Journal of New Music Research

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/nnmr20

Parallel computation of time-varying convolution

Victor Lazzarini

To cite this article: Victor Lazzarini (2020) Parallel computation of time-varying convolution, Journal of New Music Research, 49:5, 403-415, DOI: 10.1080/09298215.2020.1810280

To link to this article: https://doi.org/10.1080/09298215.2020.1810280

Published online: 26 Aug 2020.

JOURNAL OF NEW MUSIC RESEARCH
2020, VOL. 49, NO. 5, 403–415
https://doi.org/10.1080/09298215.2020.1810280

Routledge
Taylor & Francis Group

Parallel computation of time-varying convolution

Victor Lazzarini

Department of Music, Maynooth University, Maynooth, Ireland

ABSTRACT

This paper introduces a method for computing the time-varying convolution in parallel. It discusses the motivations for this approach, detailing the limitations with the current serial implementation. A detailed review of the signal processing involved is presented, describing the time-varying filter as a modification of the time-invariant case. This is followed by description of the parallel method, which is then implemented in the Open Computing Language. An analysis of tests result is provided, detailing the improvements on the existing approach and noting the cases where it is not the most suitable option.

ARTICLE HISTORY
Received 12 October 2019
Accepted 7 July 2020

KEYWORDS
Computer music; musical signal processing; time-varying filters; OpenCL

1. Introduction

Parallel computation has become an important topic in musical signal processing. Applications, such as additive and spectral modelling synthesis (Savioja et al., 2010; Tsai et al., 2010), room acoustics modelling (Hamilton & Webb, 2013; Roeber et al., 2007; Savioja, 2010), the sliding phase vocoder (Bradford et al., 2011), linear time-invariant convolution (Belloch et al., 2011), filtering (Belloch et al., 2013), binaural audio (Belloch et al., 2018), and other types of audio signal processing (Savioja et al., 2011), have been proposed as a way of harnessing the resources of computing devices such as general-purpose graphics processing units. In particular, we have observed that spectral processing can be implemented very efficiently with parallel algorithms (Crespi, 2016; Lazzarini et al., 2014), due to the particular nature of the data being manipulated. This provides the initial motivation for the present work, in which frequency-domain operations are predominant.

Time-varying convolution is a newly proposed audio processing algorithm, which has found many novel music performance uses (Brandtsegg et al., 2018). It involves the cross-synthesis of two arbitrary input signals in a very transparent manner, relying on signal content with no need for parametric controls. Due to these characteristics, it has been found to be particularly amenable to cross-adaptive applications. Depending on some of the conditions, in its current form, based on serial computation, this process can consume a significant amount of resources. This puts limits on what can be achieved in realtime signal processing, where time-varying convolution can have the most impact.

This paper is organised as follows. It begins by providing an outline of the audio signal processing mathematics behind time-varying convolution, leading to the fundamental aspects of its current (serial) implementation. Following this, the paper discusses each component of the process and how parallel computation can be achieved. This is followed by a reference implementation using a well-defined open specification. A final section is then dedicated to a discussion of test results, comparing serial and parallel computation.

2. Time-varying convolution

Recently, the topic of time-varying filters has received some attention in the digital audio signal processing literature. Arbitrary switching of filters was discussed in Laroche (2007), where a framework for the analysis of difficult cases was provided. More practical uses were also discussed, such as the use of coefficient modulation by periodic signals (Kleimola et al., 2011; Lazzarini et al., 2009, 2011; Timoney et al., 2014), building on the recent theory of periodic linear time-varying (PLTV) filters (Cherniakov, 2003), for both finite and infinite response cases. Following this, a special case of arbitrary time-varying linear finite impulse response filters (TVFIR) was introduced for time-varying convolution applications (Brandtsegg et al., 2018), which is the focus of the discussion in this paper.

CONTACT Victor Lazzarini victor.lazzarini@mu.ie

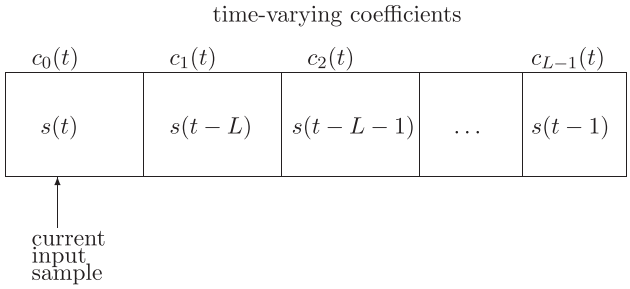

Figure 1. Input signal and time-varying filter coefficients.

In order to introduce the definition of time-varying convolution, we begin by recalling that a discrete linear finite impulse response filter (FIR) of length L applied to an input signal $x(t)$ is defined by the convolution (Oppenheim et al., 1999)

$$y(t) = \sum_{n=0}^{L-1} a_n x(t-n). \tag{1}$$

In the time-invariant case, the set of coefficients a_n determine filter impulse response $h(t)$.

$$h(t) = \sum_{n=0}^{L-1} a_n u(t-n) = a_t \tag{2}$$

From the impulse response, we can derive the filter transfer function.

$$H(z) = \sum_{n=0}^{L-1} a_n z^{-n} \tag{3}$$

The discrete filter spectrum is then given by setting $z = e^{j\omega k}$ and $\omega = 2\pi/L$, from which we can obtain its amplitude ($|H(e^{j\omega k})|$) and phase ($\arg\{H(e^{j\omega k})\}$) responses. The filter can be applied to the signal either as in Equation (1) or as a product of the filter and signal spectra.

$$Y(z) = H(z)X(z) \tag{4}$$

From this, a TVFIR can be defined by removing the assumption that the coefficients a_n of Equation (1) are fixed. In this case, we replace these with a set of time-varying coefficients $c_n(t)$,

$$y(t) = \sum_{n=0}^{L-1} c_n(t) x(t-n) \tag{5}$$

We now take the coefficients $c_n(t)$ from an arbitrary input waveform $s(t)$, by shifting in samples from the right, as

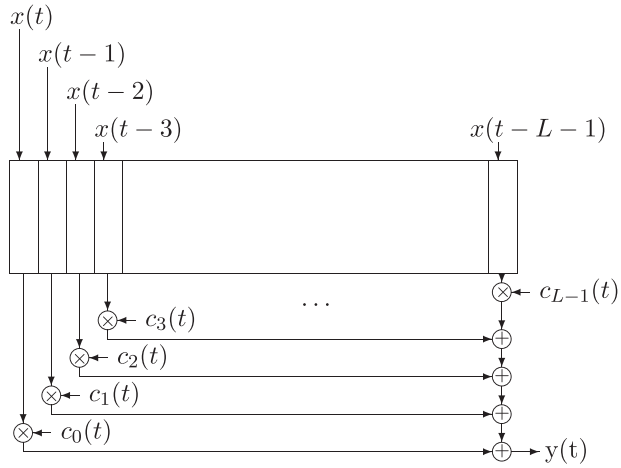

Figure 2. Time-varying convolution.

described by

$$c_n(t) = \begin{cases} s(t), & n = t \bmod \mathrm{L} \\ c_n(t-1), & \text{otherwise} \end{cases} \tag{6}$$

This mechanism defines a delay line of length L, with the samples of the signal $s(t)$ being shifted in, replacing the oldest coefficient as shown in Figure 1. The write position wraps around at the end and proceeds circularly around the coefficient buffer.

The TVFIR convolution is then computed as shown in Figure 2. Note that the input signals $x(t)$ and $s(t)$ are held in two separate delay lines. For each sample of output, we take one sample from each signal and discards one sample from each delay line. This algorithm defines the *time-varying convolution* process, a special case of TVFIRs where we take two arbitrary input signals, using one of them nominally as the filter impulse response, and the other as its input signal. However, it is important to note that there is no actual distinction between these two inputs, as they can be equally serve as 'signal' or 'impulse'.

2.1. Implementation

A first pass at the implementation of time-varying convolution would employ Equations (5) and (6). As denoted by these, at every output sample, we would only need to shift one sample in and discard another sample, of each signal. The replacement of the filter coefficients only needs to write a single value for every new output sample, holding all the others in memory, and shifting the read/write position circularly. An efficient implementation of convolution also only needs to write one single sample into the input signal delay line at a time. As an example, an implementation of the time-domain algorithm is shown as a Csound code fragment in Listing 1.

Listing 1 Time-varying convolution

```
// coefficients delay line memory (ifn)
ifn = ftgen(0,0,iL,7,0,iL,0)
// writing pointer
andx = phasor(sr/iL)
// write signal 2 as coefficients to ifn
tablew(asig2,andx,ifn,1)
// compute direct convolution of sigs 1 and 2
aout = dconv(asig1,isiz,ifn)
```

The complexity is $O(N^2)$, as for each output sample N multiplications and additions are necessary. For large filter sizes, it can become increasingly prohibitive to employ this method, especially if the aim is to be able to process signal in real time.

The algorithm outlined above describes the time-domain implementation of the process, based on a *direct* convolution approach. As an alternative to this, we have an equivalent form that can be computed in the frequency domain. By applying Equation (4), we can define time-varying convolution as

$$Y(i,k) = C(i,k)X(i,k) \tag{7}$$

Note that now we have a function of two variables, i defining time and k frequency. This effects the product of the two input spectra $C(i,k)$ and $X(i,k)$, which can be seen as snapshots of the coefficients and input at a time i. For this, we employ a rectangular window to select data from the two input signals every L samples. Using $N \geq 2L-1$, $\omega = 2\pi/N$, we have

$$C(i,k) = \sum_{n=0}^{N-1} w(n-iL)c_n(i)e^{-j\omega kn} \ (0 \leq k < N-1), \tag{8}$$

and

$$X(i,k) = \sum_{n=0}^{N-1} w(n-iL)x_i(n)e^{-j\omega kn} \ (0 \leq k < N-1), \tag{9}$$

where $w(n)$ is a rectangular window of size L (the filter length), and $i \in \mathbb{Z}, i \geq 0$. As per Figure 1 and Equation (6), the coefficients $c_n(i)$ are equivalent to a block of samples taken from the input signal $s(t)$ starting at time $t = i$. By choosing an appropriate size N for each transform, it is possible to employ a suitable FFT algorithm to implement these transforms, reducing the computational complexity of the implementation. To obtain the time-varying convolution in the time-domain, we first apply the inverse DFT to each output block i of size N, and then overlap-add these,

$$y_i(n) = \sum_{k=0}^{N-1} Y(i,k)e^{j\omega kn} \ (0 \leq n < N-1). \tag{10}$$

Assuming $y_i(n) = 0$ for $n < 0$ and $n \geq N$, then the time-varying convolution output is defined by the overlap-add expression

$$y(t) = \sum_{i=0} y_i(t-iL). \tag{11}$$

The complete process is shown in Figure 3, which describes the process of obtaining one output block of L samples, which is overlap-added to form the output signal.The signal labelled as *input 1* is used to produce the spectra $X(i,k)$ and *input 2* yields $C(i,k)$.

While this second approach is much improved in terms of computational efficiency, it can pose problems in practice. The major difficulty is that it requires all input samples to each DFT to be available before it can proceed, which is not an issue in off-line applications, but it will introduce a constant input–output latency in real time applications. The minimum latency in seconds will be equivalent to the product of the filter size L and the sampling rate f_s. As large delays are generally undesirable, this may place a practical limit to the size of filters employed in real time applications.

The solution to this problem is to consider that both approaches, in the time and frequency domain, are special cases of a more general algorithm, called *partitioned convolution* (Lazzarini, 2017a; Wefers, 2015). The method sections the filter into a number of partitions before applying convolution and assembling the output. In the former case, the partition is a single sample, and in the latter case, the complete filter (that is, there is only one partition). Starting with Equation (9), we choose a suitable value for the partition size M, $1 \leq M \leq L$, so that $P = \lceil L/M \rceil$, and modify Equation (6) as follows (with $\omega = \pi/M$), to produce the successive spectral frames for

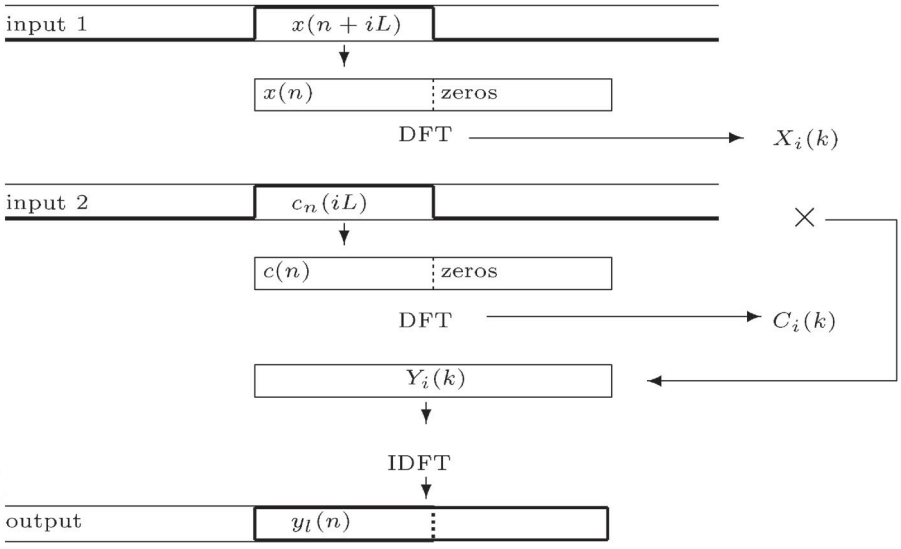

Figure 3. Frequency-domain implementation of convolution, input 1 provides the data for $X(i,k)$ and input 2 for $C(i,k)$.

the coefficients,

$$C(l,k) = \begin{cases} \sum_{n=0}^{2M-1} s_i(n) e^{-j\omega kn}, & l = i \bmod \mathrm{P} \\ C(l-1,k), & \text{otherwise} \end{cases} \tag{12}$$

In this re-definition, $C(t,k)$ now refers to zero-padded blocks of $2M$ samples in the frequency domain, taken from the second input signal at M intervals and shifted into a circular buffer. With this, we can re-define the operation as the sum of spectral products relative to each partition,

$$Y(i,k) = \sum_{m=0}^{P-1} C(m,k) X(i-m,k) \tag{13}$$

using $X(i,k)$ as defined in Equation (9).

Following the IDFT (Equation (10)), the time-varying convolution of the two signals can now be obtained by the overlap-add operation.

$$y(t) = \sum_{i=0} y_i(t - iM) \tag{14}$$

As noted above, the special cases of $P = 1$ and $P = L$ are equivalent to the two original approaches discussed above. In most applications of time-varying convolution, they will be employed only in very particular situations. The minimum latency of partitioned convolution is defined by the partition size M; decreasing it reduces the delay in getting an output out of the system, but with a corresponding increase in number of operations. The choice of partition size will be determined in terms of minimising the input/output latency while still retaining an efficient use of computation resources.

3. Parallel computation

The serial computation of time-varying convolution as described in Section 2.1 is discussed in detail in Brandtsegg et al. (2018). From a macro-level perspective, we have four sequential steps, in which the first stage can be trivially split into two concurrent operations:

(1) DFT of two input signals.
(2) Sum of the spectral products.
(3) Inverse DFT.
(4) Overlap-add (OLA).

Each one of these steps may be broken down into a combination of serial and parallel sets of operations. In the following sections, we will discuss these, providing the background to the implementation presented in Section 4.

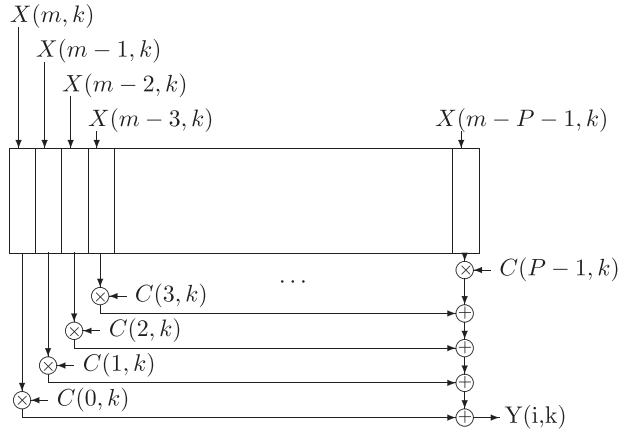

Figure 4. Paritioned time-varying convolution.

3.1. DFT

Practical efficient computation of the DFT will always employ one of the existing FFT algorithms. In the case of time-varying convolution, the best candidate for this is the original radix-2 FFT (Cooley & Tukey, 1965), since it is possible to choose transform sizes that will match its requirements. Starting with the DFT as defined by

$$X(k) = \sum_{t=0}^{N-1} x(t) e^{-2\pi jkt/N}. \tag{15}$$

We divide this into two half-size transforms, one for even t and another for odd t, $\omega^{-k} = e^{-2\pi jk/N}$

$$X_N(k) = E(k) + \omega^{-k} O(k) \tag{16}$$

Since the DFT is periodic, $X(k+N) = X(k)$, we have the following relationships.

$$X_N(k) = \begin{cases} E(k) + \omega^{-k} O(k), & 0 \le k < N/2 \\ E(k - N/2) + \omega^{-k} O(k - N/2), & N/2 \le k < N \end{cases} \tag{17}$$

Applying the identity $e^{-2\pi j(k+N/2)/N} = -e^{-2\pi jk/N}$, we can re-define the DFT as a pair of equations, for $k = 0, \ldots, N/2$:

$$\begin{aligned} X_N(k) &= E(k) + \omega^{-k} O(k) \\ X_N(k + N/2) &= E(k) - \omega^{-k} O(k) \end{aligned} \tag{18}$$

Finally, Equation (18) can be applied recursively, halving the transform size each time, down to $N = 2$. The computation then starts with single points, and is repeated $\log_2 N$ times sequentially until the transform is completed, as demonstrated in Figure 5 for $N = 8$.

Since each one of the $N/2$ applications of Equation (18) is independent, we can compute each one of the $\log_2 N$ serial steps in $N/2$ concurrent sets of operations. However, as seen in Figure 5, the indexes of the input vector are not in ascending order. Due the nature of the decimation-in-time radix-2 FFT algorithm, this array needs to be re-arranged so that the spectral samples are in the correct order at the output. This requires the sample indexes to be bit reversed. With a bit-reverse operator $b(n)$, this operation is defined as follows.

$$y(n) = x(b(n)) \tag{19}$$

The reordering of the input may also be computed in N parallel operations as an extra sequential step, provided

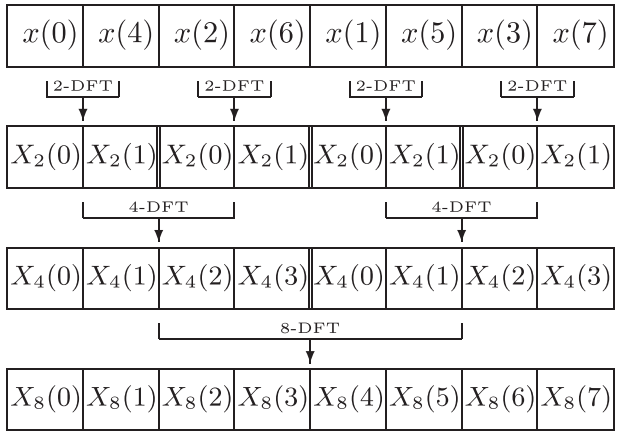

Figure 5. Sequential steps for Radix-2 FFT of size 8, each of which can be computed in parallel with 4 sets of operations.

that the mapping $b(n)$ is prepared in advance. The inverse DFT,

$$x(t) = \sum_{k=0}^{N-1} X(k) e^{2\pi jkt/N}, \tag{20}$$

is implemented using a similar approach, reordering the data and applying the required number of FFT passes, but with inverted complex exponential scaling factors.

3.1.1. Real-to-complex transforms

Since audio signals are real-valued, it is possible to take advantage of the fact that it has a spectrum with Hermitian symmetry and therefore we will be able to use a half-size transform to compute it (Chu & George, 1999; Mulgrew et al., 1999). We can re-interpret the input as a complex-valued sequence of length $M = N/2$ and apply the DFT. In order to obtain the corresponding non-negative spectrum $Y(k)$, we apply the following expressions.

$$\begin{aligned} R(k) &= \frac{1}{2}\left(X(k) + \overline{X(M-k)}\right) \\ I(k) &= \frac{j}{2}\left(\overline{X(M-k)} - X(k)\right) \end{aligned} \tag{21}$$

$$\begin{aligned} Y(k) &= R(k) + \omega^{-k} I(k), \; 0 \le k < M/2 \\ Y(M-k) &= \overline{R(k) - \omega^{-k} I(k)}, \; M/2 < k < M \end{aligned} \tag{22}$$

The complex-to-real inverse DFT will employ a similar approach, first preparing an input to a DFT whose output can be re-interpreted as a real-valued sequence. For this, we apply the following equations to obtain $R(k)$ and $I(k)$, and from those, $Y(k)$

$$\begin{aligned} R(k) &= \frac{1}{2}\left(X(k) + \overline{X(M-k)}\right) \\ I(k) &= \frac{j}{2}\left(X(k) - \overline{X(M-k)}\right) \end{aligned} \tag{23}$$

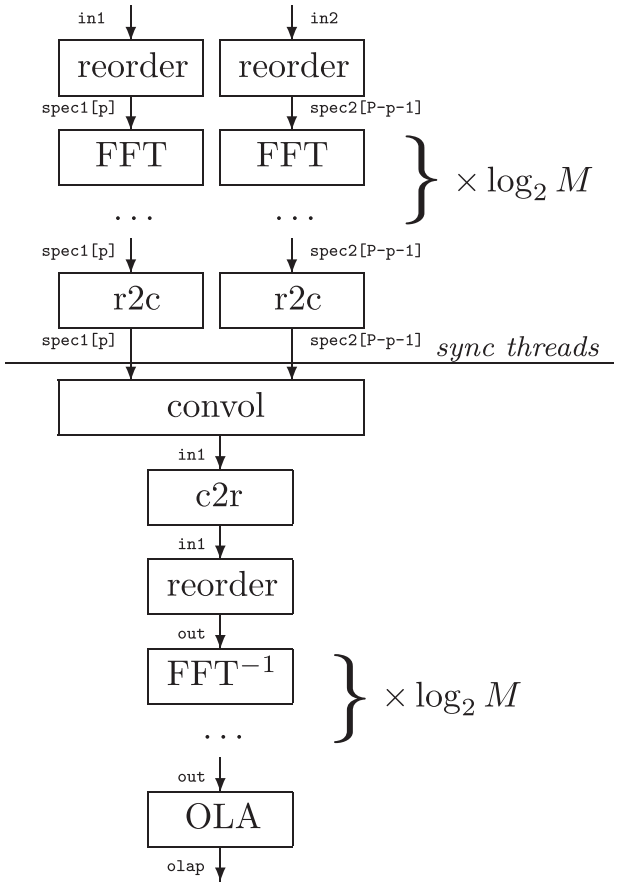

Figure 6. OpenCL processing kernels, memory buffers (teletype), and threads synchronisation, with M denoting the size of each partition, p the current partition, and P the total number of partitions.

$$\begin{aligned} Y(k) &= R(k) + \omega^k I(k),\ 0 \le k < M/2 \\ Y(M-k) &= \overline{R(k) - \omega^k I(k)},\ M/2 < k < M \end{aligned} \tag{24}$$

It is possible to compute these equations in $M/2$ concurrent sets of operations, requiring one extra pass to the ones listed in Section 3.1. The total number of sequential steps for each forward and inverse DFT is $\log_2 M + 2$.

3.2. Spectral products

At the centre of time-varying convolution, we have $(M-1)P$ complex products, plus 2P real multiplications, where M is the partition (or half-transform) size. This provides the justification for the extra step in calculating a real-to-complex transform, as we have saved half the operations we would otherwise have had to apply. As defined by the equation

$$Y(i,k) = \sum_{m=0}^{P-1} C(m,k)X(i-m,k), \quad 1 \le k < M \tag{25}$$

these are all independent of each other and can be computed in parallel in one single pass. For $k = 0$, we have the special case of treating the real and imaginary parts as two real numbers, holding the 0 Hz and Nyquist frequency points.

The final sum into M points can also be computed in parallel. As we will see in Section 4, the implementation of these operations can be combined together by making sure that the sums are atomic to avoid race conditions.

3.3. OLA

The final step is the application of the overlap-add operation, which also needs to include the scaling of each sample by the half-transform size M if this has not been applied at any other stage. In order to produce one output block of M samples, the following expression is employed.

$$y(t) = \frac{y_i(t) + y_{i-1}(t+M)}{M}, \quad 0 \le t < M, \tag{26}$$

where $y_i(t)$ is the current output of the inverse DFT operation. This equation can be computed in M concurrent operations.

4. An OpenCL implementation

In order to demonstrate the principles outlined in Section 3, a reference implementation of time-varying convolution is presented in this section. For this, the Open Computing Language (OpenCL) framework (Khronos OpenCL Working Group, 2019) was chosen due to its widespread availability, generality and good support for shared-memory programming, which is required by the type of parallelism employed here. The framework supports heterogeneous platforms, allowing for the results of this article to be transferable to a variety of computing devices.

OpenCL is composed of a C API and an intermediate language. The latter allows cross-platform programming for parallel programs within a well-defined computing environment. The former provides a means for host control of concurrent computation, including just-in-time compilation, memory access, and execution. OpenCL programs are defined as *kernels*, which run in parallel under the chosen platform, which can be a general-purpose central processing unit (CPU), a graphics processing unit (GPU), or another computing device such as a hardware accelerator. The OpenCL language resembles C very closely, therefore it is well understood in the signal processing community, allowing for results to be efficiently communicated.

The present implementation consists of a C++ class, which may be employed in any signal processing application, and an accompanying sample application in the

form of a unit generator for the Csound sound and music computing system (Lazzarini et al., 2016) using CPOF (Lazzarini, 2017b). The details of the C++ code and its use in Csound are beyond the scope of this paper, but the complete source code is available as an Online Supplement to this article. We will note, nevertheless, the most salient points as needed, while concentrating on the OpenCL implementation of the principles laid out in Section 3.

The typical configuration of an OpenCL application consists of a program run on a host computer, which manages the operations run on one or more parallel computing devices. As part of this process, the host will be responsible for passing the data to the OpenCL kernels, executing them, and fetching the output. In the particular case here, audio data will be buffered in and out of the device as required. The flowchart in Figure 6 shows the structure of the implementation in terms of its processing kernels, data buffers, and threads synchronisation. An aspect that needs significant attention is memory management, in particular, it is important to minimise transfers between the host and the device, keeping intermediate results in it as much as possible. As can be observed in the flowchart, some kernels operate on data in-place, whereas others require out-of-place processing. Data is transferred only at the start and end of the processes, and then placed in the spectral delay line buffers (`spec1[]` and `spec2[]` in Figure 6). It is important also to consider the synchronisation points for the parallel operations, so that we maximise the concurrency. These are determined by the segmentation of the operations in separate kernels. In the following sections, we detail the key aspects of each component in the implementation.

4.1. FFT

The FFT operation involves two separate kernels, `reorder` and `fft`. The former is a very simple operation, which effectively uses a bit-reversal map to re-organise the data in the correct format for the (decimation-in-time) FFT operation. It is a straight implementation of Equation (19) (Listing 2), which is run in M parallel instances, M denoting the partition size. Each kernel instance is indexed by the value of `get_global_id(0)`. Since these kernels may be used with a spectral delay line, an offset argument is provided to select the correct memory block for processing.

Listing 2 Data reorder kernel

```
kernel void reorder(global cmplx *out, global cmplx *in,
                    global const int *b, int offs) {
   int k = get_global_id(0);
   out += offs;
   out[k] = in[b[k]];
   in[b[k]] = 0.f;
}
```

The `reorder` kernel depends on an externally defined mapping array, which provides a bit-reversed number for a given DFT size, generated by the code in Listing 3 (Elster, 1998). This is done once at setup time and copied to read-only memory in the device.

Listing 3 Bit-reversal map generation

```
for (int i=0; i < M; i++)
   bp[i] = i;
 for (int i = 1, m = M >> i; i < M;
      i = i << 1,m = m >> 1)
   for (int j = 0; j < i; j++)
     bp[i + j] = bp[j] + m;
```

It is important to comment on the reasons for segmenting the reorder operation as a separate kernel, as it might appear that it could be incorporated as part of the FFT kernel. Firstly, the reorder operation is performed out of place, while the FFT is in place. In order to combine the two in a single kernel, we would need to perform a copy of the data somewhere along the way, which is not ideal. Secondly, the reordering is applied only once, whereas the FFT kernel will be invoked repeatedly $\log_2 M$ times. That would require us to implement some form of branching in the kernel. Finally, it is possible to implement the first FFT pass in the reorder kernel, but that would complicate the code design and exposition, and it is unlikely that it would result in significant performance gains. However, this remains an alternative for a trivial re-factoring of the code.

The FFT kernel itself implements Equation (18) more or less directly (Listing 4).

Listing 4 FFT kernel

```
kernel void fft(global cmplx *s, global const cmplx *w,
                int N, int n2, int offs) {
  int k, i, m, n;
  cmplx e, o;
  s += offs;
  k = get_global_id(0)*n2;
  m = k/N;
  n = n2 >> 1;
  k = k%N + m;
  i = k + n;
  e = s[k];
  o = prod(s[i],w[m*N/n2]);
  s[k] = e + o;
  s[i] = e - o;
}
```

When the FFT is executed, $M/2$ parallel instances of this kernel are invoked $\log_2 M$ times, with the same arguments except for the value of `n2`. This starts at 1 and is doubled each time. The array `s` containing the partial results is updated each time. The kernel depends on read-only array `w` of ω^k scaling factors, which is created and copied at setup time. This is the only difference in the calculation of the forward and inverse transforms. The code makes use of the locally defined inline function `prod()`, which calculates the product of two complex numbers.

4.2. Real-to-complex FFT

Following the computation of the half-size DFT, a conversion kernel needs to run on the data in order to produce the non-negative spectrum (Listing 5). This implements Equations (21) and (22). A special case is applied to the first pair of numbers of the DFT. These two positions will hold the 0 and M points, which are real valued, packed as pair. Note that since `cmplx` is an alias of the native OpenCL `float2` vectorial type, access to each element is made via the subscripts `x` and `y`. The conversion kernel processes the data in place and, as before, an offset is used to select the correct memory block to process.

Listing 5 Real-to-complex FFT conversion kernel

```
kernel void r2c(global cmplx *c, global const cmplx *w,
                int M, int offs) {
 int i = get_global_id(0);
 if(!i) {
```

```
  c[0] = (cmplx)
    ((c[0].x + c[0].y)*.5f, (c[0].x - c[0].y)*.5f);
  return;
 }
 int j = N - i;
 cmplx e, o, cj = conjg(c[j]), p;
 c += offs;
 e =.5f*(c[i] + cj);
 o = .5f*rot(cj - c[i]);
 p = prod(w[i], o);
 c[i] = e + p;
 c[j] = conjg(e - p);
}
```

As in the FFT case, we need to run $M/2$ parallel instances as each kernel operates on a pair of numbers. As before, a read-only array of complex scaling factors is required, and this can be also created and copied to the device at setup time. The code makes use of two other locally-defined inline functions `rot()` and `conjg()`, providing rotation of a number by π and the complex conjugate, respectively. The corresponding inverse conversion operation is shown in Listing 6, which is applied to the data prior to the reorder and FFT kernels.

Listing 6 Complex-to-real FFT conversion kernel

```
kernel void c2r(global cmplx *c, global const cmplx *w,
                int M) {
  int i = get_global_id(0);
  if(!i) {
   c[0] =
     (cmplx) ((c[0].x + c[0].y), (c[0].x - c[0].y));
   return;
 }
 int j = M - i;
 cmplx e, o, cj = conjg(c[j]), p;
 e = .5f*(c[i] + cj);
 o = .5f*rot(c[i] - cj);
 p = prod(w[i], o);
 c[i] = e + p;
 c[j] = conjg(e - p);
}
```

4.3. Spectral products

The two input spectra are combined using the partitioned convolution operation defined by Equation (13). The process involves employing two spectral delay lines, to which the operation is applied, as shown schematically in Figure 7. The delays are implemented using circular buffers, and the output of each DFT is computed into the corresponding position prior to the execution of the convolution kernel. Note that due to the fact that conventionally we are using future spectra of one of the input signals, the actual implementation places these in reverse index order into the delay line in order

to correctly express the process outlined in Section 2.1 (for more details, see Brandtsegg et al. (2018)). This is denoted by the reverse indexing of the `spec2[]` buffer in Figure 6. The convolution kernel in Listing 7 expects the two spectra to be placed in reverse order relative to each other.

Listing 7 Partitioned convolution kernel

```
kernel void convol(global float *out,
        global const cmplx *in, global const cmplx *coef,
        int rp, int M, int P) {
  int k = get_global_id(0);
  int n = k%b;
  int n2 = n << 1;
  rp += k/M;
  in += (rp < P ? rp : rp%P)*M;
  cmplx s = n ? prod(in[n], coef[k]) :
       (cmplx) (in[0].x*coef[k]. x, in[0].y*coef[k].y);
  atomicAdd(&out[n2], s.x);
  atomicAdd(&out[n2+1], s.y);
}
```

For each output block, $M \times P$ kernels are instantiated in parallel. When they are run, an updated read position `rp` is fed as a parameter so that the correct spectra are read circularly from the two delay lines. The product of the two spectra is calculated, and then, using the locally-defined inline atomic sum function `atomicAdd()`, the results are accumulated in the output buffer. Note that this also expects that its memory locations are reset to 0 prior to the kernels being executed.

4.4. OLA

The OLA operation completes the time-varying convolution process (Listing 8). This is implemented in another very simple kernel, implementing Equation (26), run in M parallel instances. In addition to this, we also use the kernel to copy the second half of the output buffer for use at the next time round. Following this step, we can transfer the output from the device to the host program.

Listing 8 Overlap-add kernel

```
kernel void olap(global float *buf, global const float *in,
                 int M){
  int n = get_global_id(0);
  buf[n] = (in[n] + buf[M+n])/M;
  buf[parts+n] = in[M+n];
}
```

5. Results and discussion

The present implementation, in the form of the Csound opcode `cltvconv`, was tested under two different computing devices: an Intel HD630 GPU and an Intel Core I7 CPU. These tests were set against the serial implementation presented in Brandtsegg et al. (2018), the opcode `tvconv`. Identical code (Listing 9), except for the time-varying convolution implementation, employing two audio inputs, was tested using various settings of the partition size M and filter size L, $M < L$. The sampling rate was set to 44.1 kHz and the processing block size (`ksmps`) was 10 samples. The testing was completely automated using a Python script to run the Csound code, time the operation and produce the plots and table shown here. Each run was 100 seconds long and 10 runs were

used for each setting, from which an average was taken. Figure 8 shows the results of these tests in terms of the real time ratio *d*:*c*, where *d* is the sound duration and *c* is computation time taken to synthesise it. This is a measure of the fitness of a process to run in real time. Ratios < 1 indicate that real time performance is not achievable.

Listing 9 Test code

```
instr 1
 iM = pow(2,p4)
 iL = pow(2,p5)
 idev = p6
 ain1 = diskin:a("beats.wav", 1, 0, 1)
 ain2 = diskin:a("fox.wav", 1, 0, 1)
 if idev < 2 then
  asig = cltvconv(ain1,ain2,1,1,iM, iL,idev)
 else
  asig = tvconv(ain1,ain2,1,1,iM,iL)
 endif
  out(asig)
endin
```

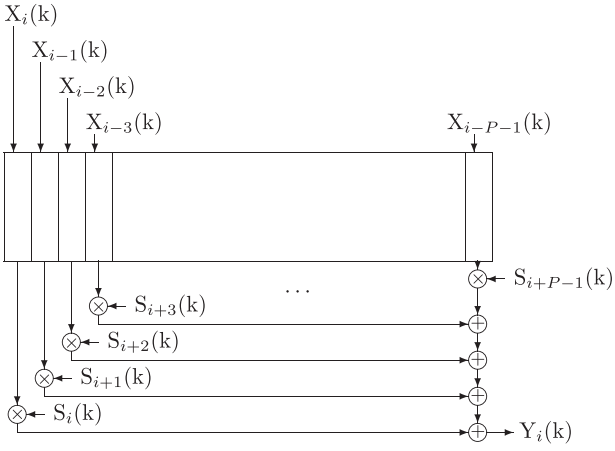

Figure 7. Spectral delay lines in the partitioned convolution computation.

From these plots, we can observe that both partition size *M* and filter length *L* have a significant impact on performance. With smaller partition sizes, serial code in the host is less costly than parallel computation for shorter filter lengths. As reported elsewhere (Crespi, 2016; Lazzarini et al., 2014), shorter DFT sizes do not make the most of the parallel computing resources. This is also the case for the partitioned convolution operation. There are more kernel calls and data transfers per output samples, which generates increased overheads. However, as *L* is increased, its performance is severely degraded, regardless of *M*. This has less of an effect in the parallel implementations, where the increase in cost is much less pronounced. We can also observe that the support for high levels of concurrency in the GPU provides a significant performance boost in comparison to the CPU device. The specific results from GPU tests are shown in Table 1, where we can see that, at minimum, the real time ratio is 2.7, increasing to 168.1 with $M = 32768$ and $L = 2^{16}$.

These results indicate that in applications where input/output latency is not a consideration, the computation of time-varying convolution should be performed in parallel, with an appropriate choice of *M*. It is important to note that the opcode does not employ any extra buffering of data, therefore the code latency is strictly defined by the choice of partition size. Of course in situations where very short partition sizes are required for minimal latency, the use of the parallel implementation is not recommended. However, if small delays can be tolerated, it is also best to use this approach when very long filters are employed. In fact, this implementation allows real time operation in cases where this is not achievable with serial code. For example, for filters that are longer than 2^{22} samples (about 95 s at 44.1 kHz), its realtime ratio falls below 1 (except in the case of large *M*, above 2^{13} samples). In contrast, parallel computation of the time-varying convolution running in the GPU allows filter sizes that are several minutes long. This opens up new possibilities for real time audio processing in interactive performances.

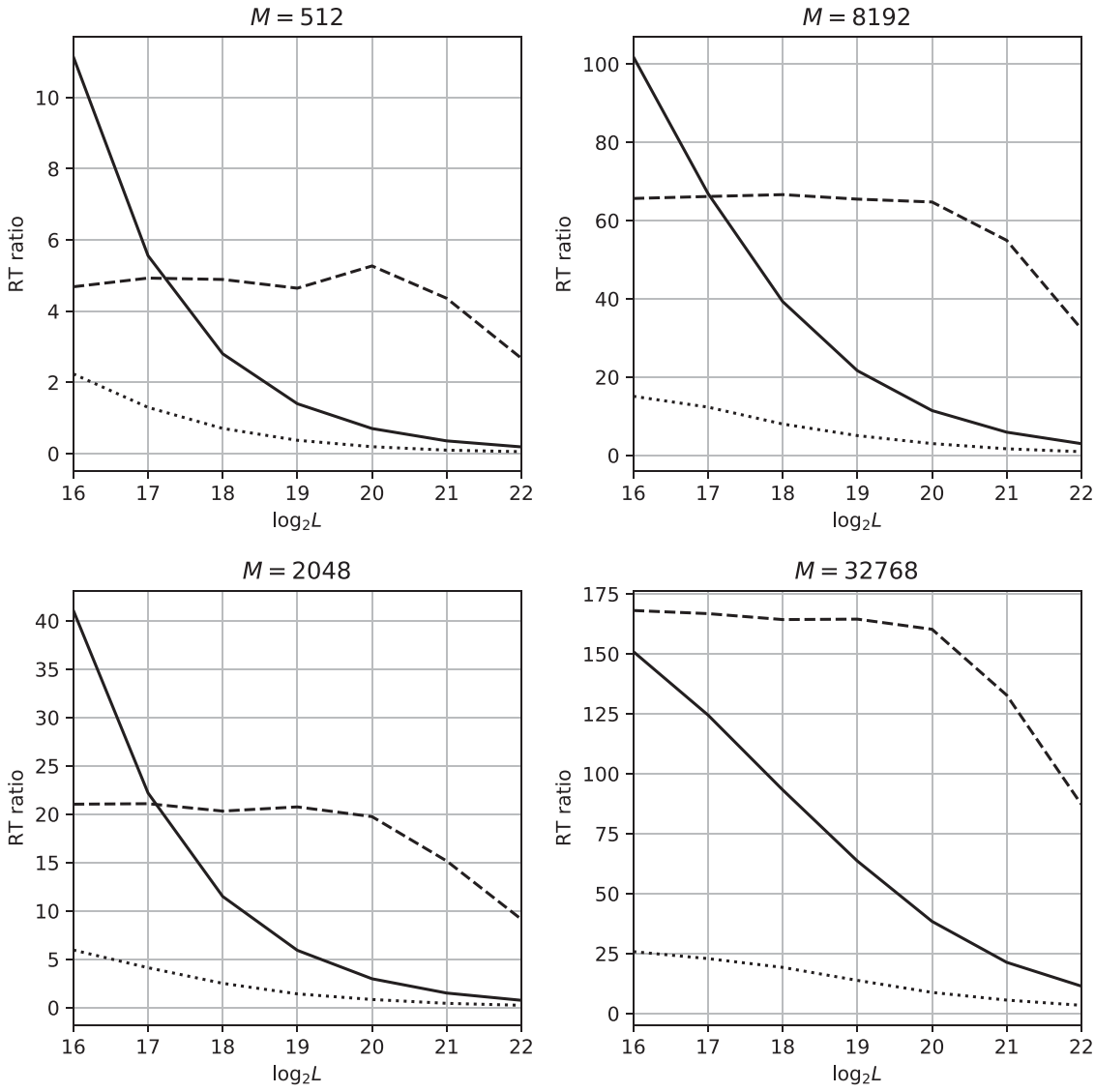

Figure 8. Performance of time-varying convolution run as serial code in the host computer (solid lines), in parallel on a GPU (dashes) and CPU (dots), in different partition sizes (M) and filter lengths (L), plotted in terms of realtime (RT) ratios.

Table 1. Realtime ratios for the GPU computation of time-varying convolution, for various settings of partition size M and filter length L.

	$\log_2 L$						
M	**16**	**17**	**18**	**19**	**20**	**21**	**22**
512	4.7	4.9	4.9	4.6	5.3	4.4	2.7
2048	21.0	21.1	20.3	20.8	19.8	15.2	9.1
8192	65.7	66.2	66.6	65.5	64.8	55.0	32.3
32768	168.1	166.8	164.3	164.5	160.3	132.8	87.2

6. Conclusions

This paper presented a method for computing the time-varying convolution in parallel. This was motivated by related work demonstrating that spectral-domain operations could be efficiently implemented in concurrent steps. The results presented here have consistently agreed with this, proving that the approach can be very useful in musical signal processing applications.

In particular, the article set out to provide an alternative to the existing serial implementation that would remove its limitations in terms of filter sizes. We have shown that the use of GPU for parallel computation, employing the methods described here, is capable of improving significantly this aspect. At the same time, it was also noted that shorter DFT sizes are not as efficient, mostly due to the overheads in data transfer from host to device and device to host, and an increased number of kernel calls. However, these issues are not intrinsically related to the parallel computation method, and may be improved in future computing devices.

Supplemental online material for this article can be accessed at http://github.com/vlazzarini/opencl_fft. The source code for the OpenCL implementation (discussed in Section 4) is available, alongside the test scripts used to obtain the results in Section 5. A short video demonstrating the use of the code in performance is also available at https://youtu.be/r3XeGe63l9Y.

Disclosure statement

No potential conflict of interest was reported by the author(s).

References

Belloch, J. A., Badia, J. M., Igual, F. D., Gonzalez, A., & Quintana-Orti, E. S. (2018). Optimized fundamental signal processing operations for energy minimization on heterogeneous mobile devices. *IEEE Transactions on Circuits and Systems I: Regular*, *65*(5), 1614–1627. https://doi.org/10.1109/TCSI.2017.2761909

Belloch, J. A., Gonzalez, A., Martinez-Zaldivar, F. J., & Vidal, A. M. (2013). Multichannel massive audio processing for a generalized crosstalk cancellation and equalization application using GPUs. *Integrated Computer-Aided Engineering*, *20*(2), 169–283. https://doi.org/10.3233/ICA-130422

Belloch, J. A., Gonzalez, A., Martnez-Zaldvar, F. J., & Vidal, A. M. (2011). Real-time massive convolution for audio applications on GPU. *Journal of Supercomputing*, *58*(3), 449–457. https://doi.org/10.1007/s11227-011-0610-8

Bradford, R., Ffitch, J., & Dobson, R. (2011). *Real-time sliding phase vocoder using a commodity GPU*. Proceedings of ICMC2011, ICMC (pp. 587–590). University of Huddersfield and ICMA, Huddersfield, UK. ISBN 978-0-9845274-0-3.

Brandtsegg, Ø., Saue, S., & Lazzarini, V. (2018). Live convolution with time-varying filters. *Applied Sciences*, *8*(1), 1–29. https://doi.org/10.3390/app8010103

Cherniakov, M. (2003). *An introduction to parametric digital filters and oscillators*. John Wiley & Sons.

Chu, E., & George, A. (1999). *Inside the FFT black box: serial and parallel fast fourier transform algorithms*. Computational Mathematics. Taylor & Francis.

Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, *19*(90), 297–301. https://doi.org/10.1090/S0025-5718-1965-0178586-1

Crespi, A. G. (2016). *Spectral manipulation of audio using general-purpose graphics processing units* [Master's thesis]. Politecnico di Milano.

Elster, A. C. (1998). Fast bit-reversal algorithms. In *Proceedings of the ICASSP 89*, Glasgow, UK (pp. 1099–1102).

Hamilton, B., & Webb, C. (2013). Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid. In *Proceedings of the 16th digital audio effects (DAFx-2013)*, Maynooth, Ireland (pp. 1–8).

Khronos OpenCL Working Group (2019). *The OpenCL specification*. Beaverton.

Kleimola, J., Lazzarini, V., Välimäki, V., & Timoney, J. (2011). Feedback amplitude modulation synthesis. *EURASIP Journal on Advances in Signal Processing*, *2011*(434378), 1–18. https://doi.org/10.1155/2011/434378

Laroche, J. (2007). On the stability of time-varying recursive filters. *Journal of the Audio Engineering Society*, *55*(6), 460–471. http://www.aes.org/e-lib/browse.cfm?elib = 14168

Lazzarini, V. (2017a). *Computer music instruments*. Springer.

Lazzarini, V. (2017b). Supporting an object-oriented approach to unit generator development: the csound plugin opcode framework. *Applied Sciences*, *7*(10), 1–32. https://doi.org/10.3390/app7100970

Lazzarini, V., Ffitch, J., Timoney, J., & Bradford, R. (2014). Streaming spectral processing with consumer-level graphics processing units. In *Proceedings of the international conference on digital audio effects (DAFx-14)*, Erlangen, Germany (pp. 1–8).

Lazzarini, V., Ffitch, J., Yi, S., Heintz, J., Brandtsegg, Ø., & McCurdy, I. (2016). *Csound: A sound and music computing system*. Springer.

Lazzarini, V., Kleimola, J., Timoney, J., & Välimäki, V. (2009). Five variations on a feedback theme. In *Proceedings of the 12th international conference on digital audio effects* (pp. 139–145).

Lazzarini, V., Kleimola, J., Timoney, J., & Välimäki, V. (2011). Aspects of second-order feedback AM synthesis. In *Proceedings of the international computer music conference* (pp. 92–98).

Mulgrew, B., Grant, P., & Thompson, J. (1999). *Digital signal processing: concepts and applications*. Macmillan Press.

Oppenheim, A. V., Schafer, R. W., & Buck, J. R. (1999). *Discrete-time signal processing* (2nd ed.). Prentice-Hall, Inc.

Roeber, N., Kaminski, U., & Masuch, M. (2007). Ray acoustics using computer graphics technology. In *Proceedings of the 10th digital audio effects (DAFx-2007)*, Bordeaux, France (pp. 1–6).

Savioja, L. (2010). Use of GPUs in room acoustic modeling and auralization. In *Proceedings of the international symposium on room acoustics (ISRA 2010)*, Melbourne, Australia (pp. 1–6).

Savioja, L., Valimaki, V., & Smith, J. (2010). Real-time additive synthesis with one million sinusoids using a GPU. In *Proceedings of the 128th AES*, London, UK (pp. 1–6).

Savioja, L., Valimaki, V., & Smith, J. O. (2011). Audio signal processing using graphics processing units. *Journal of the Audio Engineering Society*, *59*(1), 3–19. http://www.aes.org/e-lib/browse.cfm?elib = 15772

Timoney, J., Pekonen, J., Lazzarini, V., & Välimäki, V. (2014). Dynamic signal phase distortion using coefficient-modulated all pass filters. *Journal of the Audio Engineering Society*, *62*(9), 596–610. https://doi.org/10.17743/jaes.2014.0033

Tsai, P.-Y., Wang, T.-W., & Alvin, S. (2010). GPU-based spectral model synthesis for real-time sound rendering. In *Proceedings of the 13th digital audio effects (DAFx-2010)*, Graz, Austria (pp. 1–6).

Wefers, F. (2015). *Partitioned convolution algorithms for real-time auralization* (Vol. 20). Logos Verlag.