OpenCL vs: Accelerated Finite-Difference Digital Synthesis

Harri Renney
harri2.renney@live.uwe.ac.uk
Computer Science Research Centre
University of West of England
Bristol, UK

Benedict R. Gaster
benedict.gaster@uwe.ac.uk
Computer Science Research Centre
University of West of England
Bristol, UK

Tom Mitchell
tom.mitichell@uwe.ac.uk
Creative Technologies Lab
University of West of England
Bristol, UK

ABSTRACT

Digital audio synthesis has become an important component of modern music production with techniques that can produce realistic simulations of real instruments. Physical modelling sound synthesis is a category of audio synthesis that uses mathematical models to emulate the physical phenomena of acoustic musical instruments including drum membranes, air columns and strings. The synthesis of physical phenomena can be expressed as discrete variants of Newton's laws of motion, using, for example, the Finite-Difference Time-Domain method or FDTD.

FDTD is notoriously computationally expensive and the real time demands of sound synthesis in a live setting has led implementers to consider offloading to GPUs. In this paper we present multiple OpenCL implementations of FDTD for real time simulation of a drum membrane. Additionally, we compare against an AVX optimized CPU implementation and an OpenGL version that utilizes a careful mapping to the GPU texture cache. We find using a discrete, laptop class, AMD GPU that for all but the smallest mesh sizes, the OpenCL implementation out performs the others. Although, to our surprise we found that optimizing for workgroup local memory provided only a small performance benefit.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → *Compilers*.

KEYWORDS

Finite-Difference, SIMD, OpenCL, OpenGL

ACM Reference Format:
Harri Renney, Benedict R. Gaster, and Tom Mitchell. 2019. OpenCL vs: Accelerated Finite-Difference Digital Synthesis. In *Proceedings of IWOCL '19: The International Workshop on OpenCL (IWOCL '19).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

Figure 1: Audio Unit (AU) plugin

1 INTRODUCTION

The physical synthesis of sound is the general process of using mathematical models to simulate a physical source of sound. The technique of simulating sound using mathematical models, although not the earliest example, was first made popular by Karplus and Strong, using a method of physical modelling synthesis that circulates a short waveform through a filtered delay line to simulate the sound of a hammered or plucked string [12]. The algorithm was later extended by David and Smith [13] and remains in use today, due in part to its low computational footprint.

While the Karplus-Strong algorithm remains popular in the domain of real time synthesis, it does not accurately represent the way in which vibrations propagate through a medium, e.g. a drum membrane, and alternative models have been proposed that address these shortcomings[1]. For example, finite difference approximation is a common method to simulate the movement of waves through physical mediums, presented very early on in the area of acoustics and synthesis, e.g. Hiller and Ruiz studied using finite difference methods for sound synthesis in the early 1970s [10].

To simulate vibrations moving through a material, we can utilize Newton's laws of motion, describing the movement using Ordinary Differential Equations (ODEs). For these to be implemented as a discreet algorithm, ODEs are expressed as Finite Discreet Time-Domain (FDTD) equations. FDTDs are used as the basis of numerous physical modelling efforts that seek to digitally synthesize the sounds of, for example, drum membranes [16], wind instruments [3] and strings [8].

[1]It is worth noting that in the end, most sound synthesis is performed in the context of music composition and as such the need to sound like a "real" drum or some other form of instrument is subjective.

Of course, it is well known that the real-time simulation of FDTD is hard, it requires a large amount of floating point computation. Consider, for example, the case of simulating a drum membrane with a mesh of 32x32 at an audio sample rate of 48kHz. Each point in the domain requires 40 floating point instructions per sample and thus approximately 2 Giga FLOPS of compute for real time synthesis. Of course, in a real audio application, any particular sound engine must compete for resources with, synthesis, effects, mixing, and so on. To this end, acceleration of FDTDs on GPUs has been proposed as a method to offload the simulation of drum membranes and other physical models of musical phenomena. For example, Sosnick and Hsu describe a straightforward implementation using NVIDIA's Cuda [16], while Zappi et al use OpenGL [20]. In this paper we again step into the breach to use GPUs to accelerate FDTD, for real time audio synthesis, initially utilizing OpenCL[2] [9], but we then go further presenting a comparison of implementations ranging from a serial CPU implementation, an SOA AVX variant, a recreation of Zappi et al's OpenGL implementation, and two OpenCL versions, one using only global memory, and one using workgroup local memory. We find that on AMD hardware, in all but the smallest grids, OpenCL outperforms the other implementations, and the use of workgroup local memory provides little to zero benefit.

We have implemented two variants of our drum simulation. The first implementation uses our OpenCL and CPU implementations and is provided as a Apple Audio Unit (AU) [1] plugin, that can be loaded into a Digital Audio Workstation (DAW) (e.g. Ableton Live, Logic Pro, and so fourth). A screen shot of the plugin is given in Figure 1 and includes controls that change properties of the drum membrane, such as the speed at which sound travels through the material. While not directly relevant to how the FDTD was optimized, the focus of this paper, it played an important role within the context of the work as a whole, enabling practicing musicians to utilize the drum within their standard work-flow and to provide feedback on the sound quality—it is of little use to provide real time synthesis for a drum that sounds terrible!

The second implementation is focused on providing a simple test framework in which the OpenCL, OpenGL, and CPU variants can be compared. The simulations are controlled from (JSON) configuration files and are fully automated. For the most part the framework for each implementation is the same, however, for the OpenGL version we also support a simple visualization of the drum membrane.

The OpenGL version's ability to generate visual feedback is shown in Figure 2. This figure simply takes the pressure points of the current implementation and uses it to calculate a colour gradient, using an additional `drawcall`. The red dot is the positioning of the microphone and is the point that is sampled for the audio output. The yellow square is the excitation point, i.e. the point where an excitation function is fed to the membrane—in general, it would not be placed in the centre of the mesh, but for simplicity it is located there for demonstration.

[2]Throughout this paper we use OpenCL as shorthand for OpenCL 1.2 and do not consider OpenCL 2.x.

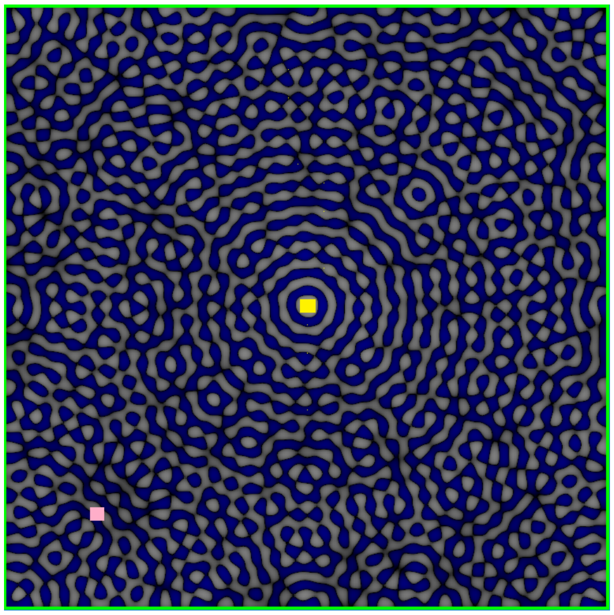

Figure 2: OpenGL Visual rendering of 2D membrane

We conclude this introduction by outlining the remainder of the paper:

(1) Section 2 provides a short overview of related and existing work;
(2) Section 3 introduces the FDTD equations, describing implementation as pseudo code, and then details the CPU, OpenCL, and OpenGL implementations;
(3) The benchmark results for the different implementations are presented in Section 4; and
(4) Finally, Section 5 concludes and provides pointers to future work.

2 RELATED WORK

To our knowledge there are no OpenCL implementations of FDTD that have been specifically developed for digital audio synthesis. As we would expect, there have been a selection of implementations of 3D FDTDs. For example, in the context of electromagnetic wave interaction, Cannon and Honary present an implementation in OpenCL [7]. They demonstrate good speedup, including multiple GPUs, however, unlike our work they fail to consider alternatives, such as OpenGL or AVX and focus on HPC style platforms and highend GPUs (Nvidia Tesla M2075), utilizing a large number of SIMD units and DDR4 memory. It is worth noting that due to the problem they are tackling, the size of meshes they consider are much larger than those we present, which has the potential to provide an easier context for acceleration, when comparing to single threaded CPU performance.

Although FPGAs are a good way of processing FDTD grids [18], the focus of this work has been towards commodity hardware, in particular, live digital musician will tend towards using a medium to highend laptop, with an external audio device to handle input and output to the analog domain. To date easy to access FPGAs is not accessible on a commodity scale within this domain.

Returning the focus to 2D FDTDs for digital audio synthesis for GPUs a number of researchers have studied this area, for example [4, 6, 16, 20]. In particular, Sosnick and Hsu [16] implement a simple 2D membrane simulation using a FDTD written in CUDA. However, there results are limited, testing only on small meshes and single buffer size. They also to not consider other optimized implementations, such as AVX CPU or other GPU programming models such as OpenGL or OpenCL.

Zappi et al [20] implement an FDTD simulation of a drum head model very similar to ours. They use a novel approach to packing the previous, current, and next step simulation data within a single RGBA texture. The OpenGL implementation used in the benchmarks presented in Section 4 uses a variant of their design. While it performs well it fails, by a small amount, to out perform our OpenCL implementation. This leaves us to hypothesize that while single texture encoding is interesting, it might in fact lead to a less cache friendly algorithm.

3 IMPLEMENTATION

In this section we describe the FDTD algorithm, firstly as equations, then as pseudo code. This is then followed by details of each implementation. The example used to demonstrate the implementation is the OpenCL global memory kernel, and, although the other implementations differ in details regarding the particular target language, they are for the most part similar. In the case were they differ, e.g. using a packed texture for the OpenGL implementation a discussion is included to outline important aspects. The full source code for each implementation used within the Section 4, outlining the benchmark results, can be found here [15].

3.1 The FDTD Algorithm

In this subsection, the numerical algorithm used for modelling the drum membrane is described. It follows the discretization of the standard 2D wave propagation equation [17].

$$p^{n+1} = \frac{2p^n + \left(\mu - 1\right) p^{n-1} + \alpha \left(p_l + p_r + p_u + p_d - 4p^n\right)}{\mu + 1} \tag{1}$$

$$P_{L,R,U,D} = \begin{cases} p^n \gamma & \text{if } n \text{ boundary} \\ p^n_{l,r,u,d} & \text{if } n \text{ free} \end{cases} \tag{2}$$

where:

- p^{n+1} is the pressure point to be calculated for the next time step, $n+1$.
- p^n is the pressure point of current time step, n.
- p^{n-1} is the pressure point of the previous time step, $n-1$.

```
for i = 0 to bufferSize
  for row = 1 to gridHeight
    for column = 1 to gridWidth
      centrePoint = getPoint(row,column)
      if(centrePoint == boundary)
        neighbours = calculateBoundary(centrePoint)
      else
        neighbours = getNeighbours(centrePoint)
    compute(centrePoint, neighbours)
    end for
  end for
  rotateGrids()
end for
```

Figure 3: Pseudocode for FDTD membrane simulation.

- μ is the damping/absorption coefficient of the modelled material. $(0.0 < \mu < 1.0)$, for all values of μ.
- α is the propagation factor. $(\alpha \leq 0.5)$, for all values of α.
- $p_{l,r,u,d}$ are the pressure points for the neighbouring values of the centre point currently under consideration.
- γ is the centre points boundary gain. This is the degree at which the pressure is reflected back into the grid. $(0.0 < \gamma < 1.0)$, for all values of γ.

This equation is used to simulate wave propagation across the 2D surface when applied to all grid points. Every time step, the equation is used to calculate the next pressure value of the currently considered grid point from the current, previous and neighbouring pressure values.

The neighbouring values are determined by the centre points boundary value (See equation 2). If the centre is not a boundary point, then the actual neighbouring pressure points are taken for $P_{L,R,U,D}$. If it is a boundary point, then the neighbour pressure values are not used and instead the centre pressure point multiplied by the boundary gain γ is used in place of the neighbour values in the equation.

α, the propagation factor determines the speed at which sound passes through the medium. It is formed from the speed of sound, the sample rate and the size of each grid point. Therefore when modelling some material, the size of the grid and the sample rate affect the speed at which waves are simulated to pass through the material if the propagation factor is not adjusted.

3.2 Pseudo Code

The pseudocode for implementing the previous equations is given in Figure 3 and outlines the sequential method of calculating the FDTD grid. This code is a straightforward way to compute a 2D FDTD grid, of any size. It works by visiting each point in the grid and calculating the next pressure value using the compute function. This basically applies equation 3.1 to generate the next time step pressure value. The grid of $n+1$ is updated with the new value.

Once the whole grid has advanced one time step by setting the centre point to the new computed point, the grids are advanced too, by rotating the grids. Aligning them correctly ready for the next time step.

3.3 OpenCL Kernel

Figure 4 is the source code for the OpenCL global memory variant of our FDTD implementation. For ease of presentation the local memory variant is not presented in this paper, but the interested reader can find the code for the all the implementations on our Gitlab repository [15].

Note how the rotation index is used to define the FDTD pointers to each grid. By incrementing the index, the address to which the pointers are set to shifts along to the next grid, simulating the advancement of a time-step. It is important to note that the rotation index is not incremented within the kernel, but in the host application between kernel calls. As a consequence, an explicit synchronization barrier is necessary between each OpenCL "ndrange" dispatches, thus, kernel calls cannot be batched. An alternative would be to utilize a kernel to increment the rotation index in between FDTD kernels; this would allow batching and avoid host side synchronization and copying of the rotation index. To date we have not felt this necessary, but it is likely that as we optimize further this will become necessary.

3.4 Implementations

This section provides an overview of each of the different implementations. The implementations, which are outlined in the following subsections, all share an FDTD grid class that stores the locations of the excitation and listener points, along with the pressure and boundary grids as a flattened 2D Structure of Arrays (SOA) data structure. Although the underlying kernels for each of the different implementations vary in design, with the goal of utilizing the different optimizations opportunities provided by each programming model.

3.4.1 CPU Serial. The serial implementation works by visiting each grid point and calculating the new pressure value using the equation 1, then moving onto the next cell sequentially. The whole grid must be calculated before one sample can be obtained. A few optimizations were used to avoid needless computation like cache alignment, avoiding copies and redundant calculations.

Cache alignment is achieved by using flattened 2D arrays of the grids. This ensures the next item in the array is usually held in the same cache line, even at the ends of the grid rows. For each time step, the pressure grids need to increment along in time. Take a look at Figure 5. After the newly calculated grid of N+1 pressure values is complete, a pointer which determines the current pressure grid addresses the N+1 grid. All the pointers shift forward one to correctly address the new set of pressure grids. This is done rather than copying all the data between the grids which would be highly inefficient.

Redundant calculations are removed including only calculating $\mu - 1$ once for calculating the next pressure point. μ can change, therefore it should still be calculated once per buffer fill, but it is not necessary to calculate every cell visited.

3.4.2 CPU AVX. The AVX optimized CPU implementation follows after most of what the serial version does. However, it vectorizes the grids for processing. Using AVX SIMD intrinsics, the grid can be vectorized into vectors of four floats. These vectors are processed and stored, using Structure of Arrays, in parallel. Intel's compiler intrinsics [2] were used to do an explicit vectorization of FDTD computation, rather than using compiler pragmas or a C++ SIMD library, for example.

Although the grid calculations are applied universally, the checks for boundary, excitation and listener points cause each element in the vectors to be checked, often using shuffles. This can likely be improved, with further considerations for packing certain data bits, but we leave this to further work.

Additionally, in future work we intend to extend the SIMD vectorization to support AVX-256/512. This would effectually double the vector size and therefore in ideal circumstances, would result in doubling of performance.

3.5 OpenCL global memory kernel

The OpenCL versions start by initializing the FDTD grid and the OpenCL configuration on the CPU. Then, when a buffer of samples is to be computed, an excitation buffer is loaded onto the GPU. Every iteration, the kernel is called which calculates the next time step and produces an output sample. The output samples remain on the GPU in a sample buffer. Only once the buffer is full is it read back by the host application on the CPU to minimise transfer overheads associated with passing data back and forth.

As with the serial implementation described earlier, the method for rotating pressure grids with pointers is also used here. At each time step, the GPU uses an index value, incremented by the CPU, to determine which grids the pointers address. In the OpenCL global version, the grids are held in the GPUs global memory and no local memory caching is performed.

3.5.1 OpenCL workgroup local kernel. An OpenCL version almost identical to the previous one was developed, but using the workgroup local memory to store the current pressure grid. In the kernel before any calculations are made the workitems load the current pressure value into a local grid accessible by all workitems in the same workgroup. This means when the neighbouring values of the centre point are needed, they can be fetched from the local grid which has shorter access times than the global grid. There are cases on the edge of workgroups where a work item will need to access a neighbour outside the workgroup, see figure 6. Therefore, conditional checks are made and if the neighbour is outside the workgroup, it will need to be fetched from global memory. This technique has been used in optimized convolution kernels, see for example [14]. These conditionals are suspected to impact any performance gained from using the local memory.

```
__kernel
void ftdtCompute(__global float* gridOne, __global float* gridTwo, __global float* gridThree,
  __global float* boundaryGain, int samplesIndex, __global float* samples, __global float* excitation,
  int listenerPosition, int excitationPosition, float propagationFactor,
  float dampingFactor, int rotationIndex) {
    // get index for current and neighbouring nodes
    int ixy = (get_global_id(1)) * get_global_size(0) + get_global_id(0);
    int ixMy = (get_global_id(1)-1) * get_global_size(0) + get_global_id(0);
    int ixPy = (get_global_id(1)+1) * get_global_size(0) + get_global_id(0);
    int ixyM = (get_global_id(1)) * get_global_size(0) + get_global_id(0)-1;
    int ixyP = (get_global_id(1)) * get_global_size(0) + get_global_id(0)+1;

    // determine each buffer in relation to time from a rotation index//
    __global float* nMOne;   __global float* n;  __global float* nPOne;
    if(rotationIndex == 0) {
      nMOne = gridOne;
      n = gridTwo;
      nPOne = gridThree;
    } else if(rotationIndex == 1) {
      nMOne = gridTwo;
      n = gridThree;
      nPOne = gridOne;
    } else if(rotationIndex == 2) {
      nMOne = gridThree;
      n = gridOne;
      nPOne = gridTwo;
    }
    // initialize pressure values//
    float centrePressureNMO = nMOne[ixy];
    float centrePressureN = n[ixy];
    float leftPressure;  float rightPressure;  float upPressure; float downPressure;

    if(boundaryGain[ixy] > 0.0) {
      leftPressure = n[ixy] * boundaryGain[ixy];
      rightPressure = n[ixy] * boundaryGain[ixy];
      upPressure = n[ixy] * boundaryGain[ixy];
      downPressure = n[ixy] * boundaryGain[ixy];
    } else {
      leftPressure = n[ixMy];
      rightPressure = n[ixPy];
      upPressure = n[ixyM];
      downPressure = n[ixyP];
    }
    // calculate next pressure value
    float newPressure = 2 * centrePressureN;
    newPressure += (dampingFactor - 1.0) * centrePressureNMO;
    newPressure += propagationFactor * (leftPressure + rightPressure +
                                        upPressure +  downPressure - (4 * centrePressureN));
    newPressure *= 1.0 / (dampingFactor + 1.0);

    // if the cell is the listener position, sets the next sound sample in buffer to value contained here
    if(ixy == listenerPosition) {
       samples[samplesIndex] = n[ixy];
    }
    if(ixy == excitationPosition) { // if the position is an excitation...
      // input excitation value into point. Then increment to next excitation in next iteration.
      newPressure += excitation[samplesIndex];
    }
    // update grid plus one
    nPOne[ixy] = newPressure;
}
```

Figure 4: OpenCL FDTD Kernel

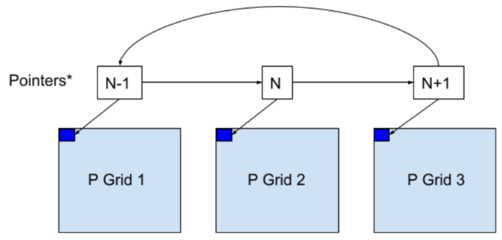

Figure 5: Pressure grid rotation achieved using pointers

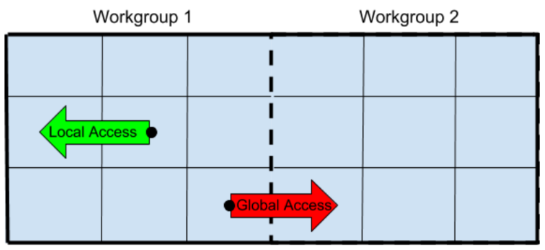

Figure 6: Neighbour access within workgroup and across workgroups.

3.6 OpenGL

At initialization, the same FDTD grid object as the earlier methods is created, but then, translated into a graphical context in order to be understood by OpenGL. This translation involves forming a texture buffer, which is loaded in and out of the GPU and processed according to the defined GLSL shaders. This formats the grid data into RGBA channels in the graphics fragment shader, where the FDTD calculations takes place.

An advantage of using OpenGL is that a render shader program can be used for visualizing the FDTD grid in its current state. Of course, OpenCL/OpenGL interop could be used, but this has a performance cost, due to different driver stacks. An alternative approach that we are planning to investigate is to use Khronos' Vulkan [19], with a combination of compute and graphic shaders.

4 RESULTS

In this section we outline the results obtained when executing the serial CPU, AVX CPU, OpenCL, and OpenGL implementations of the 2D FDTD simulation of a simple drum membrane. The results compare a variety of different mesh sizes, audio buffer sizes, and, for OpenCL, different workgroup optimizations, including workgroup size and workgroup local memory utilization. All speedup measurements are calculated in relation to the results of the naive, serial CPU implementation. Although it might not seem fair to compare massively parallel processing to a serial one, this was done to show the instant benefit by considering using a GPU for processing these kinds of problems. Clearly with a little more work, the CPU can still be improved. This can be seen to an extent with the CPU AVX version.

System Specifications	
CPU	Intel Core i7-8550U 4 Cores 1.99GHZ
GPU	AMD Radeon 530 with 2GB GDDR5
CPU RAM	8GB 2400MHz DDR4

Table 1: Laptop specification used for benchmarking

	OpenCL Global	**OpenCL Local**
Workgroup Size	time (ms)	time (ms)
4x4	136.072666	149.841333
8x8	46.170966	49.19716
16x16	46.5292	45.363366

Table 2: Mean buffer compute time calculated over 100 filled buffers. Buffer size = 512, Grid Size = $256x256$

The benchmarks are run on a single test machine, which is a mid-range PC laptop, with a discrete AMD GPU and Intel i7. The complete specification of the test setup is given in Table 1.

Table 2 compares the different workgroup sizes configurable in OpenCL. The workgroups represent the number of work items that can be processed concurrently. In this case, work items are equivalent to computation of each pressure point in the grid.

Table 2 is using a larger grid size of $256x256$. There is little difference between the results of the smaller grid dimensions (e.g. $8x8$ and $16x16$) as the $16x16$ workgroup size does not make use of the increased capacity of concurrent work items.

The maximum workgroup size used for benchmarking the system's GPU is 256 ($16x16 = 256$), corresponding to an equal axis grid arrangement of $16x16$. Our initial tests concluded that smaller arrangements were less efficient so the largest workgroup dimensions were used for all the following tests $16x16$.

Figure 7 plots the compute time in milliseconds for each implementation with a buffer size of 512 samples.

Here a collection of different sample buffer sizes were tested. Taking Figure 11 specifically, it can be seen that although the AVX implementation starts with a good speedup. It cannot handle the increasing grid size and plateaus at 1-2 times speedup. The GPU versions are slower for processing smaller grids. This is likely due to the transfer overhead between CPU-GPU communication. Although slower for small grids, the GPU versions outperform the CPU versions as the grid scales. The OpenCL versions continue to speedup as it scales. Interestingly, the OpenGL version does not reliably increase. To date we have not established why this is the case, but hypothesize that it might well be down to the constraints and perhaps redundant steps involved in the graphics pipeline. More likely the smart single texture encoding proposed by Zappi et al [20] might actually be causing the slow down, due

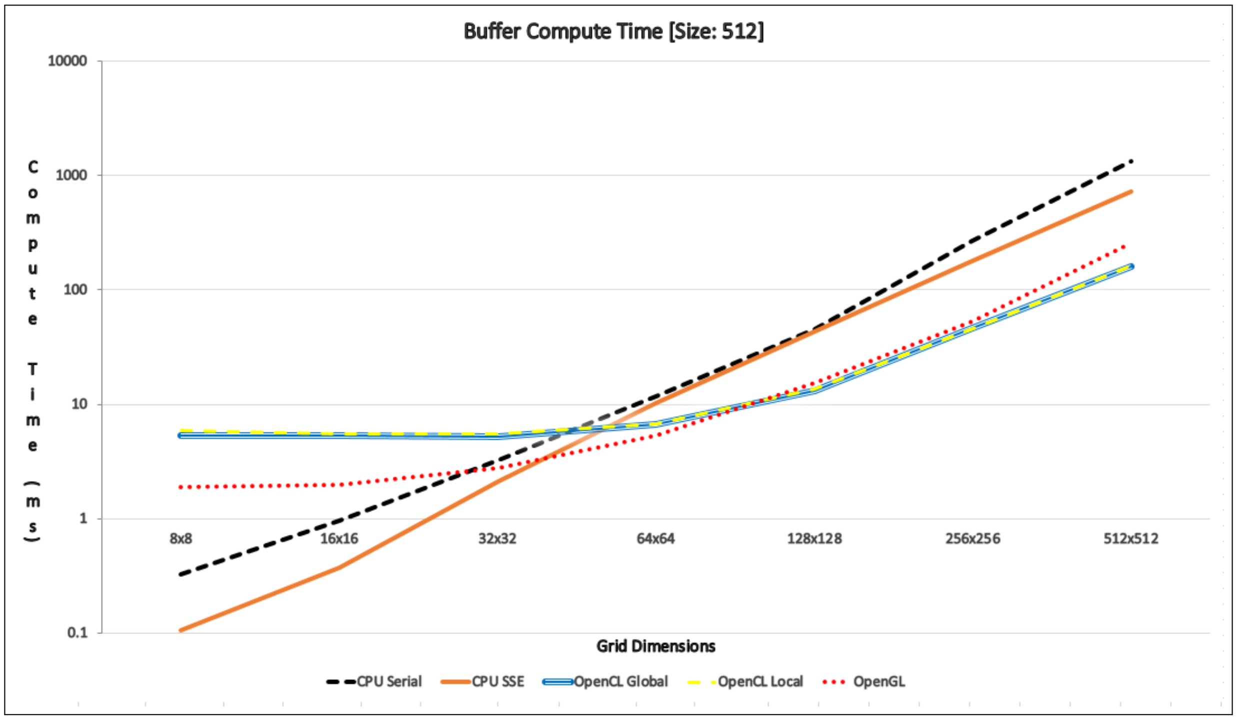

Figure 7: Average compute time measured for buffer size 512

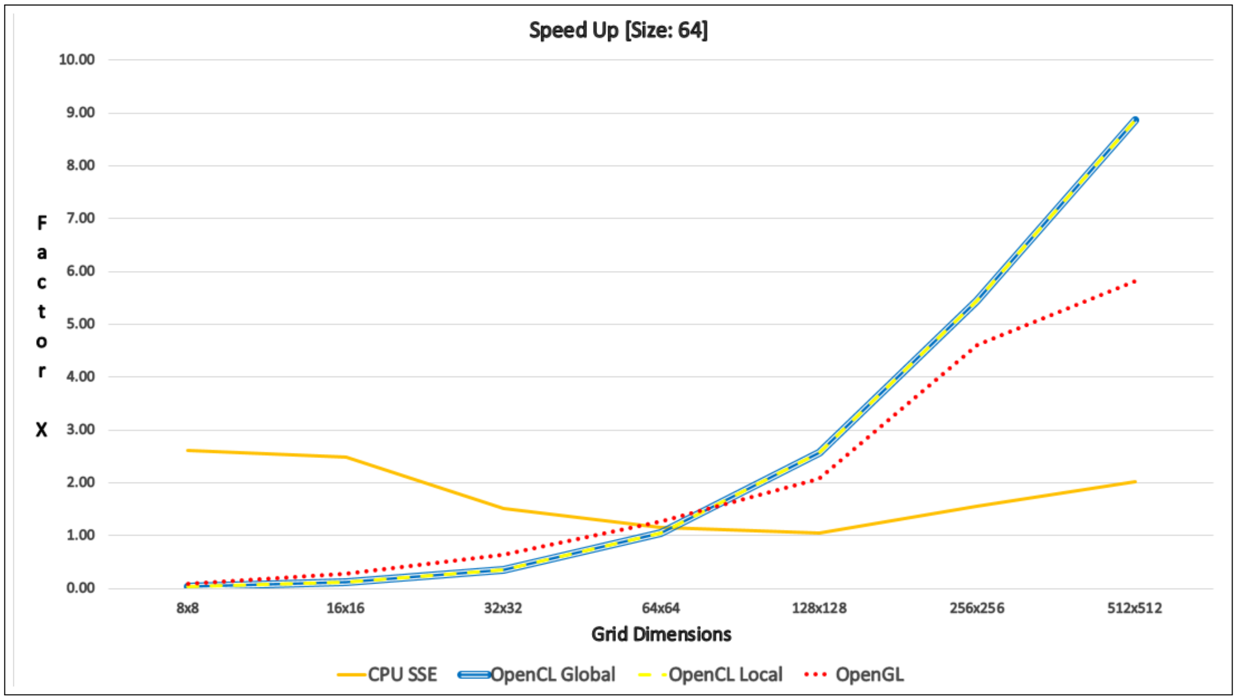

Figure 8: Speedup measured for buffer size 64

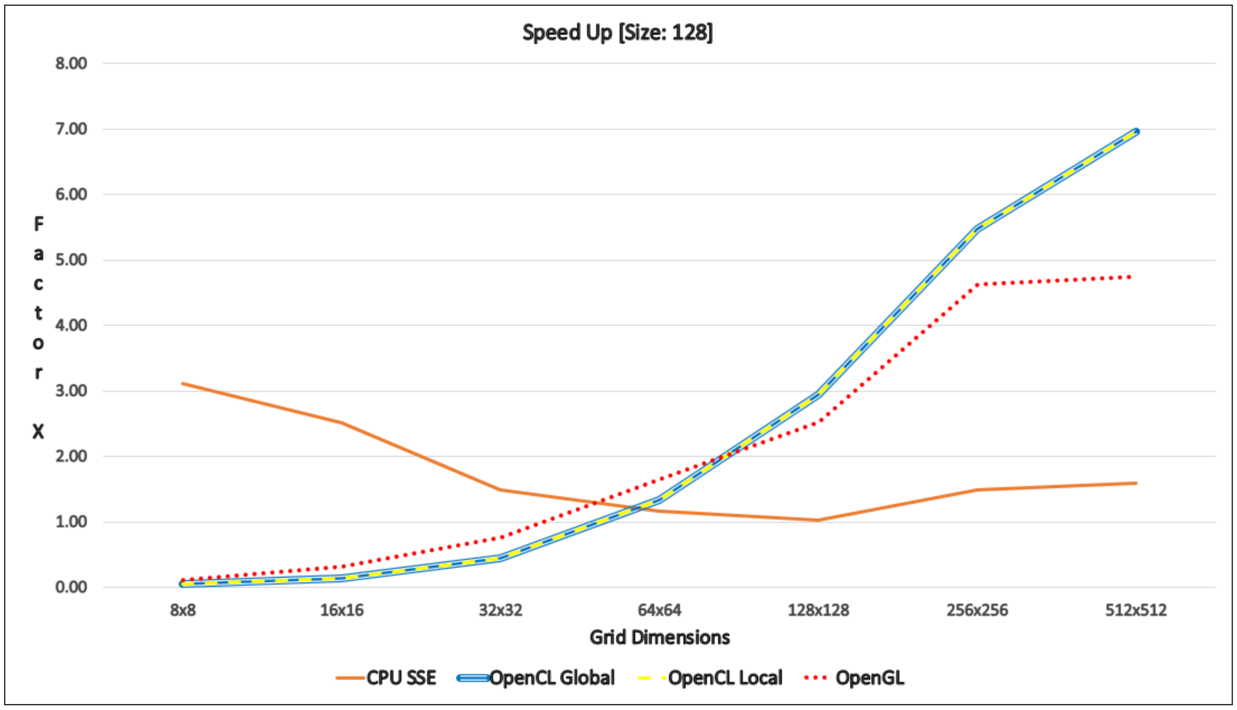

Figure 9: Speedup measured for buffer size 128

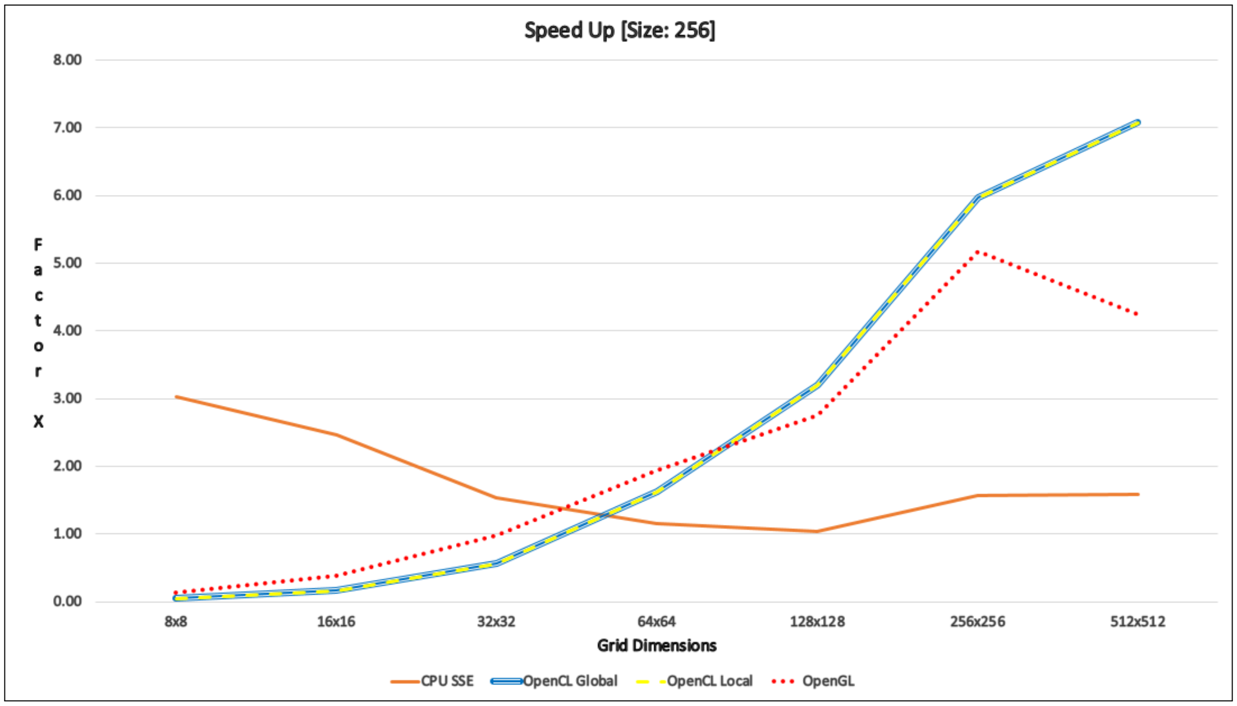

Figure 10: Speedup measured for buffer size 256

	CPU Serial		**CPU SSE**		**OpenCL Global**		**OpenCL Local**		**OpenGL**	
Dimensions	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup
8x8	0.3247	1.0	0.105525	3.08	5.30303	0.06	5.96503	0.06	1.899273	0.17
16x16	0.954201	1.0	0.377217	2.53	5.29219	0.18	5.48204	0.18	1.97544	0.48
32x32	3.27136	1.0	2.12098	1.54	5.230725	0.63	5.5009	0.63	2.75727	1.19
64x64	11.7197	1.0	10.1995	1.15	6.62889	1.77	6.7361	1.77	5.337273	2.20
128x128	45.8261	1.0	43.01075	1.07	13.21315	3.47	13.672	3.47	15.256833	3.00
256x256	271.803	1.0	178.2885	1.52	46.5482	5.84	45.4415	5.84	53.18915	5.11
512x512	1327.78	1.0	726.5175	1.83	162.295	8.18	160.995	8.18	257.278	5.16

Table 3: Mean buffer compute time calculated over 100 filled buffers. Buffer size = 512

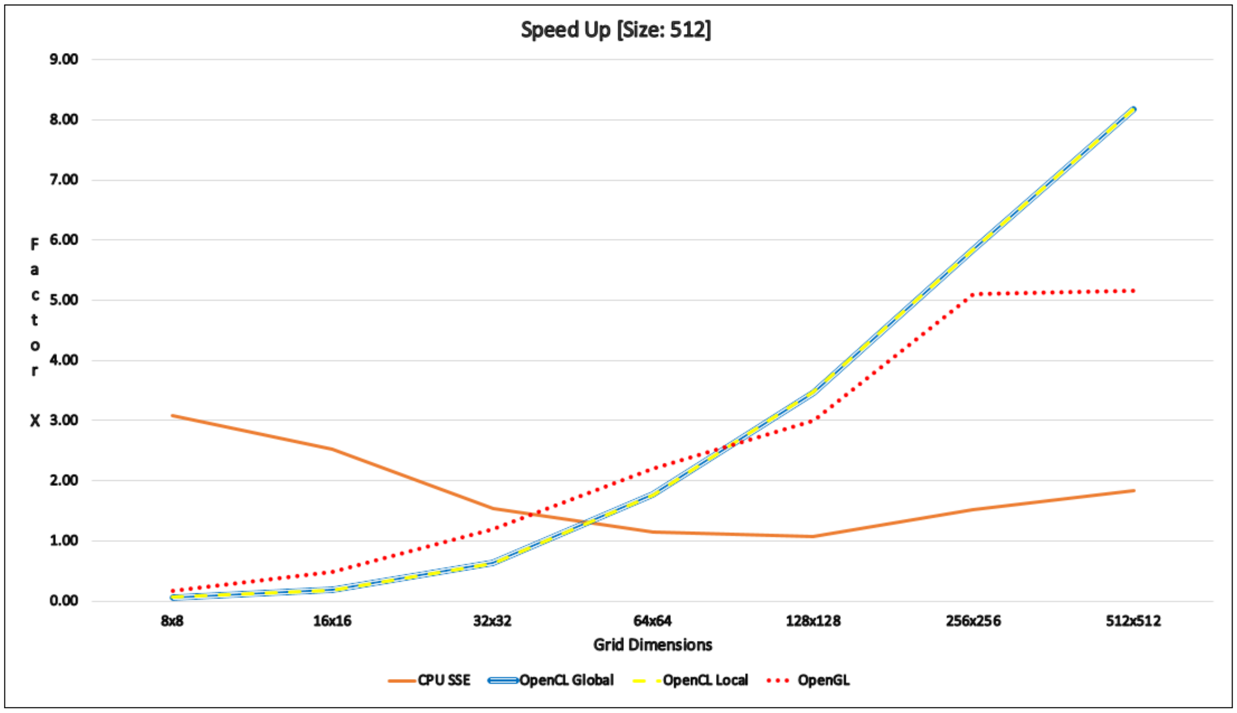

Figure 11: Speedup measured for buffer size 512

to read/write conflicts to the same buffer for different time steps. It may well be that, as in the OpenCL implementation, separating these into their own texture, enabling only the next time step texture to be write-only, would provide better memory access performance. However, this is speculation and we leave this analysis to future work.

It can be seen across the benchmark results that for smaller grid sizes, e.g. 8x8, 16x16 and 32x32, the CPU versions perform well in comparison to the GPU versions. This is likely caused by the overhead involved with data transfer to and from the GPU and, in general, keeping the host and GPU in sync. This outweighs the benefit from the acceleration provided by the compute kernel itself. To confirm this would require further micro-benchmarking. However, from dimensions 64x64 on wards, the GPU versions perform increasingly better than the CPU versions as the benefits from processing on the GPU exceed the data transfer overhead. By 512x512, the OpenCL version scales to around 8 times faster than the original serial implementation. The results suggest this would continue to increase further with larger grids.

The results in the OpenCL global and local caching version were not significantly different. Taking a look at Table 3, at the higher dimensions the local caching version is a couple of milliseconds faster. This would not be a noticeable performance increase. The caching method used in the implementation was basic and could be improved with more advanced techniques, which may produce faster computation.

5 CONCLUSION AND FUTURE WORK

We have described implementations for a physical model simulation of a drum membrane in C++, AVX, OpenGL, and OpenGL, providing an analysis and performance of the

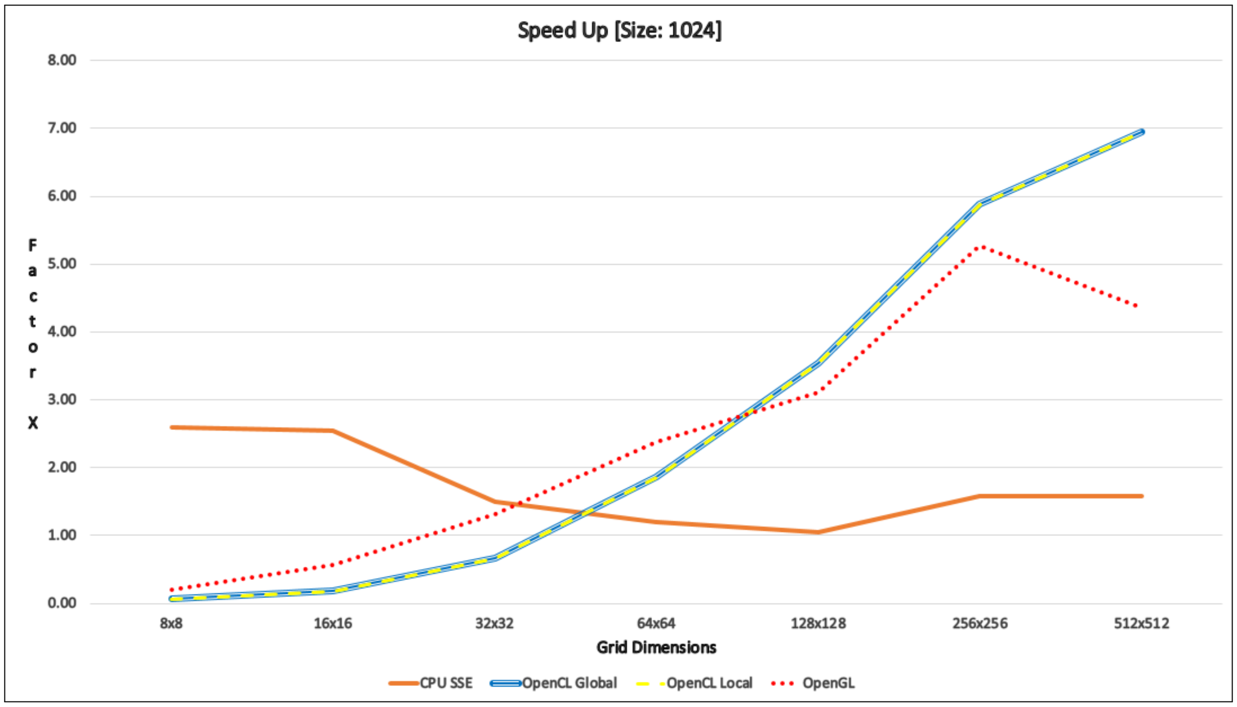

Figure 12: Speedup measured for buffer size 1024

different algorithms. The results found that as mesh size increased, the performance of OpenCL, on a particular Intel (CPU)/AMD (GPU) test system, surpassed the others, at times by a factor of eight over the serial CPU and two compared to OpenGL.

At the time of writing, the results are limited to a single platform and we plan to extend to others, including GPUs from other manufactures and also different CPUs. It would be interesting to look at the performance in the context of mobile devices, e.g. iOS and Android, as these platforms are becoming increasingly popular with musicians.

Of particular interest for future work, is targeting modern graphics APIs, such as Apple's Metal 2 and Khronos' Vulkan [5, 19]. Both of these APIs provide advanced compute capabilities that in some cases surpass that of OpenCL 1.2, e.g. Vulkan 1.1's subgroups extension, while retaining close integration with their graphics components.

Finally, our current CPU implementation is limited to only using 128-bit SIMD, while recent versions of AVX support 512-bit SIMD [11] and scatter/gather memory operations, which are likely to provide significant benefits on laptop class Intel i9 processors, as found in the latest Apple MacBook Pro models, for example. Further, OpenCL can be used to target CPUs and measurements can be taken to show how well it utilizes the CPUs parallel capabilities.

ACKNOWLEDGMENTS

We would like to thanks our colleagues in the Computer Science Research Centre and Creative Technology Lab for useful discussions and feedback on this work. In particular, Nathan Renney and David Creasey provided useful insights into physical modeling for digital synthesis. This work was supported in part by a grant from the HSA Foundation and the UWE 50-50 PhD fund.

REFERENCES

[1] [n. d.]. Apple's AudioUnit. https://developer.apple.com/documentation/audiounit?language=objc. Accessed: 2019-01-24.
[2] [n. d.]. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide. Accessed: 2019-01-23.
[3] Andrew Allen and Nikunj Raghuvanshi. 2015. Aerophones in Flatland: Interactive Wave Simulation of Wind Instruments. *ACM Trans. Graph.* 34, 4, Article 134 (July 2015), 11 pages. https://doi.org/10.1145/2767001
[4] Andrew Allen and Nikunj Raghuvanshi. 2015. Aerophones in flatland: Interactive wave simulation of wind instruments. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 134.
[5] Inc. Apple. [n. d.]. Metal. https://developer.apple.com/metal/. Accessed: 2019-01-25.
[6] Stefan Bilbao and Maarten van Walstijn. 2005. A Finite difference plate Model.. In *ICMC*.
[7] Patrick D. Cannon and Farideh Honary. 2015. A GPU-Accelerated Finite-Difference Time-Domain Scheme for Electromagnetic Wave Interaction With Plasma. *IEEE Transactions on Antennas* (2015).
[8] Cumhur Erkut and Matti Karjalainen. 2002. Virtual strings based on a 1-D FDTD waveguide model: Stability, losses, and traveling waves. In *Audio Engineering Society Conference: 22nd International Conference: Virtual, Synthetic, and Entertainment Audio*. Audio Engineering Society.
[9] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. 2012. *Heterogeneous computing with openCL: revised openCL 1.* Newnes.
[10] L. Hiller and P. Ruiz. 1971. Synthesizing Musical Sounds by Solving the Wave Equation for Vibrating Objects. *Journal of the Audio Engineering Society* (1971).

[11] Intel. [n. d.]. Intel Advanced Vector Extensions 512 (Intel AVX 512). https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/avx-512-overview.html. Accessed: 2019-01-25.

[12] Kevin Karplus and Alex Strong. 1983. Digital Synthesis of Plucked-String and Drum Timbres. *Computer Music Journal* 7, 1 (1983), 43–55.

[13] Kevin Karplus and Alex Strong. 1983. Extensions of the Karplus-Strong Plucked String Algorithm. *Computer Music Journal* 7, 2 (1983), 56–69.

[14] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 54, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014977

[15] Harri Renney. [n. d.]. CPU, AUX, OpenCL, and OpenGL FDTD implementation source code. https://gitlab.uwe.ac.uk/Physical-Modeling/iwocl-fdtd-benchmarking. Accessed: 2019-01-25.

[16] Marc Sosnick and William Hsu. 2010. Efficient finite difference-based sound synthesis using GPUs. In *Proceedings of the Sound and Music Computing Conference*. 42–44.

[17] Maarten Van Walstijn and Konrad Kowalczyk. 2008. On the numerical solution of the 2D wave equation with compact FDTD schemes. *Proc. Digital Audio Effects (DAFx), Espoo, Finland* (2008), 205–212.

[18] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. 2016. FPGA-based deep-pipelined architecture for FDTD acceleration using OpenCL. In *15th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2016, Okayama, Japan, June 26-29, 2016*. 1–6. https://doi.org/10.1109/ICIS.2016.7550742

[19] Vulkan working group. [n. d.]. Vulkan 1.1. https://www.khronos.org/vulkan/. Accessed: 2019-01-24.

[20] Victor Zappi, Andrew Allen, and Sidney Fels. 2017. Shader-based Physical Modelling for the Design of Massive Digital Musical Instruments. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 145.