7.3b).

The consideration of different computational setups has been a crucial aspect, since it demonstrated the limited speed-up obtained for the PHCG variants with periodic boundary condition and an elastic wall law. The PGO algorithm proposed

in the current study leads to considerably improved execution times for these cases (case 2, and 4 in Tables 7.5 and 7.6). When comparing the results against (Itu et al. 2012a, b), one has to take into consideration that previously a grid space of 0.05 cm was used, for which a speed-up of 4.7× was obtained for the PHCGCA algorithm with the GTX460 (this speed-up changes to 4.01× for a grid space of 0.1 cm with the GTX680).

To our knowledge, the only other previous research focused on the acceleration of the one-dimensional blood flow model has been reported in (Kumar et al. 2003). An Origin 2000 SGI machine with eight processing elements was used and the Message Parsing Interface (MPI) libraries have been applied for the communication between the processing nodes. An elastic wall law was used together with a Taylor-Galerkin numerical scheme and the results were mainly reported for up to four processing nodes. An arterial model composed of 55 arteries, very similar to the one adopted herein, has been used and the speed-up factor did not exceed around 3.5× even if the grid space was decreased to 0.05 cm (~3.25× for $\Delta x = 0.1$ cm). We have seen that in our case the speed-up is of 6.45× for a grid space of 0.1 cm. In (Kumar et al. 2003) non-reflecting and resistance-based boundary condition, which are both non-periodic boundary conditions, have been used. Herein we obtained for the MCO algorithm, compared to the SCO algorithm, a similar speed-up as in (Kumar et al. 2003) (3.37× for the computational setup with an elastic wall law and wind-kessel boundary condition). These results show that a GPU is better suited for the acceleration of the one-dimensional blood flow model than a multi-threaded CPU based configuration.

Overall, we think that the advantages of a GPU-based implementation of the one-dimensional blood flow model outweigh the costs. Most importantly, the acceleration of the execution time is crucial when the blood flow model is applied in a clinical setting (Itu et al. 2012a, b, 2013a, b). This is given not only by the fact that results are required in a timely manner, but also by the necessity of applying tuning procedures which increase the computational intensity. Such accelerated approaches (when validated) are ideal for interventional settings, where near real-time information is needed to make the clinical decision. Under such settings, the measurement data is often acquired when the patient is undergoing an intervention. As a result, the tuning and simulation process for the hemodynamic simulation should be fast enough to generate pressure and flow information that can be used during the procedure.

Secondly, as pointed out in the introduction, research activities can also benefit from the acceleration if results depend on running hundreds or even thousands of computations with different configurations.

On the other side, costs are limited, both financially (the results reported herein have been obtained on a regular desktop computer equipped with an NVIDIA graphics card) and from the development time point of view. Especially, when the PHCGCC algorithm is employed, only the computation of the interior points needs to be ported to the GPU. Furthermore, even when the PGO algorithm is required, i.e. when periodic outlet boundary conditions are imposed, the additional development time does not exceed a couple of weeks.

7.3 GPU Accelerated Voxelizer

Solid voxelization represents the process of transforming a polygonal mesh into a voxel representation by associating each polygon of a mesh with the cells in the voxel grid. Voxel representation of solids are currently used in many applications such as physics simulations, collision detection, volume rendering, and many others. The main advantage of the voxel representation of a solid is that each voxel in the grid can be accessed directly by knowing its position in space or its position relative to another voxel, without performing a search operation, whereas in a mesh representation information is described sparsely as a set of polygons, by providing the position of each point explicitly.

Although there are many studies on this topic, solid voxelization remains a difficult problem, mostly because of computational complexity and issues related to robustness

To perform a Fluid-Structure interaction (FSI) using the Lattice-Boltzmann method (LBM) the moving geometry has to be embedded in a Cartesian grid of uniformly distributed points using a signed distance field $\phi(\mathbf{x})$. However, the geometry is typically given as a sequence of non-uniform polygonal meshes. A surface voxelization operation is required to compute the distance field. The main challenge of voxelization consists in associating each vertex $\mathbf{v}i$ of a polygonal mesh to each node $\mathbf{x}i$ of the Cartesian grid. Typically, the size of the grid is between 500,000 and 50,000,000 nodes while the size of the mesh is between 50,000 and 300,000. This makes the voxelization a computationally expensive operation.

For the FSI computations, since the surface is moving, the voxelization operation is required at each solver iteration to update the position of the surface. With the classical method (CPU based implementation), the surface voxelization operation is the performance bottleneck as it occupies around 50% of the total computation time (Nita et al. 2015). Therefore, it is crucial that an efficient implementation is developed.

7.3.1 The Classical Method

The Cartesian grid is defined from a three-dimensional image by its dimensions (N_x, N_y, N_z), an origin $\mathbf{o}$ and a grid spacing δx (the grid nodes are uniformly distributed hence $\delta_x = \delta_y = \delta_z$). The grid size is chosen to satisfy the flow solver stability constraints. The origin $\mathbf{o}$ and the grid spacing δx is used to transform from physical coordinates to grid coordinates (and vice versa) i.e. to find the voxel (i, j, k) that corresponds to a point $\mathbf{p} = (p_x, p_y, p_z)$**.** The transformation is defined as follows:

$$i = \frac{p_x}{\delta_x} - o_x, j = \frac{p_y}{\delta_x} - o_y, j = \frac{p_z}{\delta_x} - o_z, \tag{7.27}$$

where $\lfloor x \rfloor$ denotes the floor function. And the inverse transformation:

$$p_x = i\delta x + o_x, p_y = i\delta x + o_y, p_z = k\delta x + o_z, \tag{7.28}$$

A mesh is defined as a set of triangles $T=(\mathbf{v}_1,\mathbf{v}_2,\mathbf{v}_3)$, for each triange we compute an axis-aligned bounding-box (AABB) as $AABB=(\mathbf{v}min,\mathbf{v}max)$ so that the triangle will completely fit inside it, furthermore the AABB is enlarged in all directions using a small value $(2-3\delta x)$ so that the triangle vertices will never be located exactly on the AABB wall.

The classic method for surface voxelization consists in simply looping over each grid node $\mathbf{x}_i$ in each AABB and computing the signed distance $\phi(\mathbf{x}_i)$. To find all the grid nodes inside the AABB, one needs to transform $(\mathbf{v}min,\mathbf{v}max)$ to grid coordinates to get $(imin,jmin,kmin)$ and $(imax,jmax,kmax)$ and then loop over all i,j and k values located inside the bounds.

The signed distance function is defined as follows:

$$\phi(\mathbf{x}) = d(\mathbf{x},\mathbf{x}_\perp)\,\mathrm{sgn}\left[(\mathbf{x}-\mathbf{x}_\perp)^\circ \cdot \mathbf{n}\right], \tag{7.29}$$

where $\mathbf{n}$ is the triangle normal and $\mathbf{x}_\perp$ is the closest point to $\mathbf{x}$ on the triangle. The second factor in the above expression represents the sign, i.e. it will be negative or positive depending on which side of the triangle, the point $\mathbf{x}$ is located. For adjacent triangles the AABBs will intersect and will result in multiple ϕ values for the same grid point $\mathbf{x}$, one value for each AABB that point $\mathbf{x}$ is included in (Fig. 7.15). In this case the absolute minimum value of ϕ will be chosen.

For the GPU implementation, the loop that processes the mesh triangles is parallelized so that one GPU thread will process one triangle. However there are several downsides that causes very poor GPU utilization in this case. The main problem arises at the adjacent triangles where the AABBs intersect. In the intersection

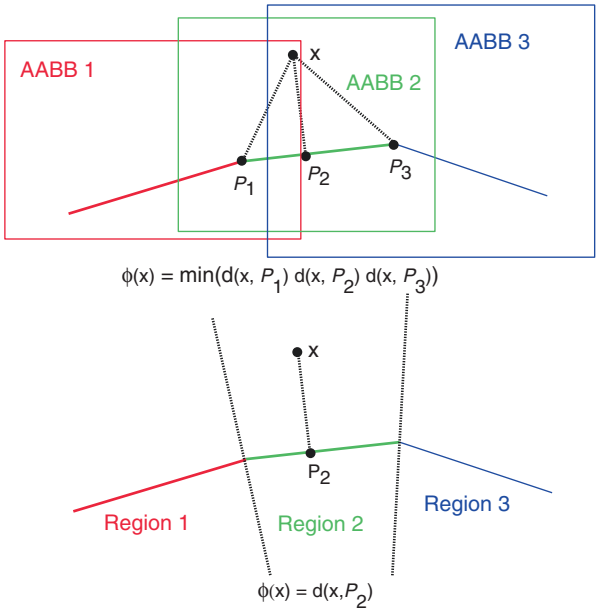

Fig. 7.15 Two-dimensional analogy of the surface voxelization algorithm. The classic approach (up): ϕ is computed for all the nodes inside an AABB. And the separating planes technique (down): nodes that correspond exclusively to a facet are identified using separating planes

regions, there will be multiple threads that need to update the ϕ value at the same grid node $\mathbf{x}$. In this case a synchronization operation is required to ensure that only one thread will update one location at the same time. The synchronization operation drastically reduces parallelism and GPU performance.

The other limitation is given by the fact that each GPU thread will process a different number of grid nodes because of the different AABB sizes. More specifically, the number of the grid nodes in an AABB is influenced by the size and orientation of the corresponding triangle. To achieve maximum performance with a GPU based implementation, all the threads should execute the same operations.

7.3.2 The Separating Plane Technique

The classical method can be improved by redefining the way grid nodes are associated with mesh triangles. Instead of computing the ϕ value for all the nodes in an AABB it is possible to identify a priori the nodes for which each mesh triangle will give the minimum ϕ. Hence, there will no longer be threads that will need to update ϕ at the same location $\mathbf{x}$. This method was initially presented in Janßen et al. Janßen et al. (2015)).

For each triangle we define a region so that each point $\mathbf{x}$ in that region has the closest point $\mathbf{x}_\perp$ located on that triangle. To define such a region for a triangle, three planes are required, one for each edge. More specifically, if a node is located on the negative side of all three planes then that node is considered to belong exclusively to that triangle.

We check if a point $\mathbf{x}$ is located in a triangle region in the following way (Fig. 7.16):

1. For each vertex $\mathbf{v}i$ on the mesh, the vertex normal is computed as an angle weighted average of the normals of adjacent triangles:

$$\mathbf{n}_\Sigma = \frac{\sum \alpha_i \mathbf{n}_i}{\left|\sum \alpha_i \mathbf{n}_i\right|} \tag{7.30}$$

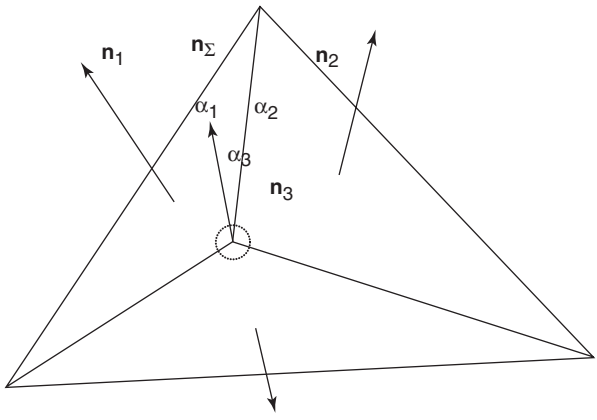

Fig. 7.16 Defining a vertex normal as an angle weighted average of the normals from adjacent triangles

2. For each edge $(\mathbf{v}_i, \mathbf{v}_j)$, with the associated vertex normals $(\mathbf{n}_i, \mathbf{n}_j)$ a separating plane is defined:

$$\mathbf{n}_s \cdot (\mathbf{x} - \mathbf{v}_i) = 0 \tag{7.31}$$

Where $\mathbf{n}_s$ is the separating plane normal and is computed as an edge bi-normal:

$$\mathbf{n}_s = \frac{1}{2}\left[(\mathbf{v}_j - \mathbf{v}_i) \times (\mathbf{n}_i + \mathbf{n}_j)\right] \tag{7.32}$$

3. A point $\mathbf{x}$ is considered to be located inside a region of a triangle $T = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ if it is located on the negative side of all three separating planes:

$$\begin{cases} \mathbf{n}_{s_1} \cdot (\mathbf{x} - \mathbf{v}_1) \le 0 \\ \mathbf{n}_{s_2} \cdot (\mathbf{x} - \mathbf{v}_2) \le 0 \\ \mathbf{n}_{s_3} \cdot (\mathbf{x} - \mathbf{v}_3) \le 0 \end{cases} \tag{7.33}$$

For any two adjacent, non-intersecting triangles, the regions defined by Eq. (7.33) will not intersect. If each GPU thread processes the nodes in separated regions then there will never be any concurrency hence the synchronization is no longer required. This drastically improves the GPU parallelism and performance.

7.3.3 Results

To test our implementation we considered a known CFD benchmark case consisting of a large brain aneurysm (Steinman et al. 2013). Figure 7.17 displays the mesh along with the voxelized surface.

The mesh contains 318,000 triangular elements and the size of the grid in which the surface is embedded is $171 \times 180 \times 142$. We performed the computations for this case using the CPU and GPU implementations for both the classic and the separating planes method. The hardware we used consists of an Intel i7 (8-cores) CPU and a GTX Titan Black GPU.

The execution times were:

- for the classic method on the CPU the execution time was 23.5 s and on the GPU it was 234 ms which gives a speedup of around 100 times. The GPU execution time does not contain the CPU-GPU memory copy as in an FSI simulation the memory copy should only be done once in the pre-processing stage.
- for the separating planes method, the GPU execution time was 21.4 ms. Compared to the current implementation that we use for FSI computations, the new GPU-accelerated one is around 1000 times faster. Using this approach, the performance of the FSI computations can be taken to an unprecedented level.

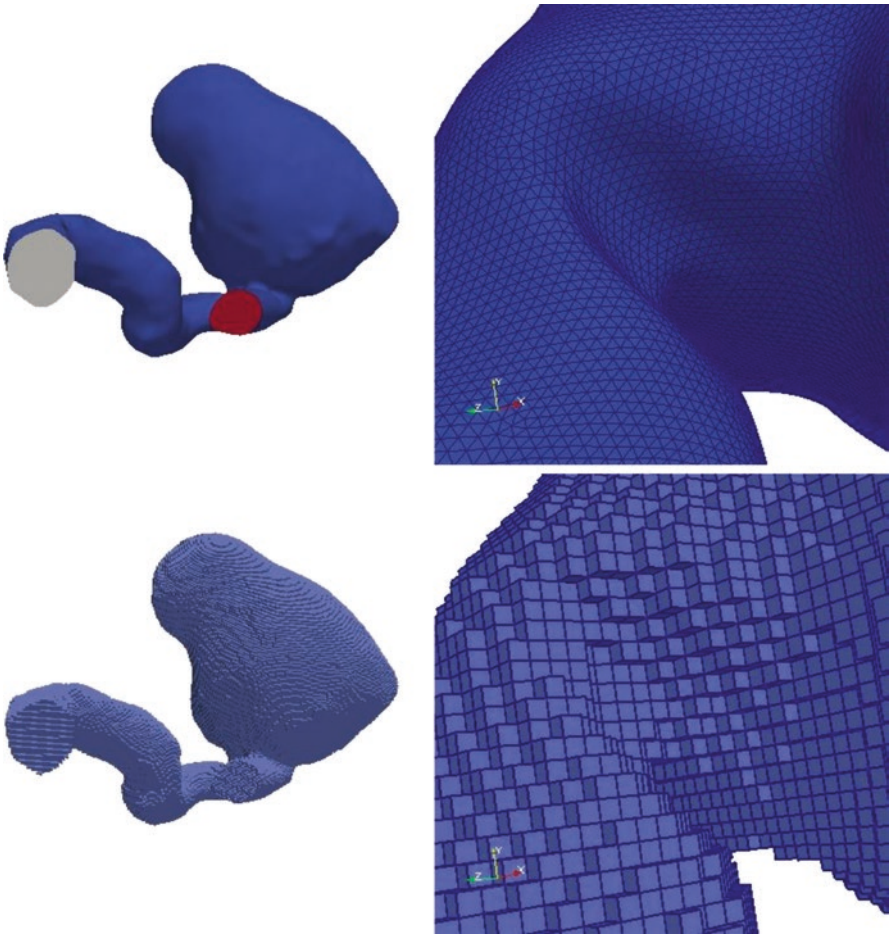

Fig. 7.17 Test case: a large brain aneurysm mesh of 318,000 triangular elements (up) and the voxelized surface (down)

7.4 GPU Accelerated Solution of Large Linear Systems of Equations Using the Preconditioned Conjugate Gradient Method

The focus for this section has been on the development of fast solutions for very large sparse linear systems of equations of the type $A \bullet x = b$, using parallel methods, where A is an N × N symmetric positive definite matrix. Such systems routinely appear when computing numerical solutions to PDEs, such as but not limited to the Finite Element Method. A widely-used iterative approach for solving such linear systems is the Conjugate Gradient (CG) method (Hestenes and Stiefel 1952).

In each iteration, the CG method performs a Sparse-Matrix Vector multiplication, a process that converges in at most N iterations to the exact solution. While current GPU technology excels in fast processing of a large number of parallel computational threads, its global memory size can still create a bottleneck in solving large linear systems. We have developed a methodology for overcoming this limitation using a streaming based algorithm. Parallel algorithms for iterative solutions to the above system have been proposed already, e.g. Ortega (1988), and using the CUDA framework (Verschoor and Jalba 2012). However, such solutions do not take into account RAM memory limitations on the size of the solution.

7.4.1 Single-GPU Solution

We propose a streaming based algorithm in order to overcome the global memory size limitation of the GPU, so as to be able to solve large systems arising in numerical solutions of various biomechanical PDEs. These may include but are not limited to fluid flow, bone or soft tissue deformation, etc. GPU cards have limited RAM memory (currently up to 12 GB), which limits the size of the system of equations (currently to around 12 million equations). To alleviate this limitation we introduce a streaming based solution whose core idea is to store the matrix A on the CPU RAM, and to transfer it slice by slice to the GPU during the matrix-vector multiplication step of the PCG method (Fig. 7.18). Our streaming based strategy can be used either in the context outlined in the next section, or in the more general context of iterative methods that need to handle during each iteration an operation involving data that exceeds the GPU memory.

When applying the PCG method, the majority of the memory required for the solution is occupied by matrix A and by the matrix used for the preconditioning (Saad 2003). To reduce the memory requirements, here we apply a Jacobi (diagonal) pre-conditioner which is stored as vector, while A is stored in a sparse matrix format (e.g. ELLPACK) (Bell and Garland 2008). The other operations of the PCG method are either vector-vector or scalar-vector operations. The seven vectors are stored throughout the entire execution on the GPU due to their limited memory size. Figure 7.18a displays the slicing strategy of A. To ensure coalesced memory accesses by the threads of the same warp, A is stored in column major order. The slicing however is performed on a row basis. To limit the number of copy operations to two for one slice, we still store data in column major order, but only at slice level (all values of one slice are stored in consecutive locations). To reduce the execution time, the memory transfer operation of one slice is overlapped with the processing of another slice. To implement the overlapping behavior, two memory slices are allocated on the GPU (marked 'A' and 'B' in Fig. 7.18a: while data is copied into one slice, the other one is processed.

The matrix-vector multiplication is performed by using two different streams: one for the memory transfer operations—host (CPU) to device (GPU)—and one for the processing of the slices—kernel execution (Fig. 7.18b). The operations of different streams are executed out of order and need to be synchronized. Therefore we use

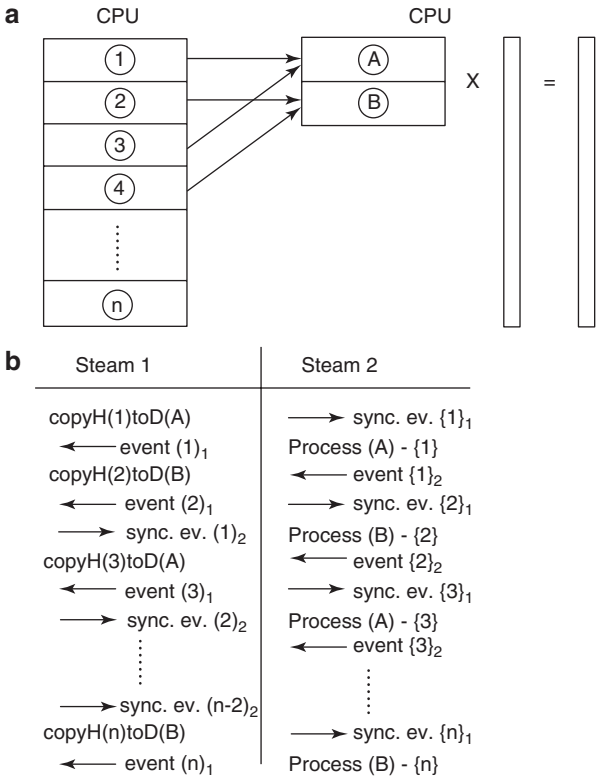

Fig. 7.18 (**a**) Slicing strategy for matrix A and successive memory transfer to the GPU; (**b**) Streaming-based execution of the matrix-vector multiplication

CUDA events: an event $(i)_1$ is recorded after copying slice i from host to device, while an event $(i)_2$ is recorded after processing slice i. The synchronization at events $(i)_1$ is used to enable the processing of one slice only after the corresponding memory transfer operation is finished. The synchronization at events $(i)_2$ is used to enable the overwriting of one slice only after it was processed. This ensures a correct synchronization between the streams, irrespective of the relative duration of the memory transfer and processing operations.

Next, we present specific results for linear systems of equations as they typically arise in solid mechanics applications. To test the method described abobe, we used four different FE models. For each model, a volume of cube was meshed with 8-node hexahedral elements, with each node having three DOFs (translation in x, y and z dimensions). The four linear systems were first solved with the commercial software ANSYS (Release 14.0.3, ANSYS, Inc) on a six-core processor (Intel (R) Xeon (R) E-5-2670. 2.60 GHz) with 256 GB of RAM. Next, we solved the systems with the streaming based GPU algorithm on an eight-core i7 processor, 3.4 GHz, with 8 GB of RAM and a NVIDIA Kepler GTX680 graphics card, with 2 GB of RAM. Table 7.7 compares the execution times of the CPU and GPU based algorithms. The speed-up varies between **44.2**× and **181.2**× for the largest system of equations.

Table 7.7 Comparison of CPU and GPU based algorithms

Config.	Nr. of equations	CPU–Ansys	Streaming based GPU algorithm			
		Exec. time (s)	Exec. time (s)	Nr. iter.	Number of slices in A	Speed-up
Test 1	2.260.713	3441	77.8	525	30	**44.2**
Test 2	4.102.893	21,334	175.9	647	30	**121.3**
Test 3	5.582.601	35,125	268.2	721	18	**131.0**
Test 4	7.057.911	66,046	363.1	785	44	**181.9**

7.4.2 Multi-GPU Solutions

The method described in the previous section was also used to evaluate multi-GPU based implementation of the preconditioned conjugate gradient method.

The strategy we implemented for running the PCG method on multiple GPUs is:

- One node is considered to be the master and performs all operations which do not involve the matrix-vector multiplication (initialization, copy operations, vector-vector operations, scalar-vector operations).
- All nodes, including the master node, perform the matrix-vector multiplication step:
  - Each GPU stores a section of matrix A, which includes multiple slices (data transfer to the various GPUs is performed during initialization). All sections are approximately equal.
  - At each iteration, before starting the matrix-vector multiplication, each GPU receives the vector values used during the multiplication.
  - Each GPU performs in parallel the matrix-vector multiplication for the section of the matrix which was assigned to it.
  - Each GPU sends the resulting vector section back to the master node.

The Message Passing Interface (MPI) is used for transferring data between the nodes. The implementation described in this section is motivated by two goals:

- Reduce the execution time of a single-GPU based implementation
- Handle cases when matrix A does not fit into the RAM memory of the CPU

Table 7.8 displays the results for a two-node configuration for the *Test 2* and *Test 3* configurations used in the previous section. As one can observe, if a 1 Gbit/s node-to-node transfer speed is used the execution time increases compared to the single-node implementation. This is given by the time required to transfer the vectors at each iteration. Conversely, if a 10 Gbit/s node-to-node transfer speed is used, the execution time decreases by a factor of around 1.5, which means that the time spent during data transfer is overcompensated by the time saved through the parallel computation of the matrix-vector product.

Table 7.8 Comparison of single-node and multi-node GPU based implementation of the PCG method.

Configuration	1 Node (s)	2 Nodes (1 Gbit/s) (s)	2 Nodes (10 Gbit/s)
Test 2	176	450	116 s (1.51×)
Test 3	268	704	179 s (1.50×)

Table 7.9 Comparison of single-node single-GPU and multi-GPU based implementation of the PCG method

Configuration	Slice distribution	Exec. time (s)	Speed-up	
			Measured	Theoretical
Titan (×16)	55	164.1 ± 0.87	–	–
Titan(×16), 680(×16)	35/20	120.0 ± 0.64	1.37	1.57
Titan(×16), 680(×8), 680(×8)	35/10/10	137.1 ± 0.69	1.20	1.57

The strategy described above was also used for a hardware configuration containing multiple GPUs in a single node. The major difference is that instead of using MPI to transfer data from one node to another, all data transfers are performed through the PCI Express bus. The specific hardware configuration used for testing is: E6989 Rampage IV Extreme Main Board which has 4 PCI slots (2 at ×16, 2 at ×8), 3 GPUs: 1× GTX Titan Black, 2× GTX680. Due to the different transfer speeds over the PCI bus, the ideal partitioning of slices to different GPUs depends on the PCI bus to which each GPU is physically connected. Hence, we implemented a methodology for automatically determining the number of slices for each GPU so as to obtain an execution time for the matrix-vector multiplication which requires approximately the same time on each GPU (so as to avoid idle times for the processors). Table 7.9 displays the results for three different configurations. One can observe that the multi-GPU implementation leads to smaller execution times. However, the measured speed-up is smaller than the theoretical value due to transfer of data between GPUs.

7.5 GPU Accelerated Random Forest Classification

Machine Learning algorithms have been proven to be useful in a variety of application domains (Zhang 2000). Herein we focus on one of the most common machine learning applications: classification. We study how to effectively implement a random forests (RF) algorithm for data classification on GPUs by evaluating the performance of the algorithm in terms of execution time, compared to the CPU-based version. The random forest consists of multiple decision trees which can be generated and evaluated independently and can classify large amounts of data, described by a large number of attributes. Therefore, the random forest algorithm is very well suited for a massively parallel approach (implemented on GPUs).

Random forest is an ensemble classifier consisting of decision trees that combines two sources of randomness to generate base decision trees: bootstrapping instances for each tree and considering a random subset of features at each node (Breiman 2001). It is a supervised learning method: the training data consists of a set of training examples; each example is a pair consisting of an input object (typically a vector of features) and the corresponding desired output value. A supervised learning algorithm analyzes the training data and produces a model, which is then used to predict the output for new examples.

During the learning phase, the data that has reached a given leaf is used to model the posterior distribution. During the test phase, these posterior distributions enable the prediction for new unseen observations reaching a given leaf.

Because the training phase is done offline, the time required by this phase is not critical. Therefore, we only focus on the acceleration of random forest classification since in most of the cases this phase is done online and the execution time may be critical. The algorithm behind the testing phase is based on the fact that each internal node of a tree contains a simple test that splits the space of data to be classified and each leaf contains an estimate based on training data of the posterior distribution over the classes. The input data transformed into a feature vector is classified by propagating the information through all the trees and performing an elementary test at each node that directs it to one of the child nodes. Each decision node contains a test function that compares a feature response with a threshold to generate a binary decision. Once the sample reaches the leaf in each three in the forest, the posterior probabilities are combined (voting or averaging) to compute the final posterior probability. Traversing a large number of decision trees sequentially is ineffective when they are built independently of each other. Since the trees in the forest are independently built and the only interaction is the final counting of the votes, the voting part (classification) of the RF algorithm can be efficiently parallelized (Grahn et al. 2011).

The first step of the GPU based implementation of the RF classifier is to load all decision tree data structures of a RF into the GPU memory. Prediction is performed in a loop over all pixels of the input image. We determine the feature response for each pixel which will become the input vector for each tree. Each decision tree is then traversed in parallel to retrieve the probability distribution over all classes for the given pixel. The probability distributions from every tree in the forest are averaged and, finally, the result is copied back into the CPU memory.

To accelerate prediction on the GPU, we use multiple threads to process each image and multiple threads to process the RF trees in parallel. After images are loaded, we calculate the integral image in a pre-processing step. Calculating image integrals is expensive with respect to processing time. We accelerate it by calculating the integral for each of the five image channels in parallel with separate threads on the GPU. Prediction time depends on the complexity of the features but scales linearly with the number of trees, depth of the trees and the number of pixels in the input image. To take advantage of the massively parallel computing power of the GPU, instead of pre-computing all values for all possible features, we sampled the feature space at runtime and calculated the feature responses on demand.

```
Algorithm- Random Forest prediction on the GPU
1. for all trees in the forest do
2.     while curNode has valid children do
3.         if children_found then
4.          float right = evaluate_Harr_feature(boxCenter, curNode);
5.            if (right < 0)
6.              curNode <- leftChildPosition
7.            else
8.              curNode <- rightChildPosition
9.          else if leaf_node_reached then
10.         probability + = leaf_Node_Histogram
11.         total ++
12.         end if
13.    end while
14. end for
15. return probability / total
```

Fig. 7.19 Algorithm for the binary decision tree evaluation

Our strategy for storing the RF decision trees involves the mapping of the data structure describing the RF to a 2-D texture array which is stored in the GPU texture memory. These texture arrays are read only, and, since they are cached, this improves the performance of reading operations (Grahn et al. 2011). The data associated with a tree is laid out in a four-component float texture, whereas the data of each node is stored on three separate columns in each channel of the texture array. We store the data of each node of a tree in the forest in sequential horizontal positions and different trees on separate rows. Data stored in the texture memory contains the position of the left and right child nodes, threshold values and all feature parameters required to evaluate the test for a node. If a node cached in the texture memory is a leaf node, then we add the probability distribution learned during training and the index of the leaf. To navigate through the tree during the evaluation, we use the tex2D function which performs a texture lookup in a given 2-D sampler based on 2-D node coordinates and channel information (Fig. 7.19). Our strategy involves launching a kernel which evaluates the probability of the random forest with the number of threads equal to the number of candidates. As the number of feature candidates can exceed the maximum number of threads per block with a maximum of 1024, 1536 and 2048 for compute capability 1.2/1.3, 2.x and 3.x, respectively, several thread blocks are launched.

In the following we present results for a spine structure detection application, whereas the 3D input image was obtained from microCT (computed tomography) and where the random forest classifier is used to detect lesions. After the system was trained, tests were run with three different implementations of the classification algorithm. The implementation was tested with two available data sets and performance benchmarks for our implementation have been compared with two CPU based implementations. The experimental results indicate that our GPU-based implementation of the Random Forest algorithm outperforms the two CPU based algorithms (CPU single-core and CPU multi-core).

To test the method for the bone lesion detection, we used a Random Forest detector consisting of 100 trees, 27K normalized candidates (voxels) for each vertebra and 12K features. The execution configuration of the classification kernel specifies

Table 7.10 Comparison of execution times of CPU and GPU based algorithms for the RF classifier

Implementation	Patient 1 (s)	Patient 2 (s)
CPU Multi-core	27.62 ± 0.39	48.60 ± 0.41
CPU Multi-core + GPU	17.85 ± 0.32	22.57 ± 0.34

Table 7.11 Total execution time for the detection system

Implementation	Patient 1 (s)	Patient 2 (s)
CPU Single-core	73.01 ± 0.58	182.33 ± 0.64
CPU Multi-core	10.19 ± 0.15	27.00 ± 0.25
CPU Multi-core + GPU	0.421 ± 0.02	0.979 ± 0.03

that the number of threads is equal to the number of candidates (27K), each block contains 128 threads (27K/128 blocks are used) and each thread traverses a candidate through all trees of the random forest.

Prediction time was compared on a machine equipped with Intel Core i7 CPU and a NVIDIA GPU GeForce GTX Titan Black. Table 7.10 compares the execution times of the CPU and GPU based algorithms. The speed-up varies between **170**× and **190**× compared to the single-core implementation and between **24**× and **28**×. Table 7.11 displays the total execution time which includes various other non-parallelized steps like the loading of the input image, etc. The RF based position detector classifier occupies the vast majority of the total execution time (~60%).

Acknowledgments The authors would like to thank Puneet Sharma, Ali Kamen and Dorin Comaniciu for their input.

References

Alastruey J, Khir A, Matthys K, Segers P, Sherwin S, Verdonck P, Parker K, Peiro J (2011) Pulse wave propagation in a model human arterial network: assessment of 1-D visco-elastic simulations against in vitro measurements. J Biomech 44:2250–2258. doi:10.1016/j.jbiomech.2011.05.041

Alerstam E, Svensson T, Andersson-Engels S (2008) Parallel computing with graphics processing units for high speed Monte Carlo simulation of photon migration (PDF). J Biomed Opt 13:060504. doi:10.1117/1.3041496

Bell N, Garland M (2008) Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004

Bessems D (2008) On the propagation of pressure and flow waves through the patient-specific arterial system. PhD Thesis, Technical University of Eindhoven, Netherlands

Bessems D, Giannopapa C, Rutten M, van de Vosse F (2008) Experimental validation of a time-domain-based wave propagation model of blood flow in viscoelastic vessels. J Biomech 41:284–291. doi:10.1016/j.jbiomech.2007.09.014

Breiman L (2001) Random forests. Mach Learn 45:5–32

Chen G, Li G, Pei S, Wu B (2009) High performance computing via a GPU. In: Proceedings of the first international conference on information science and engineering, Shanghai, China, pp 238–241

Cococcioni M, Grasso R, Rixen M (2011) Rapid prototyping of high performance fuzzy computing applications using high level GPU programming for maritime operations support.

In: Proceedings of the 2011 IEEE symposium on computational intelligence for security and defense applications (CISDA), Paris, 11–15 April 2011
Courant R, Friedrichs K, Lewy H (1928) Uber die partiellen Differenzengleichungen der mathematischen Physik. Mathematishe Annalen 100:32–74
Cousins W, Gremaud PA, Tartakovsky DM (2013) A new physiological boundary condition for hemodynamics. SIAM J Appl Math 73(3):1203–1223
Formaggia L, Lamponi D, Quarteroni A (2003) One dimensional models for blood flow in arteries. J Eng Math 47:251–276. doi:10.1023/B:ENGI.0000007980.01347.29
Formaggia L, Lamponi D, Tuveri M, Veneziani A (2006) Numerical modeling of 1D arterial networks coupled with a lumped parameters description of the heart. Comput Methods Biomech Biomed Engin 9:273–288. doi:10.1080/10255840600857767
Fung Y (1993) Biomechanics: mechanical properties of living tissues. Springer, New York
Garcia V, Debreuve E, Barlaud M (2008) Fast k-nearest neighbor search using GPU. In: Proceedings of the CVPR workshop on computer vision on GPU, Anchorage, Alaska, USA
Grahn H, Lavesson N, Lapajne MH, Slat D (2011) CudaRF: a CUDA-based implementation of random forests. In: Siegel HJ, El-Kadi A (ed) AICCSA, IEEE, pp 95–101
Habchia C, Russeil S, Bougeard D, Hariona J-L, Lemenand T, Ghanem A, Della Valle D, Peerhossaini H (2003) Partitioned solver for strongly coupled fluid–structure interaction. Comput Fluids 71:306–319. doi:10.1016/j.compfluid.2012.11.004
Hasan Khondker S, Chatterjee A, Radhakrishnan S, Antonio JK (2014) Performance prediction model and analysis for compute-intensive tasks on GPUs. The 11th IFIP international conference on network and parallel computing (NPC-2014), Ilan, Taiwan, September 2014, Lecture Notes in Computer Science (LNCS), pp 612–617. ISBN:978-3-662-44917-2
Hestenes M, Stiefel E (1952) Methods of conjugate gradients for solving linear systems. J Res Natl Bur Stand 49:409–436
Itu L, Sharma P, Mihalef V, Kamen A, Suciu C, Comaniciu D (2012a) A patient-specific reduced-order model for coronary circulation. IEEE international symposium on biomedical imaging, Barcelona, Spain, pp 832–835. doi:10.1109/ISBI.2012.6235677
Itu LM, Sharma P, Kamen A, Suciu C, Moldoveanu F, Postelnicu A (2012b) GPU accelerated simulation of the human arterial circulation. In: Proceedings of the 13th international conference on optimization of electrical and electronic equipment, Brasov, Romania, pp 1478–1485. doi:10.1109/OPTIM.2012.6231764
Itu L, Sharma P, Ralovich K, Mihalef V, Ionasec R, Everett A, Ringel R, Kamen A, Comaniciu D (2013a) Non-invasive hemodynamic assessment of aortic coarctation: validation with in vivo measurements. Ann Biomed Eng 41:669–681. doi:10.1007/s10439-012-0715-0
Itu LM, Sharma P, Kamen A, Suciu C, Comaniciu D (2013b) Graphics processing unit accelerated one-dimensional blood flow computation in the human arterial tree. Int J Numer Methods Biomed Eng 29(12):1428–1455, ISSN:2040-7947
Janßen CF, Koliha N, Rung T (2015) A fast and rigorously parallel surface voxelization technique for GPGPU-accelerated CFD simulations. Commun Comput Phys 17(05):1246–1270
Jiang B, Struthers A, Sun Z, Feng Z, Zhao X, Zhao K, Dai W, Zhou X, Berens ME, Zhang L (2011) Employing graphics processing unit technology, alternating direction implicit method and domain decomposition to speed up the numerical diffusion solver for the biomedical engineering research. Int J Numer Methods Biomed Eng 27:1829–1849. doi:10.1002/cnm.1444
Jordan HF, Alaghband G (2003) Fundamentals of parallel processing. Pearson Education, Upper Saddle River, NJ
Kirk D, Hwu WM (2010) Programming massively parallel processors: a hands-on approach. Elsevier, London
Klages P, Bandura K, Denman N, Recnik A, Sievers J, Vanderlinde K (2015) GPU kernels for high-speed 4-bit astrophysical data processing. In: IEEE 26th international conference on application-specific systems, architectures and processors (ASAP)
Kumar R, Quateroni A, Formaggia L, Lamponi D (2003) On parallel computation of blood flow in human arterial network based on 1-D modelling. Computing 71:321–351. doi:10.1007/s00607-003-0025-3

Malecha Z, Mirosław L, Tomczak T, Koza Z, Matyka M, Tarnawski W, Szczerba D (2011) GPU-based simulation of 3D blood flow in abdominal aorta using OpenFOAM. Arch Mech 63:137–161

Malossi C, Blanco P, Deparis S (2012) A two-level time step technique for the partitioned solution of one-dimensional arterial networks. Comput Methods Appl Mech Eng 237:212–226. doi:10.1016/j.cma.2012.05.017

Manavski SA, Valle G (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9(suppl 2):S10. doi:10.1186/1471-2105-9-s2-s10. http://www.biomedcentral.com/1471-2105/9/S2/S10

Mittal S, Vetter J (2015) A survey of CPU-GPU heterogeneous computing techniques. ACM Comput Surv. doi:10.1145/2788396

Mynard JP, Nithiarasu P (2008) A 1D arterial blood flow model incorporating ventricular pressure, aortic valve and regional coronary flow using the locally conservative Galerkin (LCG) method. Commun Numer Methods Eng 24:367–417. doi:10.1002/cnm.1117

Mynard JP, Davidson MR, Penny DJ, Smolich JJ (2012a) A simple, versatile valve model for use in lumped parameter and one-dimensional cardiovascular models. Int J Numer Methods Biomed Eng 28:626–641. doi:10.1002/cnm.1466

Mynard JP, Penny DJ, Davidson MR, Smolich JJ (2012b) The reservoir-wave paradigm introduces error into arterial wave analysis: a computer modelling and in vivo study. J Hypertens 30:734–743. doi:10.1097/HJH.0b013e32834f9793

Nita C, Itu LM, Suciu C (2013) GPU accelerated blood flow computation using the lattice Boltzmann Method. In: 17th IEEE high performance extreme computing conference, Waltham, MA, USA

Nita C et al (2015) GPU-accelerated model for fast, three-dimensional fluid-structure interaction computations. Engineering in Medicine and Biology Society (EMBC), 2015 37th annual international conference of the IEEE. IEEE

Olufsen M, Peskin C, Kim WY, Pedersen EM, Nadim A, Larsen J (2000) Numerical simulation and experimental validation of blood flow in arteries with structured-tree outflow conditions. Ann Biomed Eng 28:1281–1299. doi:10.1114/1.1326031

Ortega J (1988) Introduction to parallel and vector solution of linear systems. Plenum Press, New York

Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) GPU computing. Proc IEEE 96:879–884

Passerini T (2009) Computational hemodynamics of the cerebral circulation: multiscale modeling from the circle of Willis to cerebral aneurysms. PhD Thesis, Politecnico di Milano, Italy

Raghu R, Vignon-Clementel I, Figueroa CA, Taylor CA (2011) Comparative study of viscoelastic arterial wall models in nonlinear one-dimensional finite element simulations of blood flow. J Biomech Eng 133:081003. doi:10.1115/1.4004532

Rahimian A, Lashuk I, Veerapaneni S, Chandramowlishwaran A, Malhotra D, Moon L, Sampath R, Shringarpure A, Vetter J, Vuduc R, Zorin D, Biros G (2010) Petascale direct numerical simulation of blood flow on 200K cores and heterogeneous architectures. In: Proceedings of the ACM/IEEE international conference for high performance computing, networking, storage and analysis, New Orleans, USA, pp 1–11

Reymond P, Bohraus Y, Perren F, Lazeyras F, Stergiopulos N (2011) Validation of a patient-specific one-dimensional model of the systemic arterial tree. Am J Physiol Heart Circ Physiol 301:1173–1182. doi:10.1152/ajpheart.00821.2010

Ryoo S, Rodrigues CI, Stone SS, Stratton JA, Ueng S-Z, Baghsorkhi SS, Wen-mei WH (2008) Program optimization carving for gpu computing. J Parallel Distrib Comput 68(10):1389–1401

Saad Y (ed) (2003) Iterative methods for sparse linear systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia

Sato D, Xie Y, Weiss JN, Qu Z, Garfinkel A, Sanderson AR (2009) Acceleration of cardiac tissue simulation with graphic processing units. Med Biol Eng Comput 47(9):1011–1015

Schalkwijk J, Harmen JJ, Jonker A, Siebesma P, Van Meijgaard E (2015) Weather forecasting using GPU-based Large-Eddy simulations. Bull Am Meteorol Soc 96:715–723

Schatz MC, Trapnell C, Delcher AL, Varshney A (2007) High-throughput sequence alignment using Graphics Processing Units. BMC Bioinformatics 8:474

Sengupta S, Harris M, Garland M (2008) Efficient parallel scan algorithms for GPUs. NVIDIA Technical Report NVR-2008-003

Senzaki H, Chen CH, Kass DA (1996) Valvular heart disease/heart failure/hypertension: single-beat estimation of end-systolic pressure-volume relation in humans: a new method with the potential for noninvasive application. Circulation 94:2497–2506

Shams R, Sadeghi P, Kennedy RA, Hartley RI (2010) A survey of medical image registration on multicore and the GPU. IEEE Signal Process Mag 27(2):50–60

Shen W, Wei D, Xu W, Zhu X, Yuan S (2009) GPU-based parallelization for computer simulation of electrocardiogram, CIT '09. In: Ninth IEEE international conference on computer and information technology, October, Xiamen, China

Steinman DA et al (2013) Variability of computational fluid dynamics solutions for pressure and flow in a giant aneurysm: the ASME 2012 Summer Bioengineering Conference CFD Challenge. J Biomech Eng 135.2:021016

Stergiopulos N, Young DF, Rogge TR (1992) Computer simulation of arterial flow with applications to arterial and aortic stenosis. J Biomech 25:1477–1488. doi:10.1016/0021-9290(92)90060-E

Tanno I, Morinishi K, Satofuka N, Watanabe Y (2011) Calculation by artificial compressibility method and virtual flux method on GPU. Comput Fluids 45:162–167. doi:10.1016/j.compfluid.2011.02.005

Vardoulis O, Papaioannou T, Stergiopulos N (2012) On the estimation of total arterial compliance from aortic pulse wave velocity. Ann Biomed Eng 40:2619–2626. doi:10.1007/s10439-012-0600-x

Verschoor M, Jalba A (2012) Analysis and performance estimation of the conjugate gradient method on multiple GPUs. Parallel Comput 38:552–575

Westerhof N, Elzinga G, Sipkema P (1971) An artificial arterial system for pumping hearts. J Appl Physiol 31:776–781

Willemet M, Lacroix V, Marchandise E (2011) Inlet boundary conditions for blood flow simulations in truncated arterial networks. J Biomech 44:897–903. doi:10.1016/j.jbiomech.2010.11.036

Yu R, Zhang S, Chiang P, Cai Y, Zheng J 2010 Real-time and realistic simulation for cardiac intervention with GPU. In: Second international conference on computer modeling and simulation, Sanya, China, pp 68–76

Zaspel P, Griebel M (2013) Solving incompressible two-phase flows on multi-GPU clusters. Comput Fluids. doi:10.1016/j.compfluid.2012.01.021

Zhang D (2000). Applying machine learning algorithms in software development. In: Proceedings of Monterey workshop on modeling software system structures, Santa Margherita Ligure, Italy, pp 275–285

Zhang Y, Cohen J, Owens JD (2010) Fast tridiagonal solvers on the GPU. In: Proceedings of the 15th ACM SIGPLAN symposium on principles and practice of parallel programming, Bangalore, India, pp 127–136

Zou C, Xia C, Zhao G (2009) Numerical parallel processing based on GPU with CUDA architecture. In: International conference on wireless networks and information systems, WNIS'09, IEEE, pp 93–96