



# Real-Time CUDA Implementation of High-Order Vertical Synchrosqueezing

1<sup>st</sup> Jacek Gambrych   
Warsaw University of Technology  
Institute of Electronic Systems  
00-665 Warsaw, Poland

2<sup>nd</sup> Karol Abratkiewicz   
Warsaw University of Technology  
Institute of Electronic Systems  
00-665 Warsaw, Poland

**Abstract**—Fast and efficient implementations of Fourier’s analysis techniques for non-stationary signals have been desired recently, not only in civilian but also in military applications. The ability to work in a real-time regime with high dependability and reliability when wideband signals are processed is a challenging but highly desirable task (e.g. analysis of radar signals). This article delves into the real-time implementations of two variants of vertical synchrosqueezing (VSS). The ability to achieve real-time processing was enabled by leveraging Nvidia CUDA technology. Validation through real-life radar signal processing scenarios has confirmed the accuracy and efficiency of the proposed algorithms. To the authors’ knowledge, this is the first real-time implementation of VSS given in the literature.

**Index Terms**—signal processing, synchrosqueezing, GPGPU, CUDA

## I. INTRODUCTION

In recent years, there has been significant development of Fourier’s analysis of non-stationary signals, often called time-frequency (TF) methods in the literature. Alongside this development, one can mention methods for improving the short-time Fourier transform (STFT) readability, as pioneered by Koderá et al. [1]. Of these, the VSS technique is one of the most important and promising.

Vertical synchrosqueezing is a signal processing technique used in TF analysis to improve the readability of the TF representation of a signal. It involves squeezing the vertical axis of the TF representation towards the instantaneous frequency of the signal, thereby enhancing the visibility of its frequency components and allowing for more accurate identification of transient features. This approach can be advantageous in analyzing non-stationary signals, which has made vertical synchrosqueezing a technique successfully applied in many areas, including radar systems [2], [3], voice signal processing [4], seismic analysis [5], and engine vibration monitoring [6].

In the literature, the high-order synchrosqueezing transform is relatively widely described [7], [8]. However, almost none of the existing works consider the computational burden and possibility of implementing the signal concentration method on real-time platforms. To the authors’ knowledge, the first attempt to address the real-time implementation of the VSS was presented by the authors in [2], where the techniques were used for radar signal analysis. This lack of consideration results from synchrosqueezing’s fundamental disadvantage, such as the high computational complexity of more sophisticated

and better-performing algorithms. Thus, in most cases, it is impossible to perform real-time computations on a standard microprocessor.

The answer to this problem may be the general-purpose computing on graphical processing units (GPGPU) and, particularly, CUDA technology. CUDA, introduced by Nvidia in 2007, is one of the most widespread technologies that enable GPGPU. Apart from its enormous computing power, one of the main advantages of CUDA is that it is a comprehensive technology. CUDA includes hardware, an application programming interface based on a subset of the C/C++ programming language, and numerous highly optimized libraries dedicated, among others, to digital signal processing, such as cuFFT. All these features make CUDA an easy-to-use and highly cost-effective solution for performing parallel computations [9]. As a result, it is very often used for computationally demanding signal processing applications that must operate in real-time [10], [11].

Thus, two selected synchrosqueezing algorithms presented in [2] that give excellent results were implemented using CUDA technology to make real-time computations possible even for highly demanding tasks. The presented paper describes the implementation details as well as the obtained performance of the efficient and real-time algorithms for non-stationary radar signal processing.

## II. HIGHER-ORDER VERTICAL SYNCHROSQUEEZING

The VSS improves the STFT separability by relocating the transform values according to the instantaneous frequency estimator. The STFT is defined as follows [12]:

$$F_x^h(t, \omega) = \int_{\mathbb{R}} x(\tau + t) h^*(\tau) e^{-j\omega\tau} d\tau, \quad (1)$$

where  $x(t)$  is the signal under analysis,  $h(t)$  is an analysis window,  $j^2 = -1$ , and  $z^*$  is the complex conjugate of  $z$ . The energy distribution referred to as spectrogram is given as

$$S_x^h(t, \omega) = |F_x^h(t, \omega)|^2. \quad (2)$$

In general, (1) can be computed in two ways. The first results from the definition (1) and relies on the convolution computation of the window and signal modulated according to the angular frequency  $\omega$ . The method is less efficient but permits full control of processing parameters. The second method uses the Fourier transform, which is more efficient, especially

when using the fast Fourier transform (FFT). However, the algorithms have some limitations, e.g. the relationship between the number of FFT points and the analysis window width.

The component separability is obtained according to  $(t, \omega) \mapsto (t, \hat{\omega}(t, \omega))$  using the instantaneous frequency estimator  $\hat{\omega}(t, \omega)$ . The method preserves the distribution phase unaffected and allows inverse transform. The VSS reads as [7]

$$S_x^h(t, \omega) = \int_{\mathbb{R}} F_x^h(t, \Omega) e^{j\Omega(t-t_0)} \delta(\omega - \hat{\omega}_x(t, \Omega)) d\Omega, \quad (3)$$

where  $\delta$  is a Dirac delta distribution.

The instantaneous frequency estimator  $\hat{\omega}(t, \omega)$  has an essential influence on the synchrosqueezed STFT quality. The more nonlinear the instantaneous signal frequency, the more complex an estimator is needed to efficiently concentrate the TF distribution. Among several estimators proposed in the literature [7], the authors selected those with the lowest computational complexity and the possibility of estimating parameters of non-linear and frequency oscillating terms [2]. The first one gives the enhanced VSS (EVSS1) and its frequency reassignment operator

$$\hat{\omega}_x^{[1]}(t, \omega) = \Im \left( j\omega - \frac{\hat{\omega}_x^N(t, \omega)}{\hat{\omega}_x^D(t, \omega)} \right), \quad (4)$$

where  $\hat{\omega}_x^N(t, \omega)$  and  $\hat{\omega}_x^D(t, \omega)$  are defined by (5) and (6) and  $\mathcal{D}^n h = \frac{d^n h(t)}{dt^n}$  and  $\mathcal{T}^n h = t^n \cdot h(t)$ . Operator (4) is an extension (of the third-order) of the simplest operator [12]

$$\hat{\omega}_x(t, \omega) = \Im \left( j\omega - \frac{F_x^{\mathcal{D}^3 h}(t, \omega)}{F_x^h(t, \omega)} \right). \quad (7)$$

The investigated approach, called EVSS1 assumes a signal phase to be locally non-linear. From this feature, one can concentrate strongly oscillating and non-linear components, allowing its further reconstruction.

The second analyzed VSS variant comes from the assumption on the third-order polynomial phase model composed of the individual polynomial signal parameter estimators in the TF domain. From its model, the variant is called canonical third-order VSS (CVSS3), and its frequency reassignment operator reads  $\hat{\omega}_x^{[3]}(t, \omega) =$

$$\Im \left( \hat{\omega}_x(t, \omega) + \hat{q}_x(t, \omega) (t - \hat{t}_x(t, \omega)) + \hat{p}_x(t, \omega) (t - \hat{t}_x(t, \omega))^2 \right), \quad (8)$$

where  $\hat{p}_x(t, \omega)$  is the complex angular jerk estimator given by

$$\hat{p}_x(t, \omega) = \frac{\hat{p}_x^N(t, \omega)}{\hat{p}_x^D(t, \omega)}, \quad (9)$$

where  $\hat{p}_x^N(t, \omega)$  and  $\hat{p}_x^D(t, \omega)$  are defined by (10) and (11) and  $\hat{q}_x(t, \omega)$  is the chirp rate estimator given by

$$\hat{q}_x(t, \omega) = \frac{F_x^{\mathcal{T}^2 h}(t, \omega) F_x^h(t, \omega) + F_x^h(t, \omega)^2 - F_x^{\mathcal{T} h}(t, \omega) F_x^{\mathcal{D} h}(t, \omega)}{F_x^{\mathcal{T} h}(t, \omega)^2 - F_x^{\mathcal{T}^2 h}(t, \omega) F_x^h(t, \omega)}, \quad (12)$$

and

$$\hat{t}_x(t, \omega) = t - \frac{F_x^{\mathcal{T} h}(t, \omega)}{F_x^h(t, \omega)}. \quad (13)$$

The MATLAB-based implementation of the presented methods is available under the link: <https://github.com/kabratkiewicz/Time-frequency-toolbox>

### III. IMPLEMENTATION

As mentioned in the previous section, both presented algorithms (CVSS3 and EVSS1) may be computed in two different ways. The differences from the signal processing point of view are not relevant, but from the implementation point of view, they are crucial and lie in the way of computing the modified STFTs required in (5), (6), (10), (11), and (12). In the first approach, all STFTs are computed using FFT, while in the second approach, STFT samples are computed element by element according to the formula (1). The great advantage of the first approach is the possibility of using the highly efficient cuFFT library. On the other hand, the second one allows all computations to be combined within one kernel function<sup>1</sup>, thus reducing the overhead associated with calling several kernel functions. It also allows better use of the various types of CUDA memories. As the advantages of both approaches are significant, it was decided to implement both versions to check which one would be more efficient.

Fig. 1 shows the general flowchart of the algorithm considering where the computations are performed. As can be seen, in the first step, regardless of whether the computations are performed using FFT or not, to calculate the synchrosqueezed STFT it is necessary to compute all modified STFTs. This requires all modified windows given by  $\mathcal{D}^n h = \frac{d^n h(t)}{dt^n}$  and  $\mathcal{T}^n h = t^n \cdot h(t)$  to be calculated.

#### A. Modified window computation

As the window samples are computed only once for a given window width, these computations are not crucial for software performance. Thus, they are performed in the prologue on a central processing unit<sup>2</sup> (CPU). Then, the calculated window coefficients are copied to the graphics processing unit<sup>3</sup> (GPU) memory. In the CUDA architecture, two types of memory are suitable for this purpose: constant and global. Due to its features, constant memory was chosen for this purpose.

#### B. Copying input signal between host and device

Then, the input signal is transferred to the global memory. The need to transfer data between the CPU and GPU is one of the fundamental GPGPU drawbacks<sup>4</sup>. Despite the very high throughput provided by the latest generations of the PCI Express bus, the enormous computing power of GPUs is partially lost on data transfers. Fortunately, in this case, the input signal is a one-dimensional vector of complex samples with a size typically not exceeding tens of kB. As a result, data transfer between the CPU and GPU takes relatively little time in relation to the computation time, and its negative impact is very limited in this case. All subsequent calculations are carried out on the device.

<sup>1</sup>A kernel function is a parallel subroutine executed by all threads according to single instruction multiple threads (SIMT) architecture [13].

<sup>2</sup>Typically, in CUDA technology, CPU is referred to as the host.

<sup>3</sup>Typically, in CUDA technology, GPU is referred to as the device.

<sup>4</sup>This issue does not occur in the Nvidia Tegra system on a chip device where CPU and GPU share the same memory space.

$$\begin{aligned}\hat{\omega}_x^N(t, \omega) = & F_x^{\mathcal{D}^3 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) - F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) - F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) + \\ & - F_x^{\mathcal{D}^3 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) + F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) + F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega),\end{aligned}\quad (5)$$

$$\begin{aligned}\hat{\omega}_x^D(t, \omega) = & F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) - F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) - F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) + \\ & F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) - F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) + F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega).\end{aligned}\quad (6)$$

$$\begin{aligned}\hat{p}_x^N(t, \omega) = & F_x^{\mathcal{T} h}(t, \omega) \left[ F_x^{\mathcal{D}^2 h}(t, \omega) \right]^2 - F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{T} h}(t, \omega) - F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{T} h}(t, \omega) F_x^{\mathcal{D}^2 h}(t, \omega) + \\ & F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) \left[ F_x^{\mathcal{D}^2 h}(t, \omega) \right]^2 - F_x^{\mathcal{D}^3 h}(t, \omega) F_x^{\mathcal{T} h}(t, \omega) F_x^{\mathcal{D}^2 h}(t, \omega) + F_x^{\mathcal{D}^3 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{T} h}(t, \omega),\end{aligned}\quad (10)$$

$$\begin{aligned}\hat{p}_x^D(t, \omega) = & F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) - F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) - F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) + \\ & F_x^{\mathcal{D}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) - F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega) + F_x^{\mathcal{D}^2 \mathcal{T}^2 h}(t, \omega) F_x^{\mathcal{D}^2 \mathcal{T} h}(t, \omega) F_x^{\mathcal{T}^2 h}(t, \omega).\end{aligned}\quad (11)$$

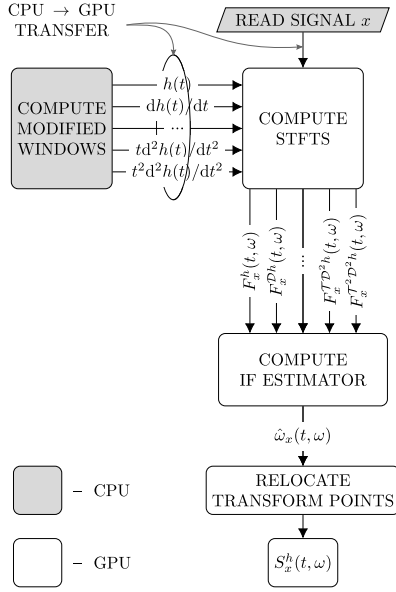


Fig. 1. EVSS1/CVSS3 algorithm flowchart

### C. GPU computation

As mentioned before, there are two possible approaches to computing necessary STFTs and, as a result, two variants of implemented algorithms. First, an algorithm that is based on a convolutional approach (1) to computing STFTs is described.

1) *Convolutional approach*: The convolutional version of the algorithm allows all the GPU calculations presented in Fig. 1 to be performed by the dedicated single kernel function. Such an approach minimizes the related overhead.

The most important task in the calculation of VSS given by (3) is to compute the frequency reassignment operator  $\hat{\omega}(t, \omega)$ . Thus a relatively natural way of decomposing the problem of VSS is to assign to each CUDA thread exactly one IF estimate. Such an approach requires the use of a two-dimensional grid of threads. It has been decided that the X-dimension will correspond to the discrete-time and Y-dimension to angular frequency. This causes data transfer to be coalesced.

Another choice that had to be made was the configuration of the kernel function. In CUDA technology, all threads are grouped in thread blocks. All threads within the same thread block share resources such as registers or an L1 cache. Moreover, they can exchange data with each other using shared

memory<sup>5</sup> and synchronize [13]. Considering the arrangement of the input and intermediate data and the possibility of sharing the calculated harmonic signal ( $e^{-j\omega\tau}$  in (1)) among all threads within the block, it was decided that the thread blocks would be one-dimensional with Y-dimension equal to one (one FFT bin).

The code of the kernel function itself can be divided into several stages (Fig. 1). The first one is responsible for calculating the samples of all modified STFTs that appear in formulas (5), (6), (7), (10), (11), and (12).

First, the input signal fragments needed to perform calculations by the whole thread block are transferred from the global to the shared memory. Although they are required later in the algorithm, downloading them now helps to hide latencies resulting from access to global memory.

Then the harmonic signal is computed and stored in the shared memory. As already mentioned, one thread block performs computations for one frequency bin, which means that one thread block requires only one harmonic signal. Such dimensionality of thread blocks minimizes the use of the shared memory. Moreover, a single thread computes at most one harmonic signal sample for typical window lengths. Thus computing and using a harmonic signal is efficient.

The following two steps of windowing the harmonic signal and calculating the dot product are tied together. The windowed harmonic signal sample is first computed and stored in the shared memory in the inner loop. After synchronization between all threads within the block, the dot product (single STFT sample) is calculated in the next loop. These procedures are repeated in the outer loop for all modified windows. As a result, one thread computes a one-dimensional array of modified STFTs samples (for a single moment of discrete-time and single frequency bin). Since only the same thread will use these samples in further calculations, and the modified window number is not high, it was decided that they will be stored in registers<sup>6</sup>.

<sup>5</sup>Shared memory is the third type of CUDA memory. It is a user-controlled L1 cache [14] with greater bandwidth and much lower latency than the global memory.

<sup>6</sup>Registers are another type of memory in the CUDA architecture. They have the highest throughput, but their number is limited. Too high register utilization limits the number of thread blocks concurrently running on a streaming multiprocessor [15].

In this case, using register memory is the most crucial advantage of this version of the algorithm. Thanks to this, it is possible to obtain a very high compute-to-global memory access (CGMA) ratio<sup>7</sup> and, as a result, very high performance.

The second stage of the kernel function (Fig. 1) is responsible for computing the IF estimates. This means that this is the only place where there are differences between the two presented forms of VSS. In the case of CVSS3, each thread computes a single frequency reassignment operator given by (7), the second-order modulation operator given by (12), and the third-order modulation operator given by (9). The modified IF estimate, given by (8), is then computed. In contrast, for the EVSS1 form, only the enhanced IF estimates defined in (4) are computed. Again, the results, separate for each thread, are stored in the register memory.

The transformation points are relocated in the last step, resulting in the VSS defined in (3). In practice, it comes down to adding the product  $F_x^h(t, \omega')e^{j\omega'(t-t_0)}$  to the output array element with an address along the X-axis that corresponds to the thread X-dimension identifier and along the Y-axis with an address that corresponds to the estimated angular frequency. This means that many concurrent threads can add partial results to the same global array elements. Thus, the atomic operations that serialize all updates to the same locations must be applied to ensure the correctness of the final result. As is known, serializing any part of a parallel program can significantly reduce the execution speed. Moreover, many atomic operations can also make it impossible to hide latencies in access to global memory [16]. Fortunately, serialization only takes a small fraction of the code at the end of the kernel's function, resulting in little performance impact.

2) *FFT approach*: In the first step, the window-length segments of the input signal are multiplied by modified windows. The dedicated optimized kernel function has been created for this operation. Zero padding is done by zeroing the whole data block during allocation and appropriate memory addressing within the kernel function. This is possible thanks to the assumption that neither the window size nor the FFT size changes during a single program run. It should be emphasized here that the data are stored continuously. Thus subsequent FFT calculations are performed by one execution of the `cufftExecC2C()` function in a batched manner.

The last stage of calculations performed on the GPU, the IF estimator computation and relocation of transform points, is carried out using a dedicated kernel function. Its code is identical as in the case of the second and third stages of the kernel function described in the previous subsection. The only differences are that the input data is in the global memory, not in registers, and it is necessary to transpose each of the modified STFTs because of the manner they are stored for cuFFT library.

<sup>7</sup>CGMA ratio informs how many arithmetic operations are performed per one global memory transfer operation, which has a significant impact on computational performance.

#### D. Copying results between device and host

The last stage of the program depends on the adopted execution path. If the data is further processed on the GPU, other kernel or library functions are called here. If the data is processed on the CPU, it must first be copied from the GPU to the CPU. In this case, transfer operation has a negative impact on the overall software performance. This is because the output data size depends not only on the signal length but also on the number of frequency bins. Thus the output data size may be several orders of magnitude larger (typically in the order of tens of MB) than the input data size. As a result, copying data from the GPU to the CPU can take as long as computations.

Fortunately, in CUDA technology, there is a so-called CUDA streams mechanism that partially mitigates this disadvantage. This optimization technique introduces additional parallelism. It allows kernel execution and asynchronous data transfer to overlap, significantly reducing the overall computation time [13].

## IV. RESULTS

A detailed discussion of the results, from the signal processing point of view, of both the CVSS3 and the EVSS1 form of synchrosqueezing can be found in [2]. Therefore, this aspect of the presented algorithms will be discussed briefly. The signal selected for validation tests was a non-linear frequency-modulated (NLFM) pulse originating from the S-Band Air Traffic Control radar at Warsaw's Chopin Airport. Fig. 2a shows the spectrogram obtained using classical STFT. Fig. 2b shows the result of the convolutional version of the CVSS3 algorithm. A clear NLFM pulse is apparent, allowing the direct pulse to be distinguished. Since the local transform points were relocated to the signal instantaneous frequency ridge, the distribution is more sharp and readable than in the case of classical STFT. It should be noted here that the EVSS1 algorithm gives almost the same results in this case, while the FFT-based versions provide the same results apart from the numerical issues, and therefore, there is no need to present their visual results.

The correctness of the calculations is proven by the high reconstruction quality factor (RQF) [7] values, which for the selected signal are 45 dB for CVSS3 and 45.7 dB for EVSS1, respectively, while for the calculations performed in Matlab, they are 45.3 dB and 45.9 dB, respectively.

However, in the presented work, the main emphasis is on examining the performance of the described real-time CUDA implementations of the presented algorithms. All calculations were performed on single-precision floating point numbers. The performance tests were carried out on the RTX 4070 Super GPU connected to CPU via the PCI-E 4.0 x16 bus, running under the Ubuntu OS and CUDA SDK 12.8. During the tests, three tools were used to measure the time and validate the measurements: the CUDA event mechanism, Nvidia Visual Profiler (nvprof), and high-resolution CPU timer from the real-time extensions library. All the computations were called repeatedly. Then the shortest execution time was chosen to

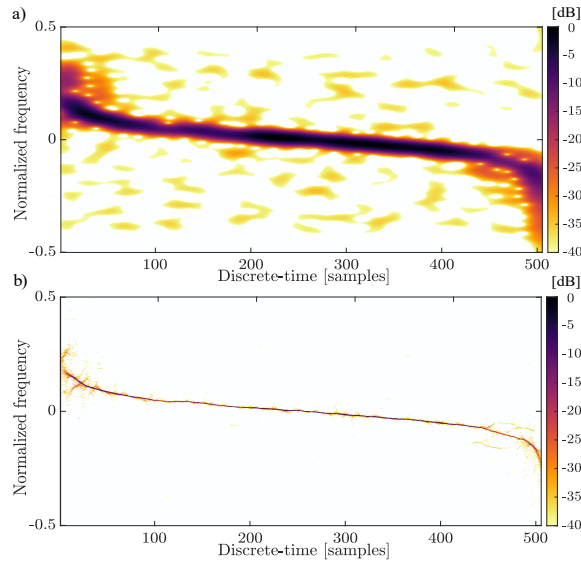


Fig. 2. Test NLFM signal: a) Spectrogram of the test signal obtained with classical STFT; b) Spectrogram of the test signal obtained with CVSS3

become independent from the instantaneous load changes of both the CPU and the GPU.

As it turned out, for typical processing parameters, i.e., FFT length less than 10000 and window length less than 100, the convolutional versions of both algorithms proved to be more efficient than the FFT-based. In addition, the CVSS3 version is slightly slower than EVSS1. As a result, performance tests were conducted for the convolutional version of CVSS3.

The execution time tests were performed for one typical case with a signal length of 16384 samples. Both processing time and data copy time between host and device were considered. Table I shows the maximum signal sampling rate expressed in million samples per second (Sa/s) for various FFT and window lengths, enabling real-time processing.

TABLE I  
MAXIMUM SAMPLING RATE FOR REAL-TIME SIGNAL PROCESSING.

Sampling rate [MSa/s]		FFT length				
		512	1024	2048	4096	8192
Window length	32	2.56	1.38	0.67	0.34	0.17
	48	2.14	1.14	0.56	0.28	0.14
	64	1.97	0.96	0.48	0.24	0.12
	80	1.68	0.85	0.42	0.21	0.11
	96	1.49	0.74	0.37	0.18	0.09

As can be seen, it is possible to process signals in real-time with sampling rates exceeding 2 MSa/s but at the price of lower frequency resolution and shorter window lengths. On the other hand, for typical processing parameters, signal sampling rates higher than 500 kSa/s are applicable.

## V. CONCLUDING REMARKS

This article has presented the real-time implementations of two variants of vertical synchrosqueezing – CVSS3 and EVSS1. In this case, real-time processing is possible thanks

to utilizing Nvidia CUDA technology. To the authors' knowledge, these are the first real-time implementations of synchrosqueezing introduced in the literature.

In-depth tests and performance studies revealed that these algorithms ideally map to the CUDA architecture. This is evidenced by the high computing performance achieved. As a result, for the GPU selected for tests, it is possible to process signals in real-time with a sampling rate of hundreds of kSa/s and up to 2 MSa/s. This is an excellent result, especially considering how computationally complex the presented algorithms are.

Further research covers implementing the presented algorithms on embedded systems like Nvidia Jetson platforms to check whether it is possible to use the proposed algorithms in portable devices, for example, to monitor vital signs.

## REFERENCES

- [1] K. Kodera, C. De Villedary, and R. Gendrin, "A new method for the numerical analysis of non-stationary signals," *Physics of the Earth and Planetary Interiors*, vol. 12, no. 2, pp. 142–150, 1976.
- [2] K. Abratkiewicz and J. Gambrych, "Real-Time Variants of Vertical Synchrosqueezing: Application to Radar Remote Sensing," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 15, pp. 1760–1774, 2022.
- [3] K. Abratkiewicz and P. Samczyński, "Radar Pulse Signal Filtering Using Vertical Synchrosqueezing," in *2022 IEEE Radar Conference (RadarConf22)*, 2022, pp. 1–6.
- [4] J. M. Miramont, M. A. Colominas, and G. Schlotthauer, "Voice Jitter Estimation Using High-Order Synchrosqueezing Operators," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 29, pp. 527–536, 2021.
- [5] W. Liu, S. Cao, Z. Wang, K. Jiang, Q. Zhang, and Y. Chen, "A Novel Approach for Seismic Time-Frequency Analysis Based on High-Order Synchrosqueezing Transform," *IEEE Geoscience and Remote Sensing Letters*, vol. 15, no. 8, pp. 1159–1163, 2018.
- [6] S. Wang, X. Chen, C. Tong, and Z. Zhao, "Matching Synchrosqueezing Wavelet Transform and Application to Aeroengine Vibration Monitoring," *IEEE Transactions on Instrumentation and Measurement*, vol. 66, no. 2, pp. 360–372, 2017.
- [7] D. Pham and S. Meignen, "High-Order Synchrosqueezing Transform for Multicomponent Signals Analysis—With an Application to Gravitational-Wave Signal," *IEEE Transactions on Signal Processing*, vol. 65, no. 12, pp. 3168–3178, 2017.
- [8] S. Meignen, T. Oberlin, and D. Pham, "Synchrosqueezing transforms: From low- to high-frequency modulations and perspectives," *Comptes Rendus Physique*, vol. 20, no. 5, pp. 449–460, July 2019.
- [9] J. Gambrych, "Influence of optimization techniques on software performance for subsequent generations of CUDA architecture," 2021, pp. 1002–1009.
- [10] M. Rupniewski, G. Mazurek, J. Gambrych, M. Nalecz, and R. Karolewski, "A Real-Time Embedded Heterogeneous GPU/FPGA Parallel System for Radar Signal Processing," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCCom/IoP/SmartWorld)*, 2016, pp. 1189–1197.
- [11] J. Kowalczyk, E. T. Psota, and L. C. Perez, "Real-Time Stereo Matching on CUDA Using an Iterative Refinement Method for Adaptive Support-Weight Correspondences," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, no. 1, pp. 94–104, 2013.
- [12] P. Flandrin, *Explorations in Time-Frequency Analysis*. Cambridge University Press, 2018.
- [13] NVIDIA Corporation, *CUDA C++ Programming Guide*. NVIDIA Corporation, 2021.
- [14] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. M. Kaufmann, 2012.
- [15] NVIDIA Corporation, *CUDA C++ Best Practices Guide*. NVIDIA Corporation, 2021.
- [16] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufmann, 2013.