GPU-Accelerated Simulation of Elastic Wave Propagation

Kristian Kadlubiak
Centre of Excellence IT4Innovations,
Faculty of Information Technology,
Brno University of Technology
Brno, Czech Republic
ikadlubiak@fit.vutbr.cz

Jiri Jaros
Centre of Excellence IT4Innovations,
Faculty of Information Technology,
Brno University of Technology
Brno, Czech Republic
jarosjir@fit.vutbr.cz

Bredly E. Treeby
Department of Medical Physics and
Biomedical Engineering
University College London
London, United Kingdom
b.treeby@ucl.ac.uk

***Abstract*—Modeling of ultrasound waves propagation in hard biological materials such as bones and skull has a rapidly growing area of applications, e.g. brain cancer treatment planing, deep brain neurostimulation and neuromodulation, and opening blood brain barriers. Recently, we have developed a novel numerical model of elastic wave propagation based on the Kelvin-Voigt model accounting for linear elastic wave proration in heterogeneous absorption media. Although, the model offers unprecedented fidelity, its computational requirements have been prohibitive for realistic simulations. This paper presents an optimized version of the simulation model accelerated by the Nvidia CUDA language and deployed on the best GPUs including the Nvidia P100 accelerators present in the Piz Daint supercomputer. The native CUDA code reaches a speed-up of 5.4 when compared to the Matlab prototype accelerated by the Parallel Computing Toolbox running on the same GPU. Such reduction in computation time enables computation of large-scale treatment plans in terms of hours.**

Keywords*—*Ultrasound simulations; Elastic model; Pseudospectral methods; k-Wave; CUDA; GPU

I. Introduction

Simulation of elastic wave propagation has many applications in ultrasonics [1] spreading from the assessment of bone quality to non-destructive testing [2], [3]. In biomedical ultrasound in particular, elastic wave propagation models have been used to investigate a propagation of ultrasound in the skull and brain.

The High Intensity Focused Ultrasound (HIFU) is a method where several rays of ultrasound are focused in such a way they form a single focal point [4], [5]. Although each ray passes through a tissue with only little effect, the energy of all beams combined together in the focal point leads to a temperature rise and/or mechanical ablation.

HIFU has a tremendous potential to improve the treatment of certain types of brain cancer. As this modality is non-invasive and accurate, it may be able to ablate only targeted tissue (e.g. a tumor) while sparring healthy adjacent tissue [6]. This is especially critical in the brain where any damage to healthy tissue can result in significant loss of function. In addition, focused ultrasound has the potential to reduce the risk of infection and bleeding, lower procedural morbidity by not opening the skull, and avoid the toxicity of radiation [7]. Nevertheless, HIFU can also be used in treatment of many different diseases due to variable intensity. For example, lower-intensity HIFU can be used to destruct blood cloths in arteries. Another use of low-intensity HIFU is an ultrasonic drug delivery [8]. In this process nanoparticles carrying a therapeutic agent are injected into the bloodstream and then exposed to ultrasound in a targeted area, which results in release of the agent only at this location. Moreover, HIFU can also be used to temporarily open the blood-brain barrier which only increases the effectiveness of the drug delivery [9].

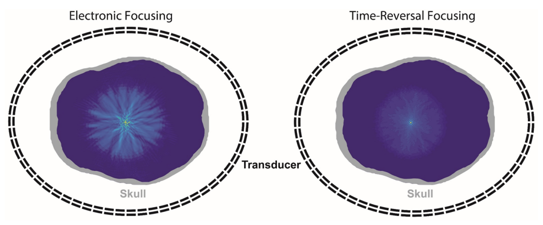

Fig. 1. Comparison of electronic and simulation based time-reversal focusing where the wave is first propagated from the desired focus in a time reversal manner, recorded in the transducer, corrected for aberrations and send back into the brain [10].

The key requirement of an effective HIFU treatment is the precise focus placement and appropriate energy delivery. This requirement can be satisfied by accurate simulation of the elastic wave propagation and techniques such as time-reversal focusing [10], see Fig. 1. Originally, numerical model of the simulation was implemented in Matlab as a part of the open-source k-Wave toolbox[1]. However, the computational requirements of the model did not allow for computation of realistic simulation cases. Therefore, it has been decided to create a native implementation in the CUDA and C++ languages to accelerate the simulation by using modern graphics processing units (GPUs).

[1]http://www.k-wave.org

A. Related work

There are several articles about simulation of wave propagation accelerated by GPUs. In [11] and [12] a finite element method and a spectral element method were used respectively. The main difference between the mentioned implementations and our work is the numerical method used to solve the elastic wave equation. The greatest advantage of finite element method is its straightforward parallelization. On the other hand, it requires very fine computation grids compared to other approaches. The spectral element method reduces the amount of grid points required, but for frequencies and domain sizes of our interest, the memory requirements exceeds the GPU's on-board memory capacity. Therefore, the model based on a pseudo-spectral method with sampling requirements approaching Nyquist theorem is used.

II. Numerical Model

When an acoustic wave passes through a compressible medium, it causes dynamic fluctuations in the acoustic pressure, density, and particle velocity. These changes can be described by a series of coupled first-order partial differential equations based on the conservation of mass, momentum, and energy within the medium [13]. For example, in the classical case of a small amplitude acoustic wave propagating through a homogeneous and lossless fluid medium, the first-order equations are given by [14]:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\frac{1}{\rho_0}\nabla p \\ \frac{\partial \rho}{\partial t} &= -\rho_0 \nabla \cdot u \\ p &= c_0^2 \rho \end{aligned} \tag{1}$$

where u represents acoustic particle velocity, p acoustic pressure and ρ acoustic density in medium, which all exhibit a time-dependent behavior. Contrary, ρ_0 and c_0 representing acoustic density of equilibrium and isotropic sound speed are properties of the medium and thus constant. Eq. (1) assumes the medium is fluid, lossless, quiescent (there is no net flow and the other ambient parameters do not change with time), and isotropic (the material parameters do not depend on the direction the wave is traveling). However, this is insufficient for simulation of ultrasound propagation in hard tissues such as bones.

Therefore, we adopted an accurate elastic wave model presented in [15]. The numerical model is based on several fundamental equations for studying lossy wave propagation which are derived from the Hook's law. Provided Kelvin-Voigt model is used to simulate viscoelasticity [16], resulting equation can be written using Einstein summation as follows

$$\sigma_{ij} = \lambda \delta_{ij} \epsilon_{kk} + 2\mu \epsilon_{ij} + \chi \delta \frac{\partial}{\partial t} \epsilon_{kk} + 2\eta \frac{\partial}{\partial t} \epsilon_{ij} \quad , \tag{2}$$

where σ is stress tensor, ϵ is dimensionless strain tensor, λ and μ are Lame parameters. Here μ is ratio of shear stress to shear strain. Coefficients χ and η represent compressional and shear viscosity respectively. The Lame parameter are related to compressional and shear sound speed of medium by

$$\mu = c_s^2 \rho_0, \quad \lambda + 2\mu = c_p^2 \rho_0, \tag{3}$$

where ρ_0 is mass density. If relation between strain and particle displacement u_i

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{4}$$

is used, then Eq. (2) can be re-written as function of a particle velocity v_i, where $v_i = \partial u_i / \partial t$

$$\begin{aligned} \frac{\partial \sigma_{ij}}{\partial t} =& \lambda \delta_{ij} \frac{\partial v_k}{\partial x_k} + \mu \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) \\ &+ \chi \delta_{ij} \frac{\partial^2 v_k}{\partial x_k \partial t} + \eta \left(\frac{\partial^2 v_i}{\partial x_j \partial t} + \frac{\partial^2 v_j}{\partial x_i \partial t}\right). \end{aligned} \tag{5}$$

To be able to simulate wave propagation in elastic medium, Eq. (5) is combined with equation representing momentum preservation. The equation is a function of stress and particle velocity and it is given by the relation

$$\frac{\partial v_i}{\partial t} = \frac{1}{\rho_0} \frac{\partial \sigma_{ij}}{\partial x_j}. \tag{6}$$

Eqs. (5) and (6) are coupled first-order partial differential equations which model pressure waves propagation in isotropic viscoelastic medium. A computationally efficient simulation model can be created using these equations and Fourier pseudospectral method [17]. This method uses Fourier collocation spectral method [18] to calculate spatial derivatives and a finite-difference method to integrate in time domain. A single simulation step is composed of several operations (for the sake of simplicity only x dimension operations are listed).

First, spatial gradients of stress field is calculated using the Fourier collocation spectral method

$$\partial_x \sigma_{xx}^- = \mathcal{F}_x^{-1}\left\{ i k_x e^{+ik_x \Delta x/2} \mathcal{F}_x \left\{\sigma_{xx}^-\right\}\right\}. \tag{7}$$

Here, $\mathcal{F}_x\{\}$ and $\mathcal{F}_x^{-1}\{\}$ are 1D forward and inverse Fourier transforms in the x dimension, i is the imaginary unit, k_x is a discrete set of wavenumber in the x dimension, and Δx represents the Cartesian grid spacing. In order to achieve higher precision, variables are stored in staggered grids to avoid odd-even decoupling between the pressure and velocity. Odd-even decoupling is a discretization error that can occur on collocated grids and which leads to checkerboard patterns in the solutions [19]. The principles of staggered grid is shown in Fig. 2. Therefore the exponential terms represents spatial shift operators which translate the output by half the grid point.

Next, the particle velocity is updated using a leapfrog time stepping scheme with a time step of Δt

$$v_x^+ = v_x^- + \frac{\Delta t}{\rho_0}\left(\partial_x \sigma_{xx}^- + \partial_y \sigma_{xy}^-\right), \tag{8}$$

where superscripts $-$ and $+$ denote the value at the current and next time step, respectively. Leapfrog method is numerical

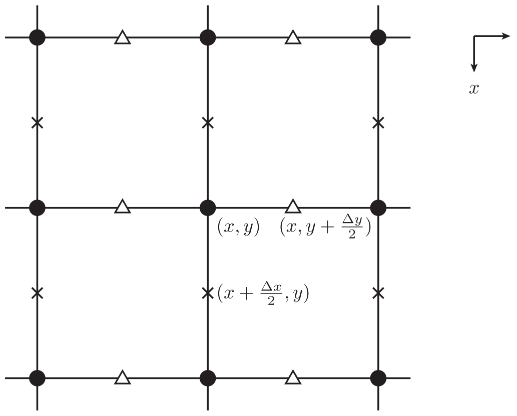

Fig. 2. Schematic showing the principles of using a staggered spatial grid in 2D. Here $\partial_x \sigma_{xx}$ is evaluated at grid points staggered in the x-direction (crosses), while $\partial_x \sigma_{yy}$ evaluated at grid points staggered in the y-direction (triangles) .

integration of the second-order differential equations where $f''(x)$ is calculated at $k\Delta t, k \in \{0, 1, \ldots, n\}$ and $f'(x)$ is evaluated at $k\Delta t + \frac{\Delta t}{2}, k \in \{0, 1, \ldots, n\}$.

Afterwards, the spatial gradients of the updated particle velocity is calculated again using the Fourier collocation spectral method

$$\partial_x v_x^+ = \mathcal{F}_x^{-1} \left\{ i k_x e^{-i k_x \Delta x/2} \mathcal{F}_x \left\{ v_x^+ \right\} \right\}. \tag{9}$$

Consequently, the spatial gradients of the time derivative of the particle velocity are calculated using Eq. (6)

$$\partial_x \partial_t v_x^- = \mathcal{F}_x^{-1} \left\{ i k_x e^{-i k_x \Delta x/2} \mathcal{F}_x \left\{ \left(\partial_x \sigma_{xx}^- + \partial_y \sigma_{xy}^- \right) / \rho_0 \right\} \right\}. \tag{10}$$

Finally, the stress field is updated using a finite-difference time scheme

$$\begin{aligned} \sigma_{xx}^+ = \sigma_{xx}^- + \lambda \Delta t \left(\partial_x v_x^+ + \partial_y v_y^+ \right) + \mu \Delta t \left(2 \partial_x v_x^+ \right) + \\ \chi \Delta t \left(\partial_x \partial_t v_x^- + \partial_y \partial_t v_y^- \right) + \eta \Delta t \left(2 \partial_x \partial_t v_x^- \right). \end{aligned} \tag{11}$$

The Fourier pseudospectral method has a significant benefit in the reduction of the number of grid points needed per wavelength while preserving the same accuracy as local finite difference methods. However, the use of the FFT to calculate spatial gradients implies the simulation domain is periodic. This causes waves leaving one side of the domain to reappear at the opposite side. This behavior can be largely eliminated by the use of a Perfectly Matched Layer (PML) [19], [20], a thin absorbing layer that encloses the computational domain and cause anisotropic absorption at the domain edges.

III. Implementation

The simulation of elastic wave propagation takes place in a discrete grid with a size and spacing derived from the physical dimensions of the examined body region and maximal frequency appearing in the domain. The time step is then derived from the minimum sound speed in the medium and the grid spacing. Apart from basic acoustic quantities (stress and velocity), the model requires the medium properties, usually estimated from the tissue scans acquired by conventional methods (e.g. CT/MR scans).

With all parameters specified, the model is able to solve the initial value problem where values of stress and velocity serve as initial conditions. Alternatively, the simulation code allows to specify stress or velocity sources operating in an additive or Dirichlet mode. The shape and placement of the source within the grid is defined by a source mask. The signal of the source is defined as a time-varying sequence of values, either common or distinct at source grid points.

Analogously to the source mask, it is also possible to define a sensor mask where desired quantities are sampled, further processed and stored in the output file in the HDF5 file format[2].

A. Nvidia CUDA library

The abbreviation CUDA[3] stands for Compute Unified Device Architecture and refers to Nvida's concept of GPU being composed of unified units. These units can be programmed to serve any purpose in the rendering pipeline according to specific needs of certain applications. This concept can also be exploited in the General Purpose computing on GPU (GPGPU), which uses graphics cards to compute generic algorithms. For this purpose, Nvidia develops and releases CUDA SDK to ease programmers writing and executing GPU applications. Besides that, it also offers a set of fine-tuned libraries implementing typical parallel algorithms such as Fast Fourier Transform (FFT) or dense algebra operations (e.g. parallel matrix multiplication).

Since GPUs are massively parallel by nature, they require the computation to be spread among millions of lightweight threads executed independently. Fortunately, the simulation code satisfy this condition by using the accelerated CUDA FFT library to perform batches of 1D FFTs, and a number of custom CUDA kernels processing the supportive operations mostly composed of element-wise matrix-matrix and matrix-vector operations. Since there are usually no dependencies between grid points, these operation can be distributed among a various number of CUDA thread blocks and threads.

B. Matlab implementation

Listing 1 shows the principle of the Matlab implementation on the calculation of the stress gradient in a 3D space outlined in Eq. (7). The complete Matlab implementation can be downloaded as part of the k-Wave toolbox.

```
%Calculate spatial gredients of stress tensor in next time step
dsxxdx = real(ifft(bsxfun(@times, ddx_k_p, fft(sxx_x + sxx_y + sxx_z, 1)), 1));
dsyydy = real(ifft(bsxfun(@times, ddy_k_p, fft(syy_x + syy_y + syy_z, 2)), 2));
dszzdz = real(ifft(bsxfun(@times, ddz_k_p, fft(szz_x + szz_y + szz_z, 3)), 3));
dsxydx = real(ifft(bsxfun(@times, ddx_k_n, fft(sxy_x + sxy_y, 1)), 1));
dsxydy = real(ifft(bsxfun(@times, ddy_k_n, fft(sxy_x + sxy_y, 2)), 2));
dsxzdx = real(ifft(bsxfun(@times, ddx_k_n, fft(sxz_x + sxz_z, 1)), 1));
dsxzdz = real(ifft(bsxfun(@times, ddz_k_n, fft(sxz_x + sxz_z, 3)), 3));
dsyzdy = real(ifft(bsxfun(@times, ddy_k_n, fft(syz_y + syz_z, 2)), 2));
dsyzdz = real(ifft(bsxfun(@times, ddz_k_n, fft(syz_y + syz_z, 3)), 3));
```

Listing 1. Calculation of the stress gradient written in Matlab in 3D space.

[2]https://www.hdfgroup.org/

[3]https://developer.nvidia.com/cuda-zone

Here `fft` and `ifft` represent forward and inverse 1D Fast Fourier Transforms, respectively. The second parameter of the FFT calls specifies the axis the FFT is performed along (1 for x, 2 for y and 3 for z). The `ddx_k_shift_pos` and similar variables holds precomputed terms $ik_x e^{+ik_x \Delta_x/2}$ from Eq. (7). Since these terms are stored as vectors rather than matrices to save memory, the `bsxfun` routine with parameter `@times` is used to apply (multiply) these terms correctly onto the entire matrix. The function `real` extracts only the real part of its complex argument.

The native CUDA simulation code follows the Matlab ones in such a way the FFTs are calculated by the CUDA FFT library and the operation in between the FFTs are packed into CUDA kernels.

C. Structure of proposed application

The basic structure of the proposed CUDA application is depicted in Fig. 3. Please note that the diagram does not contain all components of the application.

The `PstdElastic3DSolver` class represents the main application class responsible for handling the entire simulation. It provides interface for the data preprocessing, execution of the simulation time loop and data post-processing. During the pre-processing phase, the solver invokes methods of `MatrixContainer` to allocate the memory and load simulation data. The solver class also precalculates constants needed repeatedly. During the simulation phase, the solver invokes appropriate methods implemented in the `SolverCUDAKernels` class according to the simulation model and the type of the media (e.g., linear/nonlinear simulation, homogeneous/heterogeneous and absorption/lossless medium). In this process, the CPU only controls the simulation flow and executes the I/O operations. All of the computation is done on the GPU side. Once the simulation is over, the solver class invokes methods from `MatrixContainer` to store specified data into the output file and free memory. Let us note that time varying quantities are progressively stored during the simulation in a non-blocking manner with the help of the CUDA zero-copy memory and double buffering approaches. Consequently the application terminates.

The `SolverCUDAKernels` class implements computations occurring during the simulation on the GPU. Once one of its methods is invoked, `SolverCUDAKernels` calls an appropriate CUDA kernel on data provided by `MatrixContainer` class.

The `MatrixContainer` class is responsible for data handling in the simulation code. This class implements methods to create, allocate/destroy, and load data into matrices from an input HDF5 file. The create method populates the container with predefined matrices. The existence of some matrices is conditioned by the value of certain control parameters specifying the type of the simulation and the medium. The matrices in the container are then allocated by the allocate method. For each matrix, the memory on both CPU and GPU is allocated. The `MatrixContainer` is also responsible for loading data from the file and data transfers between the device (GPU) and host (CPU) memory spaces.

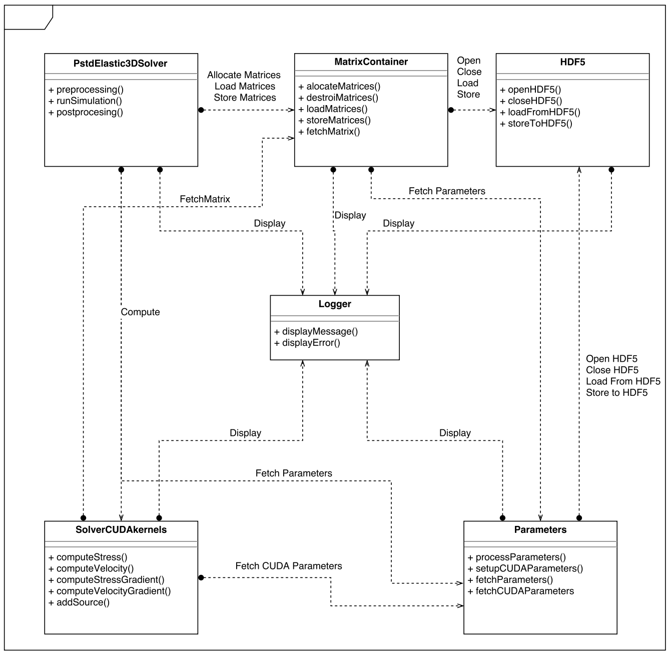

Fig. 3. Diagram of main classes being part of the CUDA application including: main `PstdElastic3DSolver` class, `SolverCUDAKernels` class containing computational kernels, `MatrixContainer` class responsible for matrix handling, `Parameters` class, `HDF5` class which manipulates HDF5 files and `Logger` class to format and display messages.

The `Parameters` class processes the command line parameters, loads simulation control parameters from the input file, sets up device constants and serves other classes with these parameters. This class is modeled as singleton.

The `Logger` class is responsible for all outputs to the standard and error output. It provides diagnostic and progress information to the user as well as inform about possible errors and exceptions.

The `HDF5` class implements a simplified interface to the HDF5 library for easy manipulation, loading and storing of the data.

D. Simulation code workflow

Providing a correct input file is present, a typical code workflow consists of following stages:

- **Initialization** – parameters provided via standard input are checked. A user can specify the input and output file, the GPU to use (otherwise the first available is selected), the number of time steps to perform (useful for benchmarking and debugging), the checkpoint creation interval, the verbosity level and quantities to be recorded. Afterwards, the appropriate GPU is selected, initialized and its compute capability checked.
- **Data loading** – in this stage, the input file is checked first. All the simulation flags are parsed and appropriate matrices allocated. The application may terminate with an error at this point due to an insufficient memory space on the GPU side (this depends on size of simulation). Providing GPU memory can hold all the simulation data,

necessary matrices are loaded from the input file. In this step, CUDA device constants are also set.

- **Data preprocessing and uploading** – during this stage, the necessary constants and auxiliary variables are pre-calculated. The conversion from the Matlab to the C-style notation is preformed. And all the data is transferred to the GPU memory.
- **Simulation run** – provided all data is stored on the GPU, the main simulation loop is started and executed for a given number of time steps with occasional flush of the simulation data to disk.
- **Postprocessing and finalizing** – At the end, the output and checkpoint file is created. The checkpoint file is only generated if the simulation cannot be completed in one go and has to be split into multiple legs, e.g. to satisfy maximum allowed wall clock limitation. When the simulation finishes, the post-processing takes place. Several sampled data may be aggregated and stored into the output file according to command line parameters.

One time step of the simulation consists of the following computations:

1) **Calculation of stress gradient** – described in Eq. (7). As the Cartesian components of σ are kept separately, we first execute a matrix addition kernel to calculate σ out of its components. Next, the 1D FFT of σ is calculated using the CUDA FFT library. Afterwards, a kernel applying the shift term of $ik_x e^{+ik_x \Delta x/2}$ to σ is executed, which is coded as multiplication of two complex numbers. Finally, the inverse 1D FFT is calculated using the CUDA FFT library again. Unfortunately, since CUDA does not allow to directly calculate 1D FFTs in other dimensions than the x one, the transposition kernels are inserted before and after the forward and the inverse FFTs, respectively, and the shift term applied in the transposed space. The 3D transpose kernels were implemented and optimized manually using the shared memory, warp synchronous programming approach and C++ templates.
2) **Calculation of particle velocity** – described in Eq. (8). The calculation is done by multiple kernels composed of a matrix addition and either an element-wise matrix multiplication (heterogeneous medium) or matrix by scalar multiplication (homogeneous medium). Then, the PML is applied near to the domain edges by an additional element-wise matrix multiplication.
3) **Application of velocity source** – done by setting or adding the value of the driving source signal at current time step into the domain at places specified by the source mask. In the case of the velocity source, we modify the v matrix.
4) **Calculation of velocity gradient** – described in Eq. (9) is very similar to the calculation of the stress gradient.
5) **Calculation of stress using Kelvin-Voigt model**.
   a) *Calculation of additional terms* – described in Eq. (10). This is also very similar to the calculation of stress gradient. The difference is that the matrix addition can be combined with either an element-wise matrix multiplication or scalar matrix multiplication into a single kernel. Next, the forward FFT is preformed (data transposed if needed), the shift operator is applied and the inverse FFT is calculated.
   b) *Calculation of stress* – described in Eq. (11). The implementation is composed of a matrix addition and either an element-wise matrix multiplication (heterogeneous medium) or scalar matrix multiplication (homogeneous medium). The implementation accounts for any combination of scalar or matrix variants of the parameters by means of C++ templates and conditional code dispatch.
6) **Calculation of stress using lossless model** – described in Eq. (11). In the lossless case, however, the spatial gradient terms of the time derivative of particle velocity (e.g. $\partial_x \partial_t v_x$) are omitted. The implementation is composed of several element-wise matrix additions and multiplications. Here both λ and/or μ can be either scalar or matrix, and the implementation has to account for any combination of λ, μ, χ and η. This is achieved by C++ templates again.
7) **Application of stress source** – the principle is the same as for the velocity source. In this case, the σ matrix is modified.
8) **Calculation of acoustic pressure** – is straightforward. The normal components of stress (σ_{xx}, σ_{yy} and σ_{zz}) are added together and the value is multiplied by $-\frac{1}{3}$.

IV. Performance Investigation

A. Computational performance

The performance of the developed native CUDA application was evaluated on several Nvidia GPUs based on the Kepler, Maxwell and Pascal architectures with 4 to 24 GB of on-board memory. Table I summarizes the hardware configuration and theoretical performance and memory bandwidth of investigated GPUs. For the lack of C++ version, the original Matlab implementation executed on one node of the Anselm supercomputer[4] integrating 2 8-core Intel Haswell E5-2660 CPUs were taken as reference points.

Fig. 4 shows the growth of the execution time of one simulation time step with increasing domain size from 64^3 to 512^3 grid points. The actual number time steps to complete the simulation is proportional to the size of the simulation domain. Bigger simulations thus take much longer time than smaller ones.

Examining Fig. 4, there is a significant difference between the native CUDA application and the Matlab version caused by manual tuning of the simulation kernels, better spatial and temporal data locality, and the use of optimized FFT kernels. Interestingly, there is a remarkably low difference between different GPU architectures. Although the raw computational

[4]http://www.it4i.cz/

TABLE I
HARDWARE CONFIGURATION OF THE INVESTIGATED CPU AND GPUS.

	Processor			Memory	
CPU name	Architecture	Cores	Peak performance	Capacity	Throughput
2×Intel E5-2470	Sandy bridge	8	18.4 GFLOPs (per core)	96 GB	38.4 GB/s (per CPU)
GPU name	Architecture	Cores	Peak performance	Capacity	Throughput
Tesla K20	Kepler	2,496	3.5 TFLOPs	5 GB	208 GB/s
GTX 980	Maxwell	2,048	4.6 TFLOPs	4 GB	224 GB/s
Titan X	Maxwell	3,072	6.1 TFLOPs	12 GB	336 GB/s
Tesla P40	Pascal	3,840	10 TFLOPs	24 GB	345 GB/s
Tesla P100	Pascal	3,584	9.3 TFLOPs	16 GB	720 GB/s

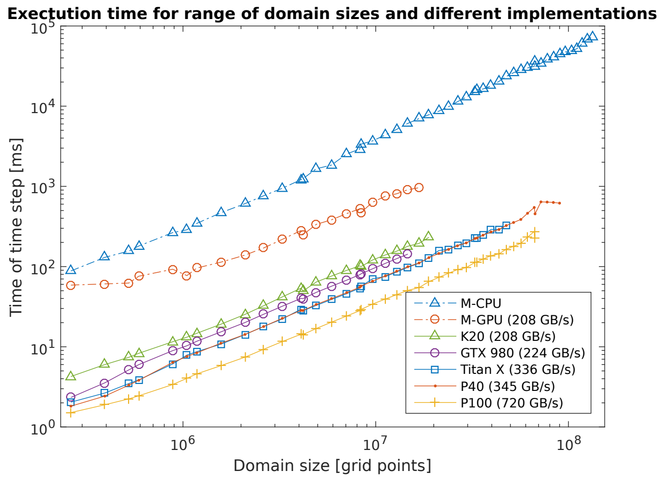

Fig. 4. Execution times for one simulation time step. The domain size vary between 64^3 and 512^3 grid points. The first two curves M-CPU and M-GPU stand for the Matlab code running on a CPU and GPU, respectively, using the Matlab Parallel Computing Toolbox. The others represent the native CUDA application. The numbers at the curves represent the maximal and average speed-up with respect to the Matlab CPU code.

performance grew by an order of magnitude between Kepler and Pascal architectures, the observed speed-up is close to 4. This suggests the code is strongly memory bound, which can be supported by an $n \log n$ time complexity of the FFT, and a linear time complexity of other simulation kernels. The execution times thus copy the memory bandwidth shown in the legend of Fig. 4. To highlight the benefits of our implementation, let us mention that the performance of the native code is approx. 5.6 times higher than the Matlab GPU code when running on the same GPU. The best performance was achieved on the Pascal P100 GPU, which was on average 108 times faster than Matlab CPU version with peak acceleration factor of 158.5.

Fig. 4 also reveals that some simulation domain sizes cannot be executed on particular GPUs. This is given by the memory requirements. Therefore, an analytic model predicting the GPU memory consumption was derived, see Eq. (12).

B. Memory consumption

Since realistic ultrasound simulations require large domains and memory capacity of GPU's is very limited, we derived a memory consumption model to help reveal approximate limits of certain GPUs. The memory consumption model can be described by following equation:

$$Mem\ [GB] \approx \frac{(46+A)NxNyNz}{1024^3/4} + \frac{6(Nx+Ny+Nz)}{1024^3/4} + \frac{2GCS+input}{1024^3/4} \quad (12)$$

In Eq. (12), Nx, Ny and Nz represent the grid dimensions. Number A in the first term has a value from interval $<0,8>$ and this number is dependent on the number of material properties that are heterogeneous. For example, when the material density is heterogeneous, 4 additional matrices have to be allocated. The GSC term holds the maximum of values $(\frac{Nx}{2}+1)NyNz$, $Nx(\frac{Ny}{2}+1)Nz$ and $NxNy(\frac{Nz}{2}+1)$. At most, the GCS number of elements is needed to store all results computed by 1D FFTs. However, the values in the frequency domain are complex numbers, and therefore, the actual number of elements in this matrix is double the GCS. The $input$ represents the amount of additional user-defined input data, usually sources. There could be up to 10 sources that can have sizes up to the size of the domain (very unlikely except for the initial pressure condition `source_p0` which is always the size of domain). Furthermore, each source has a time varying signal assigned to it. This means that for each point of source there could be a time series of values. Number 46 represents variables that are mandatory and have to be allocated for every type of the simulation. Also, 1D vector variables are taken into account. There are six 1D vectors needed for each dimension for storing values of shift terms and values of PML. The equation does not account for every allocation though, for example device constants are omitted. These kinds of allocation are not part of the equation because they are either hard to predict or negligible.

This memory model was compared with the maximal amount of memory consumed by Pascal P40 GPU. Fig. 5 shows a close agreement with small differences caused by the size of the executable, FFT plans and the CUDA driver.

C. Performance limitations

Fig. 6 shows the flat profile of the application produced by the Nvidia profiler determining the order of importance of particular kernels and the theoretical limits of the acceleration. The kernels implementing FFT are part of CUDA FFT library,

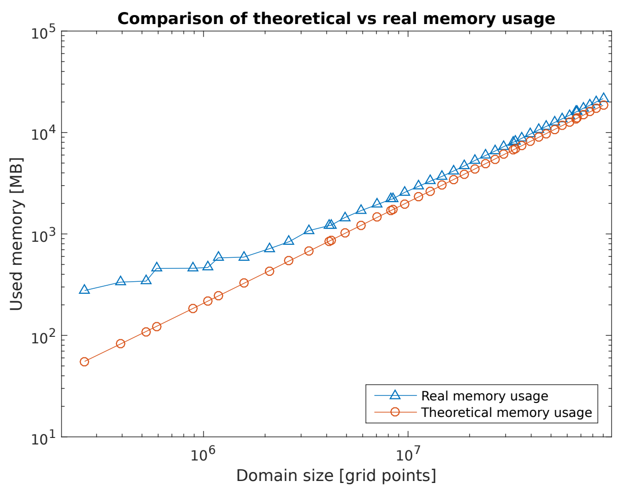

Fig. 5. Comparison of the predicted and measured GPU memory consumption for an Nvidia Pascal P40 with 24GB of memory for domain sizes between 64^3 and 512^3.

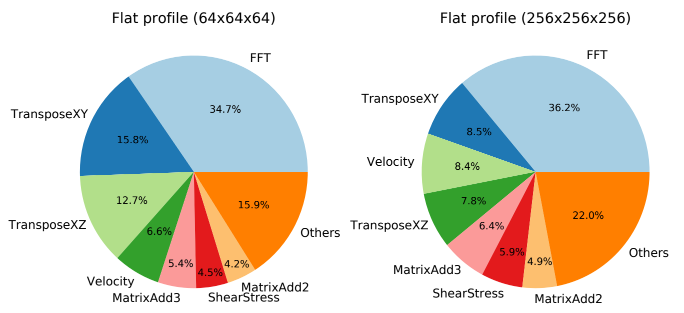

Fig. 6. The flat profile of the simulation code. On left side, values for the domain size of 64^3 grid points. On right, values for the domain size of 256^3 grid points.

and therefore, believed to be highly optimized. In addition, the transposition kernels were taken from our previous work where their quality had been proven. Therefore, there is no space to further accelerate these kernels. All other kernels are implemented by hand, therefore, their optimization present a way to increase the performance. However, these kernels do not take up the vast majority of the overall execution time. In fact, they take up between 36.9% and 47.5% of execution time on domain of a size 64^3 (left side of figure) and 256^3 (right side of figure), respectively. Considering the Amdhal's law, there is very little room for further improvement.

Table II presents important performance metrics of selected kernels. The metrics were calculated base on the load instructions only because, unlike write instructions, they are source of stalls. Relative throughput and performance are calculated according to peak values (345 GB/s and 10 TFLOPs). The throughput metric is gathered from the kernel point of view which allows the relative throughput to be more than 100% due to cache hits. Load efficiency is reaching peak values for almost all kernels. Please note that application is operating on single-precision numbers and therefore peak

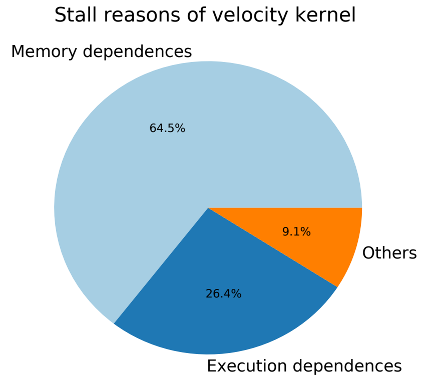

Fig. 7. Stall reasons in the velocity computation kernel on Pascal P40 simulating domain of 256^3 grid points.

value of load efficiency is 50%. However, FFT operates on complex numbers which are composed of 2 single-precision numbers (equivalent of double-precision number in terms of bit width) which allows FFT operation to reach 100% of load efficiency. All kernels exhibits poor relative performance and high achieved memory throughput. This is another indication that the application is strongly memory bound. Fig. 7 clearly supports this fact. In the figure, it can be seen that more than two thirds of the stalls are caused by memory dependencies. This means the next instruction cannot be issued due to data dependency on the previous load instruction.

V. Conclusions

This paper has presented a GPU-accelerated implementation of the elastic wave propagation in biological materials. A part of the paper is extensive performance analysis. The Matlab version accelerated by the GPU is sped-up by a factor of 5 when running on the same GPU. Considering the standard Matlab implementation executed by a dual 8 core server, the modern Pascal P100 GPU can offer more than 150 times faster execution, which not only means a great time savings, but also a great reduction of computation cost. Since the Matlab is excluded from the critical part, the execution of the simulation is no longer dependent on Matlab installation.

Using the implemented CUDA code, the simulation in domain of at most 448^3 grid points is possible. Such a simulation executing over 4,655 time steps would last for about 47.9 minutes on an Nvidia Pascal P40 GPU. A typical simulation on domain of a 256^3 grid points carried over 2,660 steps would then last for 8.6 minutes on less powerful Tesla K20. This presents a great breakthrough in the ultrasound treatment planning, since dozens of simulations are necessary before the therapy could be applied, and the time allocated to treatment planning should not exceed several hours.

The performance of the algorithm on a specific GPU is strongly dependent on its memory throughput. Therefore, it is possible to say that if a Tesla V100 GPUs with new Volta architecture is used, the acceleration factor of the elastic code will rise significantly. It has theoretical memory bandwidth

TABLE II
PROFILE OF SELECTED KERNELS ORDERED BY THE TIME OF EXECUTION.

Kernel name	Achieved throughput	Rel. throughput	Load efficiency	Performance
FFT 1	143.32 GB/s	41.54 %	99.28 %	5.43 %
FFT 2	154.46 GB/s	44.77 %	86.00 %	2.72 %
TransposeXY	288.45 GB/s	83.60 %	42.11 %	0.00 %
Velocity	609.95 GB/s	176.79 %	41.48 %	1.20 %
TransposeXZ	274.28 GB/s	79.50 %	42.11 %	0.00 %
MatrixAdd3	418.14 GB/s	121.20 %	50.00 %	0.32 %
ComputeShearStress	556.06 GB/s	161.17 %	43.30%	1.17 %
MatrixAdd2	368.21 GB/s	106.72 %	50.00 %	0.21 %

of 900 GB/s and therefore, the CUDA implementation would accelerate the Matlab solution approximately by a factor of 200.

VI. Acknowledgment

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science - LQ1602, and from the Large Infrastructures for Research, Experimental Development and Innovations project IT4Innovations National Supercomputing Center - LM2015070. This project has received funding from the European Union's Horizon 2020 research and innovation programme H2020 ICT 2016-2017 under grant agreement No 732411 and is an initiative of the Photonics Public Private Partnership. This project was also supported by the Engineering and Physical Sciences Research Council, UK, grant numbers EP/L020262/1, EP/M011119/1 and EP/M011119/1. This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID pr14.

References

[1] K. Okita, R. Narumi, T. Azuma, S. Takagi, and Y. Matumoto, "The role of numerical simulation for the development of an advanced HIFU system," *Computational Mechanics*, vol. 54, no. 4, pp. 1023–1033, oct 2014.

[2] E. Bossy, F. Padilla, F. Peyrin, and P. Laugier, "Three-dimensional simulation of ultrasound propagation through trabecular bone structures measured by synchrotron microtomography." *Physics in medicine and biology*, vol. 50, no. 23, pp. 5545–5556, 2005.

[3] K. Raum, Q. Grimal, P. Varga, R. Barkmann, C. C. Glüer, and P. Laugier, "Ultrasound to assess bone quality," *Current Osteoporosis Reports*, vol. 12, no. 2, pp. 154–162, 2014.

[4] J. W. Jenne, T. Preusser, and M. Günther, "High-intensity focused ultrasound: Principles, therapy guidance, simulations and applications," *Zeitschrift fur Medizinische Physik*, vol. 22, no. 4, pp. 311–322, dec 2012.

[5] G. Pinton, J.-F. Aubry, M. Fink, and M. Tanter, "Effects of nonlinear ultrasound propagation on high intensity brain therapy," *Medical Physics*, vol. 38, no. 3, p. 1207, 2011.

[8] M. Javadi, W. G. Pitt, C. M. Tracy, J. R. Barrow, B. M. Willardson, J. M. Hartley, and N. H. Tsosie, "Ultrasonic gene and drug delivery using eLiposomes," *Journal of Controlled Release*, vol. 167, no. 1, pp. 92–100, 2013.

[6] Y.-F. Zhou, A. Syed Arbab, and R. X. Xu, "High intensity focused ultrasound in clinical tumor ablation." *World journal of clinical oncology*, vol. 2, no. 1, pp. 8–27, 2011.

[7] P. Sminia and R. Mayer, "External beam radiotherapy of recurrent glioma: Radiation tolerance of the human brain," pp. 379–399, 2012.

[9] A. B. Etame, R. J. Diaz, C. A. Smith, T. G. Mainprize, K. Hynynen, and J. T. Rutka, "Focused ultrasound disruption of the blood-brain barrier: a new frontier for therapeutic delivery in molecular neurooncology," *Neurosurgical Focus*, vol. 32, no. 1, p. E3, 2012.

[10] J. L. B. Robertson, B. T. Cox, J. Jaros, and B. E. Treeby, "Accurate simulation of transcranial ultrasound propagation for ultrasonic neuromodulation and stimulation," *The Journal of the Acoustical Society of America*, vol. 141, no. 3, pp. 1726–1738, mar 2017.

[11] D. Komatitsch, D. Göddeke, G. Erlebacher, and D. Michéa, "Modeling the propagation of elastic waves using spectral elements onacluster of192 gpus," *Computer Science - Research and Development*, vol. 25, no. 1, pp. 75–82, May 2010.

[12] D. Micha and D. Komatitsch, "Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards," *Geophysical Journal International*, vol. 182, no. 1, pp. 389–402, 2010.

[13] E. Robinson and D. Clark, "The wave equation," *The Leading Edge*, vol. 6, p. 14, 1987.

[14] R. P. Feynman, R. B. Leighton, and M. Sands, *The Feynman Lectures on Physics, Vol. I: The New Millennium Edition: Mainly Mechanics, Radiation, and Heat*, 2011, vol. 24.

[15] B. E. Treeby, J. Jaros, D. Rohrbach, and B. T. Cox, "Modelling elastic wave propagation using the k-Wave MATLAB Toolbox," in *2014 IEEE International Ultrasonics Symposium*, no. 5. IEEE, sep 2014, pp. 146–149.

[16] M. Caputo, J. M. Carcione, and F. Cavallini, "Wave Simulation in Biologic Media Based on the Kelvin-Voigt Fractional-Derivative Stress-Strain Relation," *Ultrasound in Medicine and Biology*, vol. 37, no. 6, pp. 996–1004, 2011.

[17] Q. H. Liu, "The pseudospectral time-domain (PSTD) algorithm for acoustic waves in absorptive media," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 45, no. 4, pp. 1044–1055, 1998.

[18] M. Y. Hussaini, D. A. Kopriva, and A. T. Patera, "Spectral collocation methods," *Applied Numerical Mathematics*, vol. 5, no. 3, pp. 177–208, 1989.

[19] M. Tabei, T. D. Mast, and R. C. Waag, "A k-space method for coupled first-order acoustic propagation equations." *The Journal of the Acoustical Society of America*, vol. 111, no. 1 Pt 1, pp. 53–63, jan 2002.

[20] K. C. Meza-Fajardo and A. S. Papageorgiou, "On the stability of a non-convolutional perfectly matched layer for isotropic elastic media," *Soil Dynamics and Earthquake Engineering*, vol. 30, no. 3, pp. 68–81, mar 2010.