Acoustical Engineering 2015

Sound Synthesis Using Physical Modeling on Heterogeneous Computing Platforms

M. Pluta[a,*], B. Borkowski[a], I. Czajka[b] and K. Suder-Dębska[b]

[a]AGH University of Science and Technology, Department of Mechanics and Vibroacoustics, Al. Mickiewicza 30, 30-059, Krakow, Poland

[b]AGH University of Science and Technology, Department of Power Systems and Environmental Protection Facilities, Al. Mickiewicza 30, 30-059, Krakow, Poland

The paper presents a comparison of central processing unit (CPU) and graphics processing unit (GPU) performance in sound synthesis based on physical modeling. The goal was to achieve real-time performance with two- and three-dimensional finite difference (FD) instrument models. Two abstract instruments, a membrane and a block, were modeled and tested using a CPU and a GPU in the OpenCL framework to find a threshold of real-time model size. Two different algorithms were compared. With a parallelized algorithm, a middle-class GPU outperformed a top-class CPU by factor of 2.5 in 2D and by factor of 7.5 in 3D model. Synchronization issues in parallel GPU calculations were discussed and addressed. The results show that GPUs can significantly speed up real-time musical instrument simulations, allowing for developing more complex and realistic models.

DOI: 10.12693/APhysPolA.128.A-22

PACS: 43.40.Dx, 43.58.Ta, 43.75.Zz

1. Introduction

An increase in available computing power that could have been observed for a past few decades leads not only to incremental improvements, but at some points it opens entirely new research areas to explore. One of such areas is sound synthesizing in real time based on physical modeling of the instrument.

While most sound synthesis methods attempt to recreate only the sound of the instrument, e.g. by analysis and resynthesis of its dynamic spectrum, physical modeling-based synthesis simulates the instrument itself by creating and using its numerical model. If the model is recreated in sufficient detail, it can be used to produce the sound in the way the real instrument does. It includes the variability of registers, i.e. differences in timbre related to pitch regions, as well as different articulation techniques related to various methods and parameters of excitation. Good models can be used to study sound production phenomena in instruments or predict changes in sound caused by alterations to the instrument properties, such as shape, size, or material. On the musical performance side, such models can be played, but unlike natural instruments, model parameters can be altered as desired. It is also possible to play it automatically using a sequencer, e.g. to ensure repeatability of selected parameters, which is not possible with natural instruments without resorting to robots.

There is still an issue of control, as it is difficult to set and continuously tune performance parameters with usual computer controls or algorithms. That is why physical modeling synthesis is especially predisposed to use it with physical controllers resembling original musical instruments [1]. Here, a problem emerges: complex models that can utilize control possibilities provided by physical controllers are computationally expensive. However, physical controllers work in real time, and so must work the model, in order to provide auditory feedback.

Although there are similarities between study-oriented simulations and sound synthesis, there are also important differences. Simulations are aimed at studying acoustical phenomena in the instrument, with the possibility to produce sound, usually without strict time limits for the computations. The parameters are set before the simulation starts. Sound synthesis often works as an instrument, with the real-time auditory feedback and control over the performance parameters. This includes control over the excitation and selected parameters of the instrument related mostly to its geometry, e.g. shortening strings, lengthening pipes, etc.

In case of models working in real time there is a trade-off between model complexity and available computing power. This paper focuses on increasing the computing power to allow more complex models, by utilizing all processing units present in contemporary computers to perform the simulation: not only central processing units (CPUs), but also graphics processing units (GPUs). This kind of approach is referred to as heterogeneous computing [2].

Similar research has been conducted with encouraging results [3–5]. However, solutions chosen in those studies limit possible devices only to GPUs, and only from one vendor. The study presented here imposes no such limitations. Results will apply to much broader set of devices. The research is aimed, in future, to combine computing power of various types of devices working simultaneously.

*corresponding author; e-mail: pluta@agh.edu.pl

Initial findings were reported in [6]. This paper presents results of further study. Two different implementations of finite difference (FD) membrane simulation in 2D and one implementation of a FD block simulation in 3D were compared on a CPU and a GPU to find a threshold for the real-time model size. Section 2 of this paper presents FD models of the membrane and the block used in simulations. Section 3 discusses GPU computing frameworks. Section 4 describes implementations of FD models with regards to heterogeneous computing problems. Test results are discussed in Sect. 5. Finally, Sect. 6 presents conclusions of the study.

2. Models of instruments

Not all simulation methods used in instrument studies are appropriate for sound synthesis based on physical modeling. The limiting factor is the requirement of the real-time control over the simulation and real-time auditory feedback. The model has to be controlled in short time intervals of the order of 10 ms. Two methods are commonly used to achieve it: the finite difference (FD) method and the waveguide (WG) synthesis method. While WG method is generally more efficient [7], FD method is more versatile [8, 9], and allows for more diverse models. For this reason, FD has been chosen for the study.

The presented work is aimed at testing performance of two implementations of 2D model and comparing CPU and GPU performance in 2D and 3D models. The operation of the model itself is of secondary importance. The model should be straightforward and its proper working parameters should be known. It should also scale up and down easily, to be a good base for performance testing. A square membrane (in 2D) and a cubical block (in 3D) have been chosen to be modeled. Despite simplicity, they behave like percussion instruments and produce instrument-like sounds.

The FD method was used to obtain an approximate solution of the wave equation with dissipation [10] in two dimensions:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}-\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}=\eta\frac{\partial u}{\partial t}, \tag{1}$$

and in three dimensions:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}-\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}=\eta\frac{\partial u}{\partial t}, \tag{2}$$

where $u(x,y,t)$ or $u(x,y,z,t)$ is the displacement, c is the velocity of acoustic waves, and η is the viscosity coefficient.

For the purpose of sound synthesis, the second-order Taylor expansion gives reasonable approximation [8] in two dimensions:

$$u_{i,j}^{n+1}=\frac{1}{(1+A)}\left(\rho u_{\mathrm{sur\,2D}}+2\left(1-2\rho\right)u_{i,j}^{n}\right.$$
$$\left.-\left(1-A\right)u_{i,j}^{n-1}\right), \tag{3}$$

and in three dimensions:

$$u_{i,j,k}^{n+1}=\frac{1}{(1+A)}\left(\rho u_{\mathrm{sur\,3D}}+2\left(1-3\rho\right)u_{i,j,k}^{n}\right.$$
$$\left.-\left(1-A\right)u_{i,j,k}^{n-1}\right), \tag{4}$$

where i, j and k are spatial indexes, n is the time index, $\rho=c^2\Delta t^2/\Delta x^2$, $A=\frac{1}{2}c^2\eta\Delta t$, $u_{\mathrm{sur\,2D}}=u_{i+1,j}^{n}+u_{i-1,j}^{n}+u_{i,j+1}^{n}+u_{i,j-1}^{n}$, and $u_{\mathrm{sur\,3D}}=u_{i+1,j,k}^{n}+u_{i-1,j,k}^{n}+u_{i,j+1,k}^{n}+u_{i,j-1,k}^{n}+u_{i,j,k+1}^{n}+u_{i,j,k-1}^{n}$. It has been assumed that $\Delta x=\Delta y=\Delta z$. In order to maintain simulation stability, both spatial and temporal steps had to be related to each other by means of the following condition [8]:

$$\rho\le 1. \tag{5}$$

Subsequent displacement values $u_{i,j}^{n+1}$ or $u_{i,j,k}^{n+1}$ were calculated using the leap-frog algorithm. The membrane in 2D was clamped along the boundary [11]. This implied the boundary condition: on the whole boundary, the scalar gain of 0 had to be maintained. The initial condition was simulated by the Gaussian impulse [9]. Similar conditions were applied to the block in 3D.

3. Choice of the computing framework

Current generation of graphics processing units (GPUs) can perform general purpose computations. However, because of main purposes they serve, GPUs differ from CPUs in a number of ways [12]. The core of a CPU is versatile and fast in single-threaded applications. It has an out-of-order super-scalar architecture, a branch predictor, and can process more than one operation per cycle. Desktop and laptop CPUs contain a small number of such cores — usually 2 or 4. GPU processing elements are designed towards high throughput. They execute a number of scalar instructions in parallel. Lower complexity and slower clock allow to pack many of them in one GPU. CPUs use large multi-level cache memory. Older GPUs used much smaller cache, due to different usage scenario, where memory latency was less important. In recent GPUs, cache has been increased. Memory bandwidth is another important difference. A system memory used by CPUs in current desktops or laptops has a peak bandwidth of 25.6 GB/s. A GPU memory bandwidth reaches 30–80 GB/s in middle class laptop GPUs, and up to 300 GB/s in high-end desktop GPUs, due to higher clock speed and wider bus. However, in most cases, data to be processed usually already resides in system memory, and CPU can use it immediately, while for GPU it has to be copied from main memory to GPU memory before, and in the opposite direction after computations. Those differences show clearly that GPUs could be more efficient than CPUs in parallel computing. They are already utilized in physical modeling where a large number of nodes can be calculated simultaneously.

There are two main standards for general-purpose computing on GPUs (GPGPU): CUDA [13] and OpenCL [14]. Some sources claim that CUDA is more efficient [15, 16]. The others [17] point out OpenCL possibility to use various devices from all major GPU and CPU vendors (e.g. AMD [18], Intel [19], Nvidia [20]), contrary to CUDA being centered around Nvidia. So far, there have been some successful experiments with implementing sound synthesis based on physical modeling using CUDA [3–5]. In the present study, the OpenCL variant is explored. The main reason for this choice is the fact that OpenCL allows calculations to be executed simultaneously on various devices and types of devices. More complex models could be divided into subsystems operating on separate devices: several (three or four) GPUs and one or more multi-core CPUs. All those devices can fit into one personal computer, creating a very complex and powerful, but still compact physical modeling synthesizer.

OpenCL provides an API and a runtime for cross-platform parallel computing [21]. While CUDA architecture reflects physical architecture of Nvidia GPUs, OpenCL operates with an abstract hierarchy of elements due to compatibility with various types of devices. A host (usually a CPU) coordinates execution and data transfer to and from an array of compute devices (CPUs, GPUs, or different kinds of processors). Compute devices consist of arrays of compute units. One compute unit is composed of an array of processing elements. Computing tasks can be comprised of data-parallel kernels, applying a single function over a range of data elements in parallel. For every element in 1-, 2-, or 3-dimensional kernel index space, a work-item will be executed. All work-items execute the same program, however its execution can differ due to branching based on data or the index assigned to the work-item. Work-items are grouped into work-groups. Communication between work-items is possible only within a work-group.

A host can be programmed in C (C89 compatible) or C++. Kernels are programmed in a C-like language (based on C99). Kernel sources are loaded during the run-time of the host program, compiled just-in-time, and sent to the compute device [22].

4. Implementations of models

The first part of the study has been conducted using 2D model only, with two different implementations. On the basis of the better performing variant, a 3D implementation was created. 2D and 3D implementations worked with 2- and 3-dimensional OpenCL workspaces, accordingly. For efficient data transfers, all the grid data (displacement values for three consecutive time steps as well as boundary conditions) was stored in 1-dimensional memory array, with indexing appropriate for 2D or 3D problem. In two dimensions grids from 8×8 up to 256×256, with the increment of 8 in each dimension, were used. In three dimensions, grids ranged from $8 \times 8 \times 8$ to $32 \times 32 \times 32$, with the increment of 4. The result of the calculation was always a sound sample with 44100 Hz sampling frequency.

The first 2D implementation (hereinafter referred to as V1 or "variant 1") was the same as the one used in [6]. In this variant, calculation of each waveform sample involves one execution of the kernel on an appropriate number of work items (threads), as required by the grid size. There is a high overhead caused by memory transfers between the host and the compute device and by starting the kernel in each step of the simulation. While not optimized for high performance, this variant has the feature of being able to control the grid and update waveform at each time step.

2D V1 host-side algorithm has been implemented as follows:

1. Set the grid and the initial condition.
2. For each time step:
   (a) copy buffers (current and previous time step) to the compute device,
   (b) execute the kernel,
   (c) retrieve buffer (next time step) from the compute device,
   (d) get the result (one signal sample) from the selected grid cell,
   (e) rotate buffers.

The kernel-side of the 2D V1 algorithm consists of the following steps:

1. Check kernel space index.
2. Execute only if inside the grid.
3. Cell grid on border?
   - Yes: apply gain = 0.
   - No: calculate the next time step for a single grid cell according to Eq. (3).

The second 2D implementation (V2 or "variant 2") addresses performance problems of variant 1. It reduces the number of unnecessary memory transfers and starts of the kernel by:

- calculating more than one signal sample in one kernel run,
- using a ring buffer by the kernel and thus skipping buffer rotation.

At this point, some issues related to the architecture of parallel compute devices (especially GPUs) have to be considered. V1 kernel was executed on as many work items as the grid size required. There was no problem of synchronization between work items, since all of them were calculating displacements for the same time step. Even if the grid size exceeded maximum number of simultaneous threads, calculations were automatically divided into a number of smaller work groups of sizes possible to be handled by the compute device and calculated group after group. If the kernel has to calculate not one but a number of consecutive time steps, like in V2, it must be ensured that neighboring cells in the current and the previous step have been synchronized. Such a precaution is required because kernels execute calculations independently in each work item, so their execution time can vary due to different conditions. OpenCL has a mechanism of barriers that can stop all kernels until all of them reach certain point, but it works only within a work group. Even using barriers, larger grids would be divided into independently calculated sub-grids, according to the device's internal work group division, preventing synchronization between some areas of the grid.

The solution is to ensure that only one work group handles the grid. A mixed, serial-parallel implementation has been developed. The algorithm checks the maximum size of the work group for the device. If the grid is smaller, each work item handles one cell. Otherwise, the work item loops through a number of cells. Barriers are used after each time step, so that entire grid stays synchronized.

In comparison to V1, V2 host-side algorithm leaves more operations for the kernel to perform and has been implemented as follows:

1. Set the grid and the initial condition (next, current, and previous step in one 1-dimensional array).
2. Compare maximum work group size to the grid size and set the number of grid cells to be calculated serially in one work item.
3. Set a signal buffer length (number of time steps to calculate in one kernel run).
4. Copy grid buffer to the compute device.
5. Execute the kernel.
6. Retrieve buffers (grid and signal) from the compute device.

Kernel side of the V2 consists of the following steps:

1. Set indexing pointers for ring buffer storing the grid (for next, current, and previous step).
2. Set current sample number to 0 (beginning of the wave buffer).
3. Set current cell to 0 (the first of a number of cells to calculate in this work item).
4. Check position in work group index space and calculate cell position on the grid.
5. Cell grid on border?
   - Yes: apply gain = 0.
   - No: calculate the next time step for a single grid cell according to Eq. (3).
6. Read the observation point and store it in the wave buffer.
7. If there are more cells to calculate, increase the current cell number and go to 4.
8. Update indexing pointers for the ring buffer.
9. Barrier.
10. If there are more time samples to calculate, increase the current sample number and go to 3.

Variant 2 is more efficient than the previous one and it automatically adapts to the capabilities of the compute device (number of simultaneous threads). Signal buffer length can be tuned according to the scenario of use. The longer the buffer, the better the performance, but also higher audio and control latency, since while running, kernel is inaccessible from the host.

3D implementation uses the same algorithm as 2D V2, but with 3-dimensional grids and with next time step calculated according to Eq. (4). It introduces the possibility to define boundary not only on the ends of the grid, but in any cell, creating thus a basis for simulation of differently shaped instruments. Boundary conditions are implemented as a mask for a main grid — an additional 3D grid, the same size as the main one, sent to the kernel. The kernel checks if the cell is on border by reading mask value under the same coordinates as the calculated cell. Cubical block does not require border mechanism more complex than this used in V2, but for the sake of further study, border mask was used here as well.

5. Results

Two devices were compared: a GPU — Nvidia GeForce GT 750M, and a CPU — Intel Core i7-4700HQ, both being components of a notebook computer. At the time of the study, the former was a middle-class mobile GPU, and the latter — a top-class mobile CPU. Larger number of devices was tested in a previous study [6].

All the tests were conducted using 64-bit Linux system. Calculations were performed using single precision (32-bit) floating point values, adequate for the sound synthesis purposes. GPU results were bit-to-bit compared to CPU results to check whether GPU introduces any numerical errors of its own, but both results were identical.

In the first part of the study, 2D algorithms (V1 and V2) were executed using both devices. In all tests, a sound sample 1 s long was generated using grids from 8×8 up to 256×256 cells, with the increment of 8 in each dimension. In variant 2 wave buffer size was set to 1 s (44100 samples). Simulation execution time served as a performance estimation. Each test was performed 10 times to calculate the mean value and the standard deviation. In order to achieve the real-time performance, the calculation time had to be lower than 1 s. The results are collected in Table I and in Figs. 1–4.

Firstly, it is important to note that variant 1 of the algorithm cannot perform in real time on tested devices — even for the smallest grids (8×8), the calculation time for a 1 s long sample was longer than 1 s. So, the use of variant 2 is a necessity. Variant 2 performance gain over variant 1 is large enough even for a single core

TABLE I

2D simulation time (in s, mean value from 10 test runs) for a sound sample 1 s long. Results allowing real time operation (below 1 s) are marked in bold.

	Variant 1		Variant 2		
Grid size	GeForce GT 750M	Core i7-4700HQ (4 cores)	GeForce GT 750M	Core i7-4700HQ (4 cores)	Core i7-4700HQ (1 core)
8×8	1.467	1.115	**0.053**	**0.01**	**0.043**
16×16	1.467	1.182	**0.05**	**0.03**	**0.18**
24×24	1.474	1.236	**0.06**	**0.07**	**0.415**
32×32	1.502	1.281	**0.07**	**0.155**	**0.74**
40×40	1.538	1.36	**0.13**	**0.214**	1.009
48×48	1.603	1.393	**0.16**	**0.347**	1.35
56×56	1.747	1.449	**0.21**	**0.453**	1.73
64×64	1.791	1.572	**0.21**	**0.528**	2.107
72×72	1.882	1.743	**0.32**	**0.756**	2.63
80×80	2.029	1.888	**0.34**	**0.972**	3.12
88×88	2.268	2.042	**0.44**	1.041	3.697
96×96	2.402	2.17	**0.45**	1.192	4.399
104×104	2.493	2.337	**0.61**	1.579	5.08
112×112	2.939	2.51	**0.63**	1.682	5.9
120×120	3.085	2.736	**0.76**	1.573	6.694
128×128	2.93	2.931	**0.72**	1.969	7.64
136×136	3.62	3.282	1.03	2.343	8.64
144×144	4.074	3.551	1.02	2.647	9.578
152×152	4.432	3.823	1.57	2.788	10.618
160×160	4.923	4.057	1.57	3.184	11.8
168×168	5.31	4.369	1.88	2.97	12.893
176×176	5.767	4.67	2.3	3.328	14.14
184×184	6.242	5.02	2.431	3.794	15.4
192×192	6.539	5.363	2.28	4.249	16.96
200×200	7.092	5.735	2.67	4.102	18.224
208×208	7.517	6.122	3.29	5.58	19.715
216×216	8.19	6.514	3.53	6.034	21.4
224×224	8.557	6.949	3.24	5.306	23.024
232×232	9.247	7.36	3.79	5.933	24.648
240×240	9.58	8.872	3.85	6.83	26.686
248×248	9.965	9.534	4.09	7.31	28.154
256×256	10.359	9.979	3.73	7.919	30.481

of the CPU to perform in real time with small grids. Secondly, an interesting fact is that a middle-class GPU performs significantly faster than a top-class CPU even though GPU calculations involve copying data through the PCI Express bus in both directions. GPU can work in real time with 128×128 grid, while CPU handles only 80×80 grid (about 2.5 times less cells). In case of 128×128 grid, GPU's calculation time is more than 2.5 times shorter than CPU's.

CPU calculations that heavily utilize all cores pose a problem. As it can be seen in Fig. 1, standard deviations for CPU calculation times are very large comparing to almost non existent in case of GPU. As a consequence, any precise estimation of fully parallelized CPU

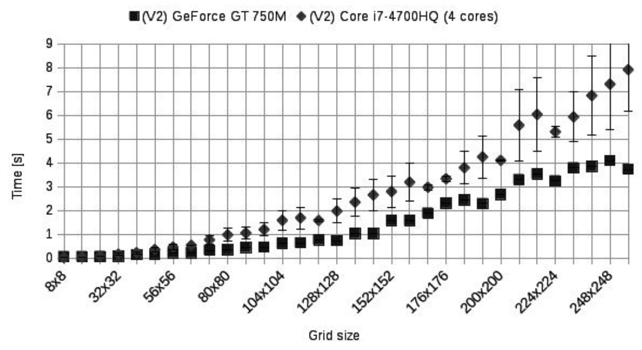

Fig. 1. Comparison between GPU and CPU in variant 2 of the 2D algorithm (mean values with standard deviations).

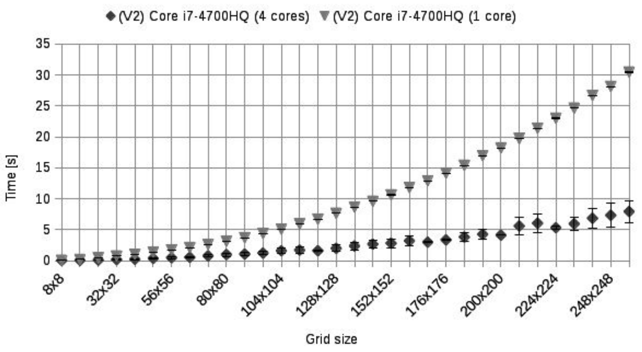

Fig. 2. Comparison between 1 core and 4 cores of CPU in variant 2 of the 2D algorithm.

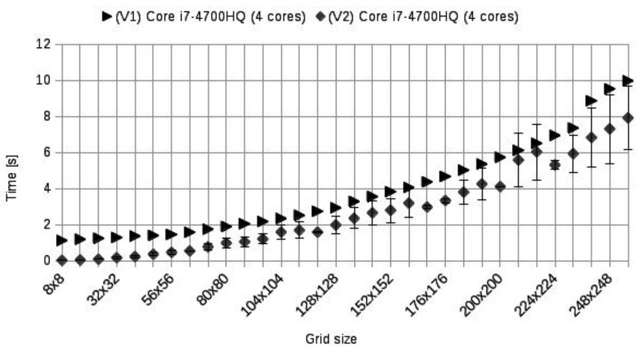

Fig. 3. Comparison between two variants of the 2D algorithm run on the CPU.

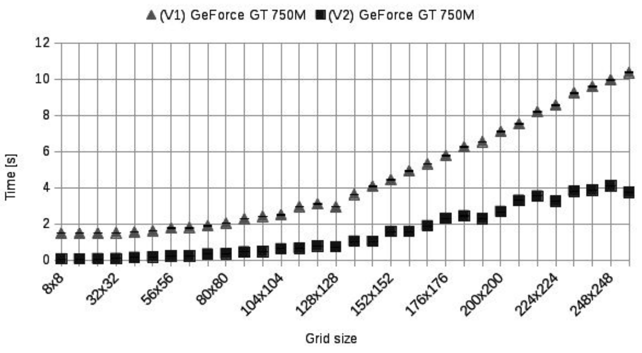

Fig. 4. Comparison between two variants of the 2D algorithm run on the GPU.

execution time would be impossible. The reason for this is the fact that during simulation, the CPU handles also the operating system, while the only task of GPU is the simulation. If only one CPU core is utilized (Fig. 2), execution time can be estimated more precisely, but it is, as expected, 4 times longer.

In case of GPU it can be observed that some grid size increments actually shorten the calculation time. It happens when the grid side is a multiple of 32 (e.g. from 120×120 to 128×128, or from 216×216 to 224×224). So, it is better to use the total number of work items that is a multiple of 1024. This fact needs more consideration, but is most probably related to the GPU hardware architecture.

Transition from variant 1 to variant 2 brings much smaller improvement in case of CPU (Fig. 3) than GPU (Fig. 4). The reason is that variant 2 reduces the time needed for copying memory buffers to and from the compute device. In case of CPU, there is no need for this operation, so no gain is observed. The gain is large in case of GPU, where memory copying was one of the performance limiting factors.

In further test, the impact of the wave buffer size on the performance in 2D V2 algorithm was studied (Fig. 5). Buffer sizes from 1 to 100 (step 1) and from 441 to 44100 (step 441) samples were examined. For each buffer size, the largest grid that was calculated in time shorter than buffer length (real-time requirement) was adopted as the the result. GPU performed in real time for its largest possible grid (128×128) already with a 39 sample buffer (less than 1 ms). Further enlarging of the buffer was not necessary. With the same buffer length, CPU needed the grid to be two times smaller in each dimension (40×40) as its largest real-time performing grid. Small buffer required in the case of GPU allows for a very responsive instrument control.

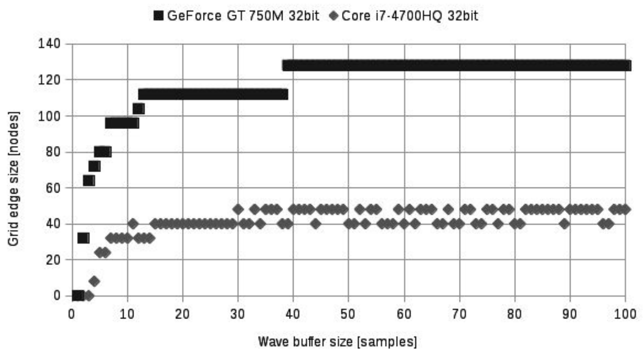

Fig. 5. The largest 2D grid simulated with V2 algorithm for various wave buffer sizes.

In some cases (possibly with a very long simulation, to avoid accumulation of rounding errors) there may be a need for double-precision (64-bit) calculations. In order to maintain the real-time performance of 64-bit V2 algorithm, it was necessary to reduce grid size from 128×128 to 104×104 in case of GPU, and from 80×80 to 72×72 in case of CPU. Double precision impact is smaller in CPU case, but even for GPU it is not as large as the 32-times drop in processing power between 32 and 64 bit indicates (according to specifications). It is because only a part of the kernel time is dedicated for floating point calculations, and the most part (e.g. indexing) uses integer numbers.

TABLE II

3D simulation time (in s, mean value from 10 test runs) for a sound sample 1 s long. Results allowing real time operation (below 1 s) are marked in bold.

Grid size	GeForce GT 750M	Core i7-4700HQ
$8 \times 8 \times 8$	**0.072**	**0.315**
$12 \times 12 \times 12$	**0.14**	1.278
$16 \times 16 \times 16$	**0.24**	2.616
$20 \times 20 \times 20$	**0.53**	3.998
$24 \times 24 \times 24$	1.1	7.165
$28 \times 28 \times 28$	2.652	10.627
$32 \times 32 \times 32$	2.981	21.048

The most interesting case is the 3D simulation. The test procedure was the same as the one used for comparing V1 and V2 algorithm (10 test runs for each grid size, 1 s wave buffer calculated). Grids ranged from $8 \times 8 \times 8$ to $32 \times 32 \times 32$ with the increment of 4. The results are presented in Table II and Fig. 6.

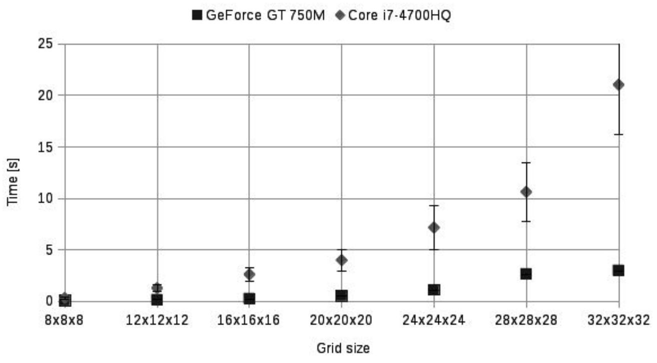

Fig. 6. Comparison between GPU and CPU in 3D algorithm.

GPU can work in real time with $20 \times 20 \times 20$ grid, while the CPU handles only $8 \times 8 \times 8$ grid (about 15.5 times less cells). In case of $20 \times 20 \times 20$ grid, GPU's calculation time is about 7.5 times shorter than this of CPU. Despite similar algorithms, performance gap between the devices in case of 3D grid is much larger than in case of 2D grid.

6. Conclusions

A feasibility study of GPU-accelerated sound synthesis based on physical modeling implemented using OpenCL framework has been presented. Middle-class mobile (notebook) GPU performance was compared to the performance of top-class mobile CPU in a series of tests involving finite difference models of two instruments: a square membrane in 2D grid and a cubical block in 3D grid. The aim was to determine the largest 2D and 3D grid size that can be calculated in real time. Using a parallelized algorithm, the GPU performed 2.5 times faster than the CPU, allowing 128×128 FD grid to operate in real time. The difference was even larger in 3D, where GPU was 7.5 times faster than CPU, allowing the use of $20 \times 20 \times 20$ FD grid.

Synchronization issues in parallel GPU calculations were discussed and addressed. The effect of transition from 32-bit to 64-bit precision was also determined (the largest 64-bit GPU FD grid is 104×104), as well as an impact of wave buffer length used in calculations — with sampling frequency of 44100 Hz it is possible to use buffers shorter than 1 ms, which is much better than the threshold needed for real time-control of the instrument.

The results indicate that GPUs can significantly speed up real-time musical instrument simulations, allowing to develop more complex and realistic models. More powerful desktop-class GPUs should perform significantly better. Taking into account that a personal computer can be equipped with up to 4 GPUs, and even some notebooks utilize 2, the results are encouraging. OpenCL allows for simultaneous calculations on different types of devices, such as CPUs and GPUs, further increasing available processing power.

There are, however, some issues that need to be resolved before designing large, complex, real-time instrument models, spanning through a number of different compute devices. One of the more important is the problem of synchronization between threads. If parts of the instrument are to be simulated on separate devices, a method of efficient communication between those devices has to be developed, and OpenCL does not provide a simple, efficient solution here. Another one is fine-tuning the algorithm to take advantages of various types and sizes of memory accessible to different compute devices, and more generally, to tune it to the architecture of the system hardware.

Considering the fact that general-purpose computing on GPUs, or more generally — heterogeneous computing, is a rather novel technique with only a few prototype implementations in the field of sound synthesis, there is much more research to be done in this area, but also much to be gained.

References

[1] C. Roads, J. Strawn, C. Abbot, J. Gordon, P. Greenspun, *The Computer Music Tutorial*, MIT Press, Cambridge MA 1996.

[2] *What is Heterogeneous Computing?*, http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-computing/, visited: 21.07.2014.

[3] M. Sosnick, W. Hsu, in *7th Sound and Music Computing Conference Proceedings*, Eds. E. Gómez, P. Herrera, R. Ramírez, Universitat Pompeu Fabra, Barcelona 2010, p. 485.

[4] M. Sosnick, W. Hsu, in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Eds. A.R. Jensenius, A. Tveit, R.I. Godøy, D. Overholt, University of Oslo and Norwegian Academy of Music, Oslo 2011, p. 264.

[5] W. Hsu, M. Sosnick-Pérez, *ACM Queue* **11**, 40 (2013).

[6] M. Pluta, in *Proceedings of Forum Acusticum 2014*, Ed. B. Borkowski, The Polish Acoustical Society, Krakow 2014, p. 227.

[7] S.A. Van Duyne, J.O. Smith III, in *Proceedings of the International Computer Music Conference, ICMC-93*, The Computer Music Association, Tokyo 1993, p. 40.

[8] U.R. Kristiansen, E.M. Viggen, *Computational Methods in Acoustics*, Norwegian University of Science and Technology — NTNU, Department of Electronics and Telecommunications, Trondheim 2010.

[9] A.B. Adib, arXiv:physics/0009068v3, 2000.

[10] P. Filipp, A. Bergassol, D. Habault, J.P. Lefebvre, *Acoustics: Basic Physics, Theory, and Methods*, Academic Press, San Diego 1998.

[11] A. Dobrucki, *Przetworniki elektroakustyczne*, WNT, Warszawa 2007.

[12] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, in *ISCA '10 Proceedings of the 37th annual international symposium on Computer architecture*, Eds. A. Seznec, U. Weiser, R. Ronen, ACM, New York 2010, p. 451.

[13] *Nvidia CUDA Zone*, https://developer.nvidia.com/cuda-zone, visited: 21.07.2014.

[14] *OpenCL*, https://www.khronos.org/opencl/, visited: 21.07.2014.

[15] K. Karimi, N.G. Dickson, F. Hamze, arXiv:1005.2581v3, 2011.

[16] J. Fang, A.L. Varbanescu, H. Sips, in *Proceedings of the 2011 International Conference on Parallel Processing*, Eds. G.R. Gao, Y. Tseng, IACC, Taipei 2011, p. 216.

[17] V. Hindriksen, *OpenCL vs CUDA Misconceptions*, http://streamcomputing.eu/blog/2011-06-22/opencl-vs-cuda-misconceptions/, 2011, visited: 21.07.2014.

[18] *AMD OpenCL Zone*, http://developer.amd.com/tools-and-sdks/opencl-zone/, visited: 21.07.2014.

[19] *Intel SDK for OpenCL Applications*, https://software.intel.com/en-us/vcsource/tools/opencl-sdk, visited: 21.07.2014.

[20] *OpenCL — Nvidia Developer Zone*, https://developer.nvidia.com/opencl, visited: 21.07.2014.

[21] AMD Staff, *OpenCL and the AMD APP SDK v2.4*, http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/, 2011, visited: 21.07.2014.

[22] *OpenCL Tutorials 1 — Quickstart*, http://opencl.codeplex.com/wikipage?title=OpenCL%20Tutorials%20-%201, visited: 21.07.2014.