Algorithms for Efficient Computation of Convolution

Karas Pavel and Svoboda David

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/51942

1. Introduction

Convolution is an important mathematical tool in both fields of signal and image processing. It is employed in filtering [1, 2], denoising [3], edge detection [4, 5], correlation [6], compression [7, 8], deconvolution [9, 10], simulation [11, 12], and in many other applications. Although the concept of convolution is not new, the efficient computation of convolution is still an open topic. As the amount of processed data is constantly increasing, there is considerable request for fast manipulation with huge data. Moreover, there is demand for fast algorithms which can exploit computational power of modern parallel architectures.

The basic convolution algorithm evaluates inner product of a flipped kernel and a neighborhood of each individual sample of an input signal. Although the time complexity of the algorithms based on this approach is quadratic, i.e. $O(N^2)$ [13, 14], the practical implementation is very slow. This is true especially for higher-dimensional tasks, where each new dimension worsens the complexity by increasing the degree of polynomial, i.e. $O(N^{2k})$. Thanks to its simplicity, the naïve algorithms are popular to be implemented on parallel architectures [15–17], yet the use of implementations is generally limited to small kernel sizes. Under some circumstances, the convolution can be computed faster than as mentioned in the text above.

In the case the higher dimensional convolution kernel is *separable* [18, 19], it can be decomposed into several lower dimensional kernels. In this sense, a 2-D separable kernel can be split into two 1-D kernels, for example. Due to the associativity of convolution, the input signal can be convolved step by step, first with one 1-D kernel, then with the second 1-D kernel. The result equals to the convolution of the input signal with the original 2-D kernel. Gaussian, Difference of Gaussian, and Sobel are the representatives of separable kernels commonly used in signal and image processing. Respecting the time complexity, this approach keeps the higher dimensional convolution to be a polynomial of lower degree, i.e. $O(kN^{k+1})$. On the other hand, there is a nontrivial group of algorithms that use general kernels. For example, the deconvolution or the template matching algorithms based on correlation methods typically use kernels, which cannot be characterized by special properties like separability. In this case, other convolution methods have to be used.

There also exist algorithms that can perform convolution in time $O(N)$. In this concept, the repetitive application of convolution kernel is reduced due to the fact that neighbouring positions overlap. Hence, the convolution in each individual sample is obtained as a weighted sum of both input samples and previously computed output samples. The design of so called *recursive filters* [18] allows them to be implemented efficiently on streaming architectures such as FPGA. Mostly, the recursive filters are not designed from scratch. Rather the well-known 1-D filters (Gaussian, Difference of Gaussian, $\ldots$) are converted into their recursive form. The extenstion to higher dimension is straighforward due to their separability. Also this method has its drawbacks. The conversion of general convolution kernel into its recursive version is a nontrivial task. Moreover, the recursive filtering often suffers from inaccuracy and instability [2].

While the convolution in time domain performs an inner product in each sample, in the Fourier domain [20], it can be computed as a simple point-wise multiplication. Due to this convolution property and the fast Fourier transform the convolution can be performed in time $O(N \log N)$. This approach is known as a *fast convolution* [1]. The main advantage of this method stems in the fact that no restrictions are imposed on the kernel. On the other hand, the excessive memory requirements make this approach not very popular. Fortunately, there exists a workaround: If a direct computation of fast convolution of larger signals or images is not realizable using common computers one can reduce the whole problem to several subtasks. In practice, this leads to splitting the signal and kernel into smaller pieces. The signal and kernel decomposition can be perfomed in two ways:

- Data can be decomposed in Fourier domain using so-called decimation-in-frequency (DIF) algorithm [1, 21]. The division of a signal and a kernel into smaller parts also offers a straightforward way of parallelizing the whole task.
- Data can be split in time domain according to overlap-save and overlap-add scheme [22, 23], respectively. Combining these two schemes with fast convolution one can receive a quasi-optimal solution that can be efficiently computed on any computer. Again, the solution naturally leads to a possible parallelization.

The aim of this chapter is to review the algorithms and approaches for computation of convolution with regards to various properties such as signal and kernel size or kernel separability (when processing k-dimensional signals). Target architectures include superscalar and parallel processing units (namely CPU, DSP, and GPU), programmable architectures (e.g. FPGA), and distributed systems (such as grids). The structure of the chapter is designed to cover various applications with respect to the signal size, from small to large scales.

In the first part, the state-of-the-art algorithms will be revised, namely (i) naïve approach, (ii) convolution with separable kernel, (iii) recursive filtering, and (iv) convolution in the frequency domain. In the second part, will be described convolution decomposition in both the spatial and the frequency domain and its implementation on a parallel architecture.

1.1. Shortcuts and symbols

In the following list you will find the most commonly used symbols in this chapter. We recommend you to go through it first to avoid some misunderstanding during reading the text.

- $\mathcal{F}[.]$, $\mathcal{F}^{-1}[.]$ ... Fourier transform and inverse Fourier transform of a signal, respectively
- W_k^i, W_k^{-i} ... k-th sample of i-th Fourier transform base function and inverse Fourier transform base function, respectively
- z^* ... complex conjugate of complex number z
- $*$ ... symbol for convolution
- e ... Euler number (e ≈ 2.718)
- j ... complex unit ($j^2 = -1$)
- f, g ... input signal and convolution kernel, respectively
- h ... convolved signal
- F, G ... Fourier transforms of input signal f and convolution kernel g, respectively
- N^f, N^g ... length of input signal and convolution kernel, respectively (number of samples)
- n, k ... index of a signal in the spatial and the frequency domain, respectively
- n', k' ... index of a signal of half length in the spatial and the frequency domain, respectively
- P ... number of processing units in use
- Φ ... computational complexity function
- $||s||$ ... number of samples of a discrete signal (sequence) s

2. Naïve approach

First of all, let us recall the basic definition of convolution:

$$h(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\mathrm{d}\tau. \tag{1}$$

Respecting the fact that Eq. (1) is used mainly in the fields of research different from image and signal procesing we will focus on the alternative definition that the reader is likely to be more familiar with—the dicrete signals:

$$h(n) = (f * g)(n) = \sum_{i=-\infty}^{\infty} f(n-i)g(i). \tag{2}$$

The basic (or *naïve*) approach visits the individual time samples n in the input signal f. In each position, it computes inner product of current sample neighbourhood and flipped kernel g, where the size of the neighbourhood is practically equal to the size of the convolution kernel. The result of this inner product is a number which is simply stored into the position n in the output signal h. It is noteworthy that according to the definition (2), the size of output signal h is always equal or greater than the size of the input signal f. This fact is related to the boundary conditions. Let $f(n) = 0$ for all $n < 0 \vee n > N^f$ and also $g(n) = 0$ for all $n < 0 \vee n > N^g$. Then computing the expression (2) at the position $n = -1$ likely gives non-zero value, i.e. the output signal becomes larger. It can be derived that the size of output signal h is equal to $N^f + N^g - 1$.

2.0.0.1. Analysis of time complexity.

For the computation of $f * g$ we need to perform $N^f N^g$ multiplications. The computational complexity of this algorithm is polynomial [13], but we must keep in mind what happens when the N^f and N^g become larger and namely what happens when we extend the computation into higher dimensions. In the 3-D case, for example, the expression (2) is slightly changed:

$$\begin{aligned} h^{3d}(n_x, n_y, n_z) &= \left(f^{3d} * g^{3d}\right)(n_x, n_y, n_z) \\ &= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} f^{3d}(n_x - i, n_y - j, n_z - k) g^{3d}(i, j, k) \end{aligned} \tag{3}$$

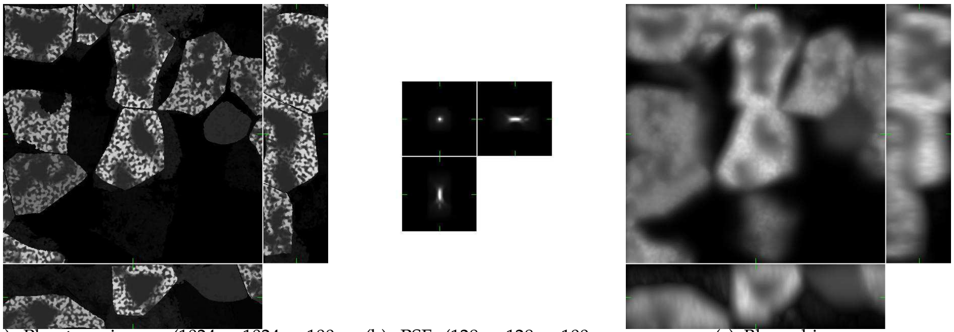
Here, f^{3d}, g^{3d} and h^{3d} have the similar meaning as in (2). If we assume $||f^{3d}|| = N_x^f \times N_y^f \times N_z^f$ and $||g^{3d}|| = N_x^g \times N_y^g \times N_z^g$, the complexity of our filtering will raise from $N^f N^g$ in the 1-D case to $N_x^f N_y^f N_z^f N_x^g N_y^g N_z^g$, which is unusable for larger signals or kernels. Hence, for higher dimensional tasks the use of this approach is becomes impractical, as each dimension increases the degree of this polynomial. Although the time complexity of this algorithm is polynomial the use of this solution is advantageous only if we handle with kernels with a small support. An example of such kernels are well-known filters from signal/image processing:

$$\underset{\text{Sobel}}{\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}} \quad \underset{\text{Gaussian}}{\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}}$$

For better insight, let us consider the convolution of two relatively small 3-D signals $1024 \times 1024 \times 100$ voxels and $128 \times 128 \times 100$ voxels—the example is shown in Fig. 1. When this convolution was performed in double precision on Intel Xeon QuadCore 2.83 GHz computer it lasted cca for 7 days if the computation was based on the basic approach.

2.0.0.2. Parallelization.

Due to its simplicity and no specific restrictions, the naïve convolution is still the most popular approach. Its computation is usually sped up by employing large computer clusters that significantly decrease the time complexity per one computer. This approach [15–17] assumes the availability of some computer cluster, however.



(a) Phantom image (1024 × 1024 × 100 pixels) (b) PSF (128 × 128 × 100 pixels) (c) Blurred image

Figure 1. Example of a 3-D convolution. The images show an artificial (phantom) image of a tissue, a PSF of an optical microscope, and blurred image, computed by the convolution of the two images. Each 3-D image is represented by three 2-D views (XY, YZ, and XZ).

2.1. Convolution on a custom hardware

Dedicated and configurable hardware, namely digital signal processors (DSP) or Field-programmable gate array (FPGA) units are very popular in the field of signal processing for their promising computational power at both low cost and low power consumption. Although the approach based on the Fourier transform is more popular in digital signal processing for its ability to process enormously long signals, the naïve convolution with a small convolution kernel on various architectures has been also well studied in the literature, especially in the context of the 2-D and multi-dimensional convolution.

Shoup [24] proposed techniques for automatic generation of convolution pipelines for small kernels such as of 3×3 pixels. Benedetti et al. [25] proposed a multi-FPGA solution by using an external memory to store a FIFO buffer and partitioning of data among several FPGA units, allowing to increase the size of the convolution kernel. Perri et al. [26] followed the previous work by designing a fully reconfigurable FPGA-based 2-D convolution processor. The core of this processor contains four 16-bit SIMD 3×3 convolvers, allowing real-time computation of convolution of a 8-bit or 16-bit image with a 3×3 or 5×5 convolution kernel. Recently, convolution on a custom specialized hardware, e.g. FPGA, ASIC, and DSP, is used to detect objects [27], edges [28], and other features in various real-time applications.

2.2. GPU-based convolution

From the beginning, graphics processing units (GPU) had been designed for visualisation purposes. Since the beginning of the 21st century, they started to play a role in general computations. This phenomenon is often referred to as general-purpose computing on graphics processing units (GPGPU) [29]. At first, there used to be no high-level programming languages specifically designed for general computation purposes. The programmers instead had to use shading languages such as Cg, High Level Shading Language (HLSL) or OpenGL Shading Language (GLSL) [29–31], to utilize texture units. Recently, two programming frameworks are widely used among the GPGPU community, namely CUDA [32] and OpenCL [33].

For their ability to efficiently process 2-D and 3-D images and videos, GPUs have been utilized in various image processing applications, including those based on the convolution. Several convolution algorithms including the naïve one are included in the CUDA Computing SDK [34]. The naïve convolution on the graphics hardware has been also described in [35] and included in the Nvidia Performance Primitives library [36]. Specific applications, namely Canny edge detection [37, 38] or real-time object detection[39] have been studied in the literature. It can be noted that the problem of computing a rank filter such as the median filter has a naïve solution similar to the one of the convolution. Examples can be found in the aforementioned CUDA SDK or in [40, 41].

Basically, the convolution is a *memory-bound* problem [42], i.e. the ratio between the arithmetic operations and memory accesses is low. The adjacent threads process the adjacent signal samples including the common neighbourhood. Hence, they should share the data via a faster memory space, e.g. *shared memory* [35]. To store input data, programmers can also use *texture memory* which is read-only but cached. Furthermore, the texture cache exhibits the 2-D locality which makes it naturally suitable especially for 2-D convolutions.

3. Separable convolution

3.1. Separable convolution

The naïve algorithm is of polynomial complexity. Furthermore, with each added dimension the polynomial degree raises linearly which leads to very expensive computation of convolution in higher dimensions. Fortunately, some kernels are so called *separable* [18, 19]. The convolution with these kernels can be simply decomposed into several lower dimensional (let us say "cheaper") convolutions. Gaussian and Sobel [4] are the representatives of such group of kernels.

Separable convolution kernel must fullfil the condition that its matrix has rank equal to one. In other words, all the rows must be linearly dependent. Why? Let us construct such a kernel. Given one row vector

$$\vec{u} = (u_1, u_2, u_3, \ldots, u_m)$$

and one column vector

$$\vec{v}^T = (v_1, v_2, v_3, \ldots, v_n)$$

let us convolve them together:

$$\vec{u} * \vec{v} = (u_1, u_2, u_3, \ldots, u_m) * \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} u_1v_1 & u_2v_1 & u_3v_1 & \ldots & u_mv_1 \\ u_1v_2 & u_2v_2 & u_3v_2 & \ldots & u_mv_2 \\ u_1v_3 & u_2v_3 & u_3v_3 & \ldots & u_mv_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_1v_n & u_2v_n & u_3v_n & \ldots & u_mv_n \end{pmatrix} = A \tag{4}$$

It is clear that $rank(A) = 1$. Here, A is a matrix representing some separable convolution kernel while $\vec{u}$ and $\vec{v}$ are the previously referred lower dimensional (cheaper) convolution kernels.

3.1.0.3. Analysis of Time Complexity.

In the previous section, we derived the complexity of naïve approach. We also explained how the complexity worsens when we increase the dimensionality of the processed data. In case the convolution kernel is separable we can split the hard problem into a sequence of several simpler problems. Let us recall the 3-D naïve convolution from (3). Assume that g^{3d} is separable, i.e. $g^{3d} = g_x * g_y * g_z$. Then the expression is simplified in the following way:

$$\begin{aligned} h^{3d}(n_x, n_y, n_z) &= \left(f^{3d} * g^{3d}\right)(n_x, n_y, n_z) && (5)\\ &= \left(f^{3d} * \left(g_x * g_y * g_z\right)\right)(n_x, n_y, n_z) \quad /associativity/ && (6)\\ &= \left(\left(\left(f^{3d} * g_x\right) * g_y\right) * g_z\right)(n_x, n_y, n_z) && (7)\\ &= \left(\sum_{i=-\infty}^{\infty}\left(\sum_{j=-\infty}^{\infty}\left(\sum_{k=-\infty}^{\infty} f^{3d}(n_x-i, n_y-j, n_z-k) g_z(k)\right) g_y(j)\right) g_x(i)\right) && (8) \end{aligned}$$

The complexity of such algorithm is then reduced from $N_x^f N_y^f N_z^f N_x^g N_y^g N_z^g$ to $N_x^f N_y^f N_z^f \left(N_x^g + N_y^g + N_z^g\right)$.

One should keep in mind that the kernel decomposition is usually the only one decomposition that can be performed in this task. It is based on the fact that many well-known kernels (Gaussian, Sobel) have some special properties. Nevertheless, the input signal is typically unpredictable and in higher dimensional cases it is unlikely one could separate it into individual lower-dimensional signals.

3.2. Separable convolution on various architectures

As separable filters are very popular in many applications, a number of implementations on various architectures can be found in the literature. Among the most favourite filters, the Gaussian filter is often used for pre-processing, for example in optical flow applications [43, 44]. Fialka et al. [45] compared the separable and the fast convolution on the graphics hardware and proved both the kernel size and separability to be the essential properties that have to be considered when choosing an appropriate implementation. They proved the separable convolution to be more efficient for kernel sizes up to tens of pixels in each dimension which is usually sufficient if the convolution is used for the pre-processing.

The implementation usually does not require particular optimizations as the separable convolution is intrinsically a sequence of 1-D basic convolutions. Programmers should nevertheless consider some tuning steps regarding the memory accesses, as mentioned in Section 2.2. For the case of a GPU implementation, this issue is discussed in [35]. The GPU implementation described in the document is also included in the CUDA SDK [34].

4. Recursive filtering

The convolution is a process where the inner product, whose size corresponds to kernel size, is computed again and again in each individual sample. One of the vectors (kernel), that enter this operation, is always the same. It is clear that we could compute the whole inner product only in one position while the neighbouring position can be computed as a slightly modified difference with respect to the first position. Analogously, the same is valid for all the following positions. The computation of the convolution using this difference-based approach is called *recursive filtering* [2, 18].

4.0.0.4. Example.

The well-known pure averaging filter in 1D is defined as follows:

$$h(n) = \sum_{i=0}^{n-1} f(n-i) \tag{9}$$

The performance of this filter worsen with the width of its support. Fortunately, there exists a recursive version of this filter with constant complexity regardless the size of its support. Such a filter is no more defined via standard convolution but using the recursive formula:

$$h(n) = h(n-1) + f(n) - f(n-n) \tag{10}$$

The transform of standard convolution into a recursive filtering is not a simple task. There are three main issues that should be solved:

1. replication – given slow (but correctly working) non-recursive filter, find its recursive version
2. stability – the recursive formula may cause the computation to diverge
3. accuracy – the recursion may cause the accumulation of small errors

The transform is a quite complex task and so-called Z-transform [22] is typically employed in this process. Each recursive filter may be designed as all other filters from scratch. In practice, the standard well-known filters are used as the bases and subsequently their recursive counterpart is found. There are two principal approaches how to do it:

- analytically – the filter is step by step constructed via the math formulas [46]
- numerically – the filter is derived using numerical methods [47, 48]

4.1. Recursive filters on various architectures

Streaming architectures.

The recursive filtering is a popular approach especially on streaming architectures such as FPGA. The data can be processed in a stream keeping the memory requirements on a minimum level. This allows moving the computation to relatively small and cheap embedded systems. The recursive filters are thus used in various real-time applications such as edge detection [49], video filtering [50], and optical flow [51].

Parallel architectures.

As for the parallel architectures, Robelley et al. [52] presented a mathematical formulation for computing time-invariant recursive filters on general SIMD DSP architectures. Authors also discuss the speed-up factor regarding to the level of parallelism and the filter order. Among the GPU implementations, we can mention the work of Trebien and Oliveira who implemented recursive filters in CUDA for the purpose of the realistic sound synthesis and processing [53]. In this case, recursive filters were computed in the frequency domain.

5. Fast convolution

In the previous sections, we have introduced the common approaches to compute the convolution in the time (spatial) domain. We mentioned that in some applications, one has to cope with signals of millions of samples where the computation of the convolution requires too much time. Hence, for long or multi-dimensional input signals, the popular approach is to compute the convolution in the frequency domain which is sometimes referred to as the *fast convolution*. As shown in [45], the fast convolution can be even more efficient than the separable version if the number of kernel samples is large enough. Although the concept of the fast Fourier transform [54] and the frequency-based convolution [55] is several decades old, with new architectures upcoming, one has to deal with new problems. For example, the efficient access to the memory was an important issue in 1970s [56] just as it is today [21, 23]. Another problem to be considered is the numerical precision [57].

In the following text, we will first recall the Fourier transform along with some of its important properties and the convolution theorem which provides us with a powerful tool for the convolution computation. Subsequently, we will describe the algorithm of the so-called fast Fourier transform, often simply denoted as FFT, and mention some notable implementations of the FFT. Finally, we will summarize the benefits and drawbacks of the fast convolution.

5.1. Fourier transform

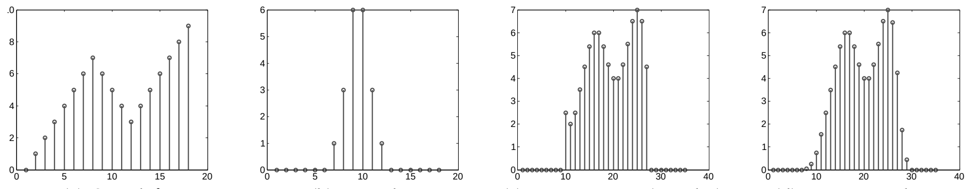
The Fourier transform $F = \mathcal{F}[f]$ of a function f and the inverse Fourier transform $f = \mathcal{F}^{-1}[F]$ are defined as follows:

$$F(\omega) \equiv \int_{-\infty}^{+\infty} f(t)\mathrm{e}^{-jt\omega}\mathrm{d}t, \qquad f(t) \equiv \frac{1}{2\pi}\int_{-\infty}^{+\infty} F(\omega)\mathrm{e}^{j\omega t}\mathrm{d}\omega. \tag{11}$$

The discrete finite equivalents of the aforementioned transforms are defined as follows:

$$F(k) \equiv \sum_{n=0}^{N-1} f(n)\mathrm{e}^{-j(2\pi/N)nk}, \qquad f(n) \equiv \frac{1}{N}\sum_{k=0}^{N-1} F(k)\mathrm{e}^{j(2\pi/N)kn} \tag{12}$$

where $k, n = 0, 1, \ldots, N-1$. The so-called normalization factors $\frac{1}{2\pi}$ and $\frac{1}{N}$, respectively, guarantee that the identity $f = \mathcal{F}^{-1}[\mathcal{F}[f]]$ is maintained. The exponential function $\mathrm{e}^{-j(2\pi/N)}$ is called the base function. For the sake of simplicity, we will refer to it as W_K.



(a) Signal f (b) Kernel g (c) Fast (circular) convolution (d) Basic convolution

Figure 2. Example of the so-called windowing effect produced by signal f (a) and kernel g (b). The circular convolution causes border effects as seen in (c). The properly computed basic convolution is shown in (d).

If the sequence $f(n), n = 0, 1, \ldots, N-1$, is real, the discrete Fourier transform $F(k)$ keeps some specific properties, in particular:

$$F(k) = F(N-k)^*. \tag{13}$$

This means that in the output signal F, only half of the samples are useful, the rest is redundant. As the real signals are typical for many practical applications, in most popular FT and FFT implementations, users are hence provided with special functions to handle real signals in order to save time and memory.

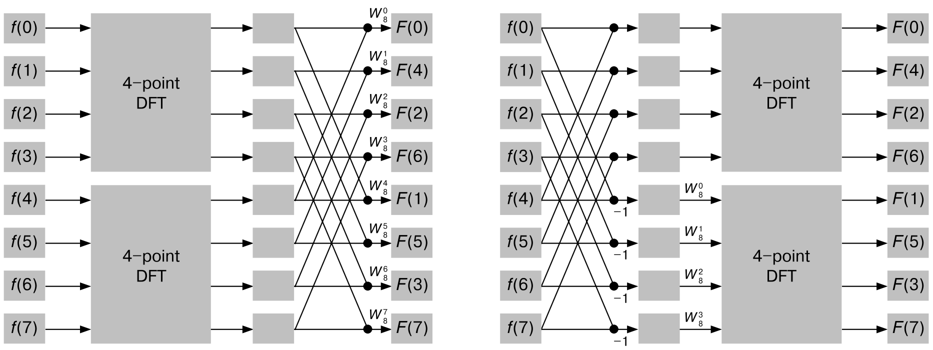
5.2. Convolution theorem

According to the convolution theorem, the Fourier transform convolution of two signals f and g is equal to the product of the Fourier transforms $\mathcal{F}[f]$ and $\mathcal{F}[g]$, respectively [58]:

$$\mathcal{F}[f * g] = \mathcal{F}[f]\,\mathcal{F}[g]\,. \tag{14}$$

In the following text, we will sometimes refer to the convolution computed by applying Eq. (14) as the "classical" fast convolution algorithm.

In the discrete case, the same holds for periodic signals (sequences) and is sometimes referred to as the circular or cyclic convolution [22]. However, in practical applications, one usually deals with non-periodic finite signals. This results into the so-called windowing problem [59], causing undesirable artefacts in the output signals—see Fig. 2. In practice, the problem is usually solved by either imposing the periodicity into the kernel, adding a so-called windowing function, or padding the kernel with zero values. One also has to consider the sizes of both the input signal and the convolution kernel which have to be equal. Generally, this is also solved by padding both the signal and the kernel with zero values. The size of both padded signals which enter the convolution is hence $N = N^f + N^g - 1$ where N^f and N^g is the number of signal and kernel samples, respectively. The equivalent property holds for the multi-dimensional case. The most time-demanding operation of the fast convolution approach is the Fourier transform which can be computed by the fast Fourier transform



(a) Decimation-in-time (b) Decimation-in-frequency

Figure 3. The basic two radix-2 FFT algorithms: decimation-in-time and decimation-in-frequency. Demonstration on an input signal of 8 samples.

algorithm. The time complexity of the fast convolution is hence equal to the complexity of the FFT, that is $O(N \log N)$. The detailed discussion on the complexity is provided in Section 6.

5.3. Fast Fourier transform

In 1965, Cooley and Tukey [60] proposed an algorithm for fast computation of the Fourier transform. The widely-known algorithm was then improved through years and optimized for various signal lengths but the basic idea remained the same. The problem is handled in a divide-and-conquer manner by splitting the input signal into N parts[1] and processing the individual parts recursively. Without loss of generality, we will recall the idea of the FFT for $N = 2$ which is the simplest situation. There are two fundamental approaches to split the signal. They are called *decimation in time (DIT)* and *decimation in frequency (DIF)* [58].

Decimation in time (DIT).

Assuming that N is even, the radix-2 decimation-in-time algorithm splits the input signal $f(n)$, $n = 0, 1, \ldots, N-1$ into parts $f_e(n')$ and $f_o(n')$, $n' = 0, 1, \ldots, N/2-1$ of even and odd samples, respectively. By recursive usage of the approach, the discrete Fourier transforms F_e and F_o of the two parts are computed. Finally, the resulting Fourier transform F can be computed as follows:

$$F(k) = F_e(k) + W_N^k F_o(k) \tag{15}$$

where $k = 0, 1, \ldots, N-1$. The signals F_e and F_o are of half length, however, they are periodic, hence

$$F_e(k' + N/2) = F_e(k'), \qquad F_o(k' + N/2) = F_o(k') \tag{16}$$

for any $k' = 0, 1, \ldots, N/2-1$. The algorithm is shown in Fig. 3(a).

[1] The individual variants of the algorithm for a particular N are called radix-N algorithms.

Decimation in frequency (DIF).

Having the signal f of an even length N, the sequences f_r and f_s of the half length are created as follows:

$$f_r(n') = f(n') + f(n' + N/2), \qquad f_s(n') = \left[f(n') - f(n' + N/2)\right] W_N^{-n'}. \tag{17}$$

Then, the Fourier transform F_r and F_s fulfill the following property: $F_r(k') = F(2k')$ and $F_s(k') = F(2k' + 1)$ for any $k' = 0, 1, \ldots, N/2 - 1$. Hence, the sequences f_r and f_s are then processed recursively, as shown in Fig. 3(b). It is easy to deduce the inverse equation from Eq. (17):

$$f(n') = \frac{1}{2}\left[f_r(n') + f_s(n')W_N^{n'}\right], \qquad f(n' + N/2) = \frac{1}{2}\left[f_r(n') - f_s(n')W_N^{n'}\right]. \tag{18}$$

5.4. The most popular FFT implementations

On CPU.

One of the most popular FFT implementations ever is so-called Fastest Fourier Transform in the West (FFTW) [61]. It is kept updated and available for download on the web page http://www.fftw.org/. According to the authors' comprehensive benchmark [62], it is still one of the fastest CPU implementations available. The top performance is achieved by using multiple CPU threads, the extended instruction sets of modern processors such as SSE/SSE2, optimized radix-N algorithms for N up to 7, optimized functions for purely real input data etc. Other popular CPU implementations can be found e.g. in the Intel libraries called Intel Integrated Performance Primitives (IPP) [63] and Intel Math Kernel Library (MKL) [64]. In terms of performance, they are comparable with the FFTW.

On other architectures.

For the graphics hardware, there exists several implementations in the literature [65–67]. Probably the most widely-used one is the CUFFT library by Nvidia. Although it is dedicated to the Nvidia graphics cards, it is popular due to its good performance and ease of use. It also contains optimized functions for real input data. The FFT has been also implemented on various architectures, including DSP [68] and FPGA [69].

5.5. Benefits and drawbacks of the fast convolution

To summarize this section, fast convolution is the most efficient approach if both signal and kernel contain thousands of samples or more, or if the kernel is slightly smaller but non-separable. Thanks to numerous implementations, it is accessible to a wide range of users on various architectures. The main drawbacks are the windowing problem, the relatively lower numerical precision, and considerable memory requirements due to the signal padding. In the following, we will examine the memory usage in detail and propose several approaches to optimize it on modern parallel architectures.

6. Decomposition in the time domain

In this section, we will focus on the decomposition of the fast convolution in the time domain. We will provide the analysis of time and space complexity. Regarding the former, we will focus on the number of additions and multiplications needed for the computation of studied algorithms.

Utilizing the convolution theorem and the fast Fourier transform the 1-D convolution of signal f and kernel g requires

$$(N^f + N^g)\left[\frac{9}{2}\log_2(N^f + N^g) + 1\right] \tag{19}$$

steps [8]. Here, the term $(N^f + N^g)$ means that the processed signal f was zero padded[2] to prevent the overlap effect caused by circular convolution. The kernel was modified in the same way. Another advantage of using Fourier transform stems from its separability. Convolving two 3-D signals f^{3d} and g^{3d}, where $||f||^{3d} = N_x^f \times N_y^f \times N_z^f$ and $||g^{3d}|| = N_x^g \times N_y^g \times N_z^g$, we need only

$$(N_x^f + N_x^g)(N_y^f + N_y^g)(N_z^f + N_z^g)\left[\frac{9}{2}\log_2\left((N_x^f + N_x^g)(N_y^f + N_y^g)(N_z^f + N_z^g)\right) + 1\right] \tag{20}$$

steps in total.

Up to now, this method seems to be optimal. Before we proceed, let us look into the space complexity of this approach. If we do not take into account buffers for the input/output signals and serialize both Fourier transforms, we need space for two equally aligned Fourier signals and some negligible Fourier transform workspace. In total, it is

$$(N^f + N^g) \cdot C \tag{21}$$

bytes, where $(N^f + N^g)$ is a size of one padded signal and C is a constant dependent on the required algorithm precision (single, double or long double). If the double precision is required, for example, then $\texttt{C} = 2 \cdot \texttt{sizeof}(\texttt{double})$, which corresponds to two Fourier signals used by real-valued FFT. In the 3-D case, when $||f^{3d}|| = N_x^f \times N_y^f \times N_z^f$ and $||g^{3d}|| = N_x^g \times N_y^g \times N_z^g$ the space needed by the aligned signal is proportionally higher: $(N_x^f + N_x^g)(N_y^f + N_y^g)(N_z^f + N_z^g) \cdot C$ bytes.

Keeping in mind that due to the lack of available memory, direct computation of fast convolution is not realizable using common computers we will try to split the whole task into several subtasks. This means that the input signal and kernel will be split into smaller pieces, so called *tiles* that need not be of the same size. Hence, we will try to reduce the memory requirements while keeping the efficiency of the whole convolution process as proposed in [23].

[2] The size of padded signal should be exactly $(N^f + N^g - 1)$. For the sake of simplicity, we reduced this term to $(N^f + N^g)$ as we suppose $N^f \gg 1$ and $N^g \gg 1$.

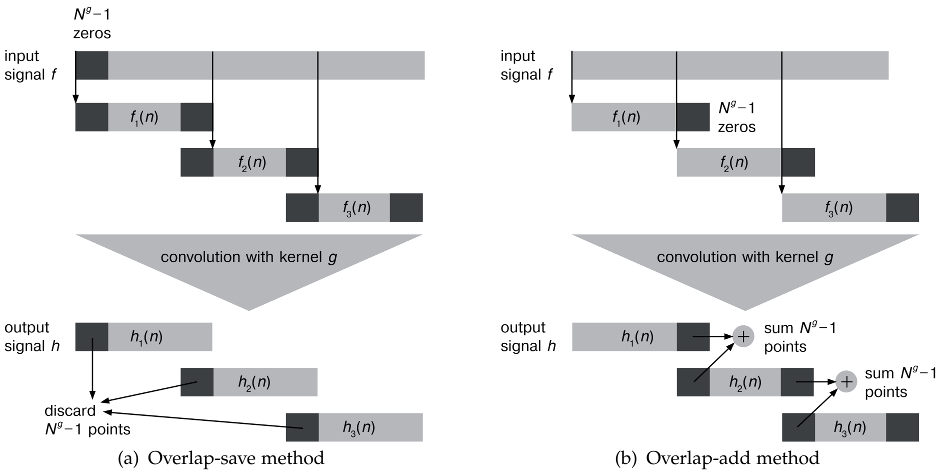


Figure 4. Using the overlap-save and overlap-add methods, the input data can be segmented into smaller blocks and convolved separately. Finally, the sub-parts are concatenated (a) or summed (b) together.

6.1. Signal tiling

Splitting the input signal f into smaller disjoint tiles $f_1, f_2, \dots, f_m$, then performing m smaller convolutions $f_i * g, i = 1, 2, \dots, m$ and finally concatenating the results together with discarding the overlaps is a well-known algorithm in digital signal processing. The implementation is commonly known as the *overlap-save method* [22].

6.1.0.5. Method.

Without loss of generality we will focus on the manipulation with just one *tile* f_i. The other tiles are processes in the same way. The tile f_i is uniquely determined by its size and shift with respect to the origin of f. Its size and shift also uniquely determine the area in the output signal h where the expected result of $f_i * g$ is going to be stored. In order to guarantee that the convolution $f_i * g$ computes correctly the appropriate part of output signal h, the tile f_i must be equipped with some overlap to its neighbours. The size of this overlap is equal to the size of the whole kernel g. Hence, the tile f_i is extended equally on both sides and we get f_i'. If the tile f_i is the boundary one, it is padded with zero values. As the fast convolution required both the signal and the kernel of the same size the kernel g must be also extended. It is just padded with zeros which produces g'. As soon as f_i' and g' are prepared, the convolution $f_i' * g'$ can be performed and the result is cropped to the size $||f_i||$. Then, all the convolutions $f_i' * g', i = 1, 2, \dots, m$ are successively performed and the output signal h is obtained by *concatenating* the individual results together. A general form of the method is shown in Fig. 4(a).

6.1.0.6. Analysis of time complexity.

Let us inspect the memory requirements for this approach. As the filtered signal f is split into m pieces, the respective memory requirements are lowered to

$$\left(\frac{N^f}{m} + N^g\right) \cdot C \tag{22}$$

bytes. Concerning the time complexity, after splitting the signal f into m tiles, we need to perform

$$(N^f + mN^g)\left[\frac{9}{2}\log_2\left(\frac{N^f}{m} + N^g\right) + 1\right] \tag{23}$$

multiplications in total. If there is no division ($m = 1$) we get the time complexity of the fast approach. If the division is total ($m = N^f$) we get even worse complexity than the basic convolution has. The higher the level of splitting is required the worse the complexity is. Therefore, we can conclude that splitting only the input signal into tiles does not help.

6.2. Kernel tiling

From the previous text, we recognize that splitting only the input signal f might be inefficient. It may even happen that the kernel g is so large that splitting of only the signal f does not reduce the memory requirements sufficiently. As the convolution belongs to commutative operators one could recommend swapping the input signal and the kernel. This may help, namely when the signal f is small and the kernel g is very large. As soon as the signal and the kernel are swapped, we can simply apply the overlap-save method. However, this approach fails when both the signal and the kernel are too large. Let us decompose the kernel g as well.

6.2.0.7. Method.

Keeping in mind that the input signal f has already been decomposed into m tiles using overlap-save method, we can focus on the manipulation with just one tile $f_i, i = 1, 2, \ldots, m$. For the computation of convolution of the selected tile f_i and the large kernel g we will employ so called *overlap-add method* [22]. This method splits the kernel g into n disjoint (nonoverlapping) pieces $g_j, j = 1, 2, \ldots, n$. Then, it performs n cheaper convolutions $f_i * g_j$, and finally it adds the results together preserving the appropriate overruns.

Without loss of generality we will focus on the manipulation with just one *kernel tile* g_j. Prior to the computation, the selected *tile* g_j has to be aligned to the size $||f_i|| + ||g_j||$. It is done simply by padding g_j with zeros equally on both sides. In this way, we get the tile g'_j. The signal tile f_i is also aligned to the size $||f_i|| + ||g_j||$. However, f'_i is not padded with zeros. It is created from f_i by extending its support equally on both sides.

Each kernel tile g_j has its positive shift s_j with respect to the origin of g. This shift is very important for further computation and cannot be omitted. Before we perform the convolution $f'_i * g'_j$ we must shift the tile f'_i within f by s_j samples to the left. The reason originates from the idea of kernel decomposition and minus sign in Eq. (2) which causes the whole kernel to be flipped. As soon as the convolution $f_i' * g_j'$ is performed, its result is cropped to the size $||f_i||$ and *added* to the output signal h into the position defined by overlap-save method. Finally, all the convolutions $f_i' * g_j', j = 1, 2, \ldots n$ are performed to get complete result for one given tile f_i. A general form of the method is shown in Fig. 4(b).

The complete computation of the convolution across all signal and kernel tiles is sketched in the Algorithm 1.

Algorithm 1. Divide-and-conquer approach applied to the convolution over large data.

```
$(f, g) \leftarrow$ (input signal, kernel)
$f \rightarrow f_1, f_2, \ldots, f_m$ {split 'f' into tiles according to overlap-save scheme}
$g \rightarrow g_1, g_2, \ldots, g_n$ {split 'g' into tiles according to overlap-add scheme}
$h \leftarrow 0$ {create the output signal 'h' and fill it with zeros}
for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
        $h_{ij} \leftarrow$ convolve$(f_i, g_j)$
            {use fast convolution}
        $h_{ij} \leftarrow$ discard_overruns$(h_{ij})$
            {discard $h_{ij}$ overruns following overlap-save output rules}
        $h \leftarrow h +$ shift$(h_{ij})$
            {add $h_{ij}$ to h following overlap-add output rules}
    end for
end for
Output $\leftarrow h$
```

6.2.0.8. Analysis of time complexity.

Let us suppose the signal f is split into m tiles and kernel g is decomposed into n tiles. The time complexity of the fast convolution $f_i * g_j$ is

$$\left(\frac{N^f}{m} + \frac{N^g}{n}\right)\left[\frac{9}{2}\log_2\left(\frac{N^f}{m} + \frac{N^g}{n}\right) + 1\right]. \tag{24}$$

We have m signal tiles and n kernel tiles. In order to perform the complete convolution $f * g$ we have to perform $m \times n$ convolutions (see the nested loops in Algorithm 1) of the individual signal and kernel tiles. In total, we have to complete

$$\left(nN^f + mN^g\right)\left[\frac{9}{2}\log_2\left(\frac{N^f}{m} + \frac{N^g}{n}\right) + 1\right] \tag{25}$$

steps. One can clearly see that without any division ($m = n = 1$) we get the complexity of fast convolution, i.e. the class $O((N^f + N^g)\log(N^f + N^g))$. For total division ($m = N^f$ and

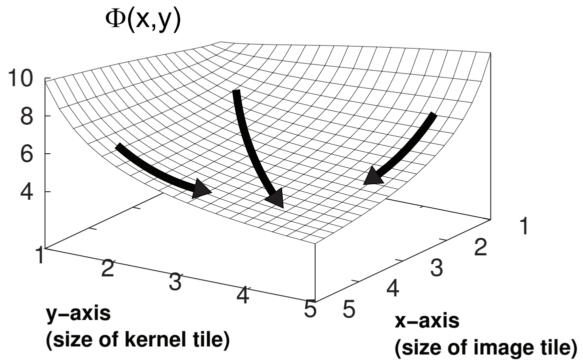


Figure 5. A graph of a function $\Phi(x, y)$ that represents the time complexity of tiled convolution. The x-axis and y-axis correspond to number of samples in signal and kernel tile, respectively. The evident minimum of function $\Phi(x, y)$ occurs in the location, where both variables (sizes of tiles) are maximized and equal at the same time.

$n = N^g$) we obtain basic convolution, i.e. the complexity class $O(N^f N^g)$. Concerning the space occupied by our convolution algorithm, we need

$$\left(\frac{N^f}{m} + \frac{N^g}{n}\right) \cdot C \tag{26}$$

bytes, where C is again the precision dependent constant and m, n are the levels of division of signal f and kernel g, respectively.

6.2.0.9. Algorithm optimality.

We currently designed an algorithm of splitting the signal f into m tiles and the kernel g into n tiles. Now we will answer the question regarding the optimal way of splitting the input signal and the kernel. As the relationship between m and n is hard to be expressed and N^f and N^g are constants let us define the following substitution: $x = \frac{N^f}{m}$ and $y = \frac{N^g}{n}$. Here x and y stand for the sizes of the signal and the kernel tiles, respectively. Applying this substitution to Eq. (25) and simplifying, we get the function

$$\Phi(x, y) = N^f N^g \left(\frac{1}{x} + \frac{1}{y}\right) \left[\frac{9}{2} \log_2(x + y) + 1\right] \tag{27}$$

The plot of this function is depicted in Figure 5. The minimum of this function is reached if and only if $x = y$ and both variables x and y are maximized, i.e. the input signal and the kernel tiles should be of the same size (equal number of samples) and they should be as large as possible. In order to reach the optimal solution, the size of the tile should be the power of small primes [70]. In this sense, it is recommended to fulfill both criteria put on the tile size: the maximality (as stated above) and the capability of simple decomposition into small primes.

6.3. Extension to higher dimensions

All the previous statements are related only to a 1-D signal. Provided both signal and kernel are 3-dimensional and the tiling proces identical in all the axes, we can combine Eq. (20) and Eq. (25) in order to get:

$$\left(nN_x^f+mN_x^g\right)\left(nN_y^f+mN_y^g\right)\left(nN_z^f+mN_z^g\right)\left[\tfrac{9}{2}\log_2\left(\tfrac{N_x^f}{m}+\tfrac{N_x^g}{n}\right)\left(\tfrac{N_y^f}{m}+\tfrac{N_y^g}{n}\right)\left(\tfrac{N_z^f}{m}+\tfrac{N_z^g}{n}\right)+1\right] \tag{28}$$

This statement can be further generalized for higher dimensions or for irregular tiling process. The proof can be simply derived from the separability of multidimensional Fourier transform, which guarantees that the time complexity of the higher dimensional Fourier transform depends on the amount of processed samples only. There is no difference in the time complexity if the higher-dimensional signal is elongated or in the shape of cube.

6.4. Parallelization

6.4.0.10. On multicore CPU.

As the majority of recent computers are equipped with multi-core CPUs the following text will be devoted to the idea of parallelization of our approach using this architecture. Each such computer is equipped with two or more cores, however both cores share one memory. This means that execution of two or more huge convolutions concurrently may simply fail due to lack of available memory. The possible workaround is to perform one more division, i.e. signal and kernel tiles will be further split into even smaller pieces. Let p be a number that defines how many sub-pieces the signal and the kernel tiles should be split into. Let P be a number of available processors. If we execute the individual convolutions in parallel we get the overall number of multiplications

$$\frac{npN^f+mpN^g}{P}\left[\frac{9}{2}\log\left(\frac{N^f}{mp}+\frac{N^g}{np}\right)+1\right] \tag{29}$$

and the space requirements

$$\left(\frac{N^f}{mp}+\frac{N^g}{np}\right)\cdot C\cdot P \tag{30}$$

Let us study the relationship p versus P:

- $p < P \ldots$ The space complexity becomes worse than in the original non-parallelized version (26). Hence, there is no advantage of using this approach.
- $p > P \ldots$ There are no additional memory requirements. However, the signal and kernel are split into too small pieces. We have to handle large number of overlaps of tiles which will cause the time complexity (29) to become worse than in the non-parallelized case (25).

- $p = P \ldots$ The space complexity is the same as in the original approach. The time complexity is slightly better but practically it brings no advantage due to lots of memory accesses. The efficiency of this approach would be brought to evidence only if $P \gg 1$. As the standard multi-core processors are typically equipped with only 2, 4 or 8 cores, neither this approach was found to be very useful.

6.4.0.11. On computer clusters.

Regarding computer clusters the problem with one shared memory is solved as each computer has its private memory. Therefore, the total number of multiplications (see Eq. (25)) is modified by factor $\frac{B}{P}$, where P is the number of available computers and B is a constant representing the overheads and the cost of data transmission among the individual computers. The computation becomes effective only if $P > B$. The memory requirements for each node remain the same as in the non-parallelized case as each computer takes care of its own private memory space.

7. Decomposition in the frequency domain

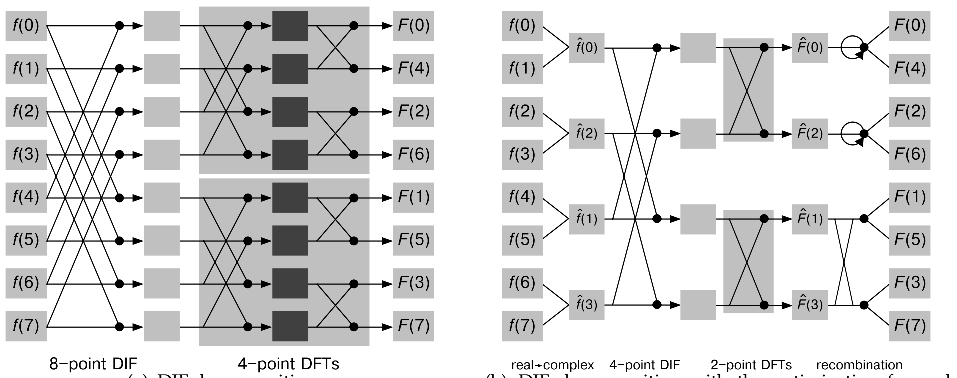
Just as the concept of the decomposition in the spatial (time) domain, the decomposition in the frequency domain can be used for the fast convolution algorithm, in order to (i) decrease the required amount of memory available per processing unit, (ii) employ multiple processing units without need of extensive data transfers between the processors. In the following text, we introduce the concept of the decomposition [21] along with optimization steps suitable for purely real data [71]. Subsequently, we present the results on achieved on a current graphics hardware. Finally, we conclude the applications and architectures where the approach can be used.

7.1. Decomposition using the DIF algorithm

In Section 5.3, the decimation-in-frequency algorithm was recalled. The DIF can be used not only to compute FFT itself but also to decompose the fast convolution. This algorithm can be divided into several phases, namely (i) so-called *decomposition* into parts using Eq. (17), (ii) the Fourier transforms of the parts, (iii) the convolution by pointwise multiplication itself, (iv) the inverse Fourier transforms, and (v) so-called *composition* using Eq. (18). In the following paragraph, we provide the mathematical background for the individual phases. The scheme description of the algorithm is shown in Fig. 6(a).

By employing Eq. (17), both the input signal f and g can be divided into sub-parts f_r, f_s and g_r, g_s, respectively. As the Fourier transforms F_r and F_s satisfy $F_r(k') = F(2k')$ and $F_s(k') = F(2k' + 1)$ and the equivalent property is held for G_r and G_s, by applying FFT on F_r F_s, G_r, and G_s, individually, we obtain two separate parts of both the signal and the kernel. Subsequently, by computing the point-wise multiplication $H_r = F_r G_r$ and $H_s = F_s G_s$, respectively, we obtain two separate parts of the Fourier transform of the convolution $h = f * g$. Finally, the result h is obtained by applying Eq. (18) to the inverse Fourier transforms h_r and h_s.

In the first and the last phase, it is inevitable to store the whole input signals in the memory. Here, the memory requirements are equal to those in the classical fast convolution algorithm. However, in the phases (ii)–(iv) which are by far the most computationally extensive, the



(a) DIF decomposition

(b) DIF decomposition with the optimization for real data

Figure 6. A scheme description of the convolution algorithm with the decomposition in the frequency domain [71]. An input signal is decomposed into 2 parts by the decimation in frequency (DIF) algorithms. The parts are subsequently processed independently using the discrete Fourier transform (DFT).

memory requirements are inversely proportional to the number of parts d the signals are divided into. The algorithm is hence suitable for architectures with the star topology where the central node is relatively slow but has large memory, and the end nodes are fast but have small memory. The powerful desktop PC with one or several GPU cards is a typical example of such architecture.

It can be noted that the decimation-in-time (DIT) algorithm can also be used for the purpose of decomposing the convolution problem. However, its properties make it sub-efficient for practical use. Firstly, its time complexity is comparable with the one of DIF. Secondly and most important, it requires significantly more data transfers between the central and end nodes. In Section 7.5, the complexity of the individual algorithms is analysed in detail.

7.2. Optimization for purely real signals

In most practical applications, users work with purely real input signals. As described in Section 5.1, the Fourier transform is complex but satisfies specific properties when applied on such data. Therefore, it is reasonable to optimize the fast convolution algorithm in order to reduce both the time and the memory complexity. In the following paragraphs, we will describe three fundamental approaches to optimize the fast convolution of real signals.

Real-to-complex FFT.

As described in Section 5.4, most popular FFT implementations offer specialized functions for the FFT of purely real input data. With the classical fast convolution, users are advised to use specific functions of their preferred FFT library. With the DIF decomposition, it is nevertheless no more possible to use such functions as the decomposed signals are no more real.

Combination of signal and kernel.

It is possible to combine the two real input signals $f(n)$ and $g(n)$, $n = 0, 1, \ldots, N-1$, into one complex signal $f(n) + jg(n)$ of the same length. However, this operation requires an additional buffer of length at least N. This poses significantly higher demands on the memory available at the central node.

"Complexification" of input signals.

Provided that the length N of a real input signal f is even, we can introduce a complex signal $\hat{f}(n') \equiv f(2n') + jf(2n'+1)$ for any $n' = 0, 1, \ldots, N/2-1$. As the most common way of storing the complex signals is to store real and complex components, alternately, a real signal can be turned into a complex one by simply over-casting the data type, avoiding any computations or data transfers. The relationship between the Fourier transforms F and $\hat{F}$ is given by following:

$$F(k') = \frac{1}{2}\left(\alpha_+(k') - jW_N^{k'}\alpha_-(k')\right), \qquad F(k'+N/2) = \frac{1}{2}\left(\alpha_+(k') + jW_N^{k'}\alpha_-(k')\right), \tag{31}$$

where

$$\alpha_\pm(k') \equiv \hat{F}(k') \pm \hat{F}^*(N/2 - k'). \tag{32}$$

As the third approach yields the best performance, it is used in the final version of the algorithm. The computation of Eq. (31), (32) will be further referred to as the *recombination* phase. The scheme description of the algorithm is shown in Fig. 6(b).

7.3. Getting further

The algorithm can be used not only in 1D but generally for any n-dimensional input signals. To achieve maximum data transfer efficiency, it is advisable to perform the decomposition in the first (y in 2D or z in 3D) axis so that the individual sub-parts form the undivided memory blocks, as explained in [21].

Furthermore, the input data can be decomposed into generally d parts using an appropriate radix-d algorithm in both the decomposition and the composition phase. It should be noted, however, that due to the recombination phase, the algorithm requires twice more memory space per end node for $d > 2$. This is due to fact that some of the parts need to be recombined with others—refer to Fig. 6(b). To be more precise, the memory requirements are $2(N^f + N^g)/d$ for $d = 2$ and $4(N^f + N^g)/d$ for $d > 2$.

7.4. GPU and multi-GPU implementation

As Nvidia provides users with the CUFFT library [32] for the efficient computation of the fast Fourier transform, the GPU implementation of the aforementioned algorithm is quite straightforward. The scheme description of the implementation is shown in Fig. 7. It should be noted that the significant part of the computation time is spent for the data transfers between the computing nodes (CPU and GPU, in this case). The algorithm is designed to keep the number of data transfers as low as possible. Nevertheless, it is highly

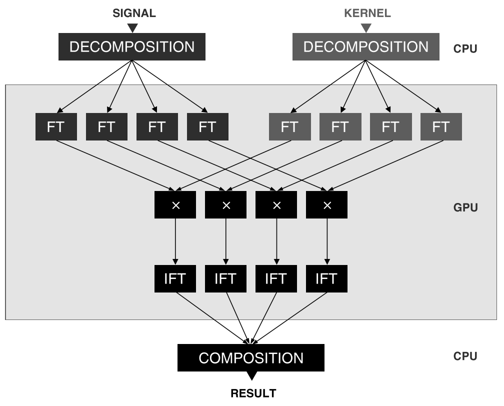


Figure 7. A scheme description of the proposed algorithm for the convolution with the decomposition in the frequency domain, implemented on GPU [21]. The example shows the decomposition into 4 parts.

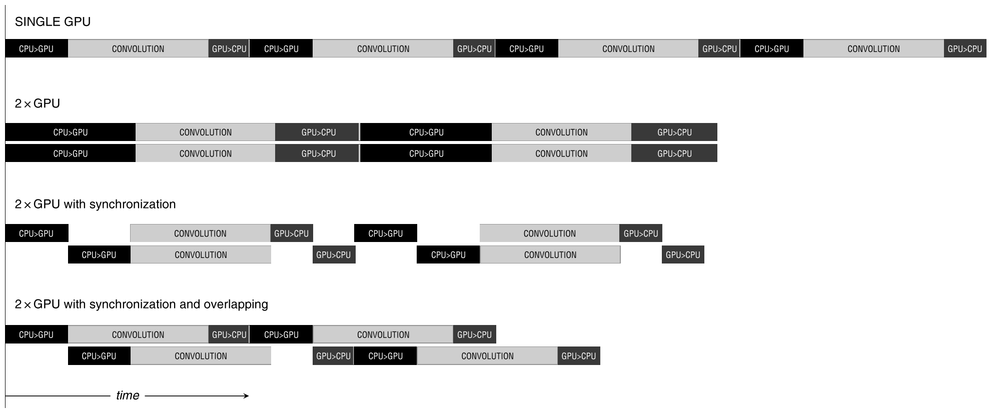

SINGLE GPU
CPU>GPU
CONVOLUTION
GPU>CPU
2 × GPU
2 × GPU with synchronization
2 × GPU with synchronization and overlapping
time

Figure 8. A model timeline of the algorithm workflow [21]. The dark boxes denote data transfers between CPU and GPU while the light boxes represent convolution computations. The first row shows the single-GPU implementation. The second row depicts parallel usage of two GPUs. The data transfers are performed concurrently but through a common bus, therefore they last twice longer. For the third row, the data transfers are synchronized so that only one transfer is made at a time. In the last row, the data transfers are overlapped with the convolution execution.

recommendable to overlap the data transfers with some computation phases in order to keep the implementation as efficient as possible.

To prove the importance of the overlapping, we provide a detailed analysis of the algorithm workflow. The overall computation time T required by the algorithm can be expressed as follows:

$$T = \max(t_p + t_d, t_a) + t_{h\to d} + \frac{t_{\text{conv}}}{P} + t_{d\to h} + t_c, \tag{33}$$

where t_p is the time required for the initial signal padding, t_d for decomposition, t_a for allocating memory and setting up FFT plans on GPU, $t_{h\to d}$ for data transfers from CPU to GPU (host to device), t_{conv} for the convolution including the FFT, recombination phase, point-wise multiplication, and the inverse FFT, $t_{d \to h}$ for data transfers from GPU to CPU (device to host) and finally t_c for composition. The number of end nodes (GPU cards) is denoted by P. It is evident that in accordance with the famous Amdahl's law [72], the speed-up achieved by multiple end nodes is limited to the only parallel phase of the algorithm which is the convolution itself. Now if the data are decomposed into d parts and sent to P end units and if $d > P > 1$, the data transfers can be overlapped with the convolution phase. This means that the real computation time is shorter than T as in Eq. (33). Eq. (33) can be hence viewed as the upper limit. The model example is shown in Fig. 8.

7.5. Algorithm comparison

In the previous text, we mentioned three approaches for the decomposition of the fast convolution: Tiling (decomposition in the time domain), the DIF-based, and the DIT-based algorithm. For fair comparison of the three, we compute the number of arithmetic operations, the number of data transfers, and the memory requirements per end node, with respect to the input signal length and the d parameter, i.e. the number of parts the data are divided into. As for the tiling method, the computation is based on Eq. (27) while setting $d = m = n$ (the optimum case). The results are shown in Table 1.

Method	# of operations	# of data transfers	Memory required per node
DIF	$(N^f + N^g)\left[\frac{9}{2}\log_2(N^f + N^g) + 1\right]$	$3(N^f + N^g)$	$4(N^f + N^g)/d$
DIT	$(N^f + N^g)\left[\frac{9}{2}\log_2(N^f + N^g) + 2\right]$	$(d+1)(N^f + N^g)$	$4(N^f + N^g)/d$
Tiling	$d(N^f + N^g)\left[\frac{9}{2}\log_2(\frac{N^f + N^g}{d}) + 1\right]$	$(d+1)(N^f + N^g)$	$(N^f + N^g)/d$

Table 1. Methods for decomposition of the fast convolution and their requirements

To conclude the results, it can be noted that the tiling method is the best one in terms of memory demands. It requires 4× less memory per end node than the DIF-based and the DIT-based algorithms. On the other hand, both the number of the operations and the number of data transfers are dependent on the d parameter which is not the case of the DIF-based method. By dividing the data into more sub-parts, the memory requirements of the DIF-based algorithm decrease while the number of operations and memory transactions remain constant. Hence, the DIF-based algorithm can be generally more efficient than the tiling.

7.6. Applications and architectures

Both the tiling and the DIF-based algorithm can be used to allow the computation of the fast convolution in the applications where the convolving signals are multi-dimensional and/or contain too many samples to be handled efficiently on a single computer. We already mentioned the application of the optical microscopy data where the convolution is used to simulate the image degradation introduced by an optical system. Using the decomposition methods, the computation can be distributed over (a) a computer grid, (b) multiple CPU and

GPU units where CPU is usually provided with more memory, hence it is used as a central node for the (de)composition of the data.

8. Conclusions

In this text, we introduce the convolution as an important tool in both signal and image processing. In the first part, we mention some of the most popular applications it is employed in and recall its mathematical definition. Subsequently, we present a number of common algorithms for an efficient computation of the convolution on various architectures. The simplest approach—so-called naïve convolution—is to perform the convolution straightly using the definition. Although it is less efficient than other algorithms, it is the most general one and is popular in some specific applications where small convolution kernels are used, such as edge or object detection. If the convolution kernel is multi-dimensional and can be expressed as a convolution of several 1-D kernels, then the naïve convolution is usually replaced by its alternative, so-called separable convolution. The lowest time complexity can be achieved by using the recursive filtering. Here, the result of the convolution at each position can be obtained by applying a few arithmetical operations to the previous result. Besides the efficiency, the advantage is that these filters are suitable for streaming architectures such as FPGA. On the other hand, this method is generally not suitable for all convolution kernels as the recursive filters are often numerically unstable and inaccurate. The last algorithm present in the chapter is the fast convolution. According to the so-called convolution theorem, the convolution can be computed in the frequency domain by a simple point-wise multiplication of the Fourier transforms of the input signals. This approach is the most suitable for long signals and kernels as it yields generally the best time complexity. However, it has non-trivial memory demands caused by the fact that the input data need to be padded.

Therefore, in the second part of the chapter, we describe two approaches to reduce the memory requirements of the fast convolution. The first one, so-called tiling is performed in the spatial (time) domain. It is the most efficient with respect to the memory requirements. However, with a higher number of sub-parts the input data are divided into, both the number of arithmetical operations and the number of potential data transfers increase. Hence, in some applications or on some architectures (such as the desktop PC with one ore multiple graphics cards) where the overhead of data transfers is critical, one can use a different approach, based on the decomposition-in-frequency (DIF) algorithm which is widely known from the concept of the fast Fourier transform. We also mention the third method based on the decomposition-in-time (DIT) algorithm. However, the DIT-based algorithm is sub-efficient from every point of view so there is no reason for it to be used instead of the DIF-based one. In the end of the chapter, we also provide a detailed analysis of (i) the number of arithmetical operations, (ii) the number of data transfers, (iii) the memory requirements for each of the three methods.

As the convolution is one of the most extensively-studied operations in the signal processing, the list of the algorithms and implementations mentioned in this chapter is not and cannot be complete. Nevertheless, we tried to include those that we consider to be the most popular and widely-used. We also believe that the decomposition tricks which are described in the second part of the chapter and are the subject of the authors' original research can help readers to improve their own applications, regardless of target architecture.

Acknowledgments

This work has been supported by the Grant Agency of the Czech Republic (Grant No. P302/12/G157).

Author details

Pavel Karas* and David Svoboda

* Address all correspondence to: xkaras1@fi.muni.cz

Centre for Biomedical Image Analysis, Faculty of Informatics, Masaryk University, Brno, Czech Republic

References

[1] J. Jan. *Digital Signal Filtering, Analysis and Restoration (Telecommunications Series)*. INSPEC, Inc., 2000.

[2] S. W. Smith. *Digital Signal Processing*. Newnes, 2003.

[3] A. Foi. Noise estimation and removal in MR imaging: The variance stabilization approach. In *IEEE International Symposium on Biomedical Imaging: from Nano to Macro*, pages 1809—1814, 2011.

[4] J. R. Parker. *Algorithms for Image Processing and Computer Vision*. Wiley Publishing, 2nd edition, 2010.

[5] J. Canny. A computational approach to edge detection. *IEEE T-PAMI*, 8:769–698, 1986.

[6] D. H. Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[7] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, 2007.

[8] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Prentice Hall, 2002. ISBN: 0-201-18075-8.

[9] K. R. Castleman. *Digital Image Processing*. Prentice Hall, 1996.

[10] P. J. Verveer. Computational and optical methods for improving resolution and signal quality in fluorescence microscopy. 1998. PhD Thesis.

[11] A. Lehmussola, J. Selinummi, P. Ruusuvuori, A. Niemistö, and O. Yli-Harja. Simulating fluorescent microscope images of cell populations. In *Proceedings of the 27th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC'05)*, pages 3153–3156, 2005.

[12] D. Svoboda, M. Kozubek, and S. Stejskal. Generation of Digital Phantoms of Cell Nuclei and Simulation of Image Formation in 3D Image Cytometry. *Cytometry part A*, 75A(6):494–509, JUN 2009.

[13] W. K. Pratt. *Digital Image Processing*. Wiley, 3rd edition edition, 2001.

[14] T. Bräunl. *Parallel Image Processing*. Springer, 2001.

[15] H.-M. Yip, I. Ahmad, and T.-C. Pong. An Efficient Parallel Algorithm for Computing the Gaussian Convolution of Multi-dimensional Image Data. *The Journal of Supercomputing*, 14(3):233–255, 1999. ISSN: 0920-8542.

[16] O. Schwarzkopf. Computing Convolutions on Mesh-Like Structures. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 695–699, 1993.

[17] S. Kadam. Parallelization of Low-Level Computer Vision Algorithms on Clusters. In *AMS '08: Proceedings of the 2008 Second Asia International Conference on Modelling & Simulation (AMS)*, pages 113–118, Washington, DC, USA, 2008. IEEE Computer Society. ISBN: 978-0-7695-3136-6.

[18] B. Jähne. *Digital Image Processing*. Springer, 5th edition edition, 2002.

[19] Robert Hummel and David Loew. Computing Large-Kernel Convolutions of Images. Technical report, New York University, Courant Institute of Mathematical Sciences, 1986.

[20] R. N. Bracewell. *Fourier Analysis and Imaging*. Springer, 2006.

[21] P. Karas and D. Svoboda. Convolution of large 3D images on GPU and its decomposition. *EURASIP Journal on Advances in Signal Processing*, 2011(1):120, 2011.

[22] A.V. Oppenheim, R.W. Schafer, J.R. Buck, et al. *Discrete-time signal processing*, volume 2. Prentice hall Upper Saddle Riverˆ eN. JNJ, 1989.

[23] D. Svoboda. Efficient computation of convolution of huge images. *Image Analysis and Processing–ICIAP 2011*, pages 453–462, 2011.

[24] R. G. Shoup. Parameterized convolution filtering in an FPGA. In *Selected papers from the Oxford 1993 international workshop on field programmable logic and applications on More FPGAs*, pages 274–280, Oxford, UK, UK, 1994. Abingdon EE&CS Books.

[25] A. Benedetti, A. Prati, and N. Scarabottolo. Image convolution on FPGAs: the implementation of a multi-FPGA FIFO structure. In *Euromicro Conference, 1998. Proceedings. 24th*, volume 1, pages 123–130 vol.1, Aug 1998.

[26] S. Perri, M. Lanuzza, P. Corsonello, and G. Cocorullo. A high-performance fully reconfigurable FPGA-based 2D convolution processor. *Microprocessors and Microsystems*, 29(8—9):381–391, 2005. Special Issue on FPGAs: Case Studies in Computer Vision and Image Processing.

[27] A. Herout, P. Zemcik, M. Hradis, R. Juranek, J. Havel, R. Josth, and L. Polok. *Low-Level Image Features for Real-Time Object Detection*. InTech, 2010.

[28] H. Shan and N. A. Hazanchuk. Adaptive Edge Detection for Real-Time Video Processing using FPGAs. Application notes, Altera Corporation, 2005.

[29] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. pages 21–51, August 2005.

[30] D. Castańo-Díez, D. Moser, A. Schoenegger, S. Pruggnaller, and A. S. Frangakis. Performance evaluation of image processing algorithms on the GPU. *Journal of Structural Biology*, 164(1):153–160, 2008.

[31] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[32] NVIDIA Developer Zone. http://developer.nvidia.com/category/zone/cuda-zone, Apr 2012.

[33] Khronos Group. OpenCL. http://www.khronos.org/opencl/, 2011.

[34] CUDA Downloads. http://developer.nvidia.com/cuda-downloads, Apr 2012.

[35] V. Podlozhnyuk. Image Convolution with CUDA. http://developer.download.nvidia.com/assets/cuda/files/convolutionSeparable.pdf, Jun 2007.

[36] NVIDIA Performance Primitives. http://developer.nvidia.com/npp, Feb 2012.

[37] Y. Luo and R. Duraiswami. Canny edge detection on NVIDIA CUDA. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, Jun 2008.

[38] K. Ogawa, Y. Ito, and K. Nakano. Efficient Canny Edge Detection Using a GPU. In *Networking and Computing (ICNC), 2010 First International Conference on*, pages 279–280, Nov 2010.

[39] A. Herout, R. Jošth, R. Juránek, J. Havel, M. Hradiš, and P. Zemčík. Real-time object detection on CUDA. *Journal of Real-Time Image Processing*, 6:159–170, 2011. 10.1007/s11554-010-0179-0.

[40] Ke Zhang, Jiangbo Lu, G. Lafruit, R. Lauwereins, and L. Van Gool. Real-time accurate stereo with bitwise fast voting on CUDA. In *IEEE 12th International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 794 –800, Oct 2009.

[41] Wei Chen, M. Beister, Y. Kyriakou, and M. Kachelries. High performance median filtering using commodity graphics hardware. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4142–4147, Nov 2009.

[42] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of parallel and distributed computing*, 68(10):1370–1380, 2008.

[43] Zhaoyi Wei, Dah-Jye Lee, B. E. Nelson, J. K. Archibald, and B. B. Edwards. FPGA-Based Embedded Motion Estimation Sensor. 2008.

[44] XinXin Wang and B.E. Shi. GPU implemention of fast Gabor filters. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 373–376, Jun 2010.

[45] O. Fialka and M. Čadík. FFT and Convolution Performance in Image Filtering on GPU. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, pages 609–614, 2006.

[46] J. S. Jin and Y. Gao. Recursive implementation of LoG filtering. *Real-Time Imaging*, 3(1):59–65, February 1997.

[47] R. Deriche. Using Canny's criteria to derive a recursively implemented optimal edge detector. *The International Journal of Computer Vision*, 1(2):167–187, May 1987.

[48] I. T. Young and L. J. van Vliet. Recursive implementation of the Gaussian filter. *Signal Processing*, 44(2):139–151, 1995.

[49] F.G. Lorca, L. Kessal, and D. Demigny. Efficient ASIC and FPGA implementations of IIR filters for real time edge detection. In *Image Processing, 1997. Proceedings., International Conference on*, volume 2, pages 406–409 vol.2, Oct 1997.

[50] R.D. Turney, A.M. Reza, and J.G.R. Delva. FPGA implementation of adaptive temporal Kalman filter for real time video filtering. In *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, volume 4, pages 2231–2234 vol.4, Mar 1999.

[51] J. Diaz, E. Ros, F. Pelayo, E.M. Ortigosa, and S. Mota. FPGA-based real-time optical-flow system. *Circuits and Systems for Video Technology, IEEE Transactions on*, 16(2):274–279, Feb 2006.

[52] J. Robelly, G. Cichon, H. Seidel, and G. Fettweis. Implementation of recursive digital filters into vector SIMD DSP architectures. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 5, pages V – 165–8 vol.5, may 2004.

[53] F. Trebien and M. Oliveira. Realistic real-time sound re-synthesis and processing for interactive virtual worlds. *The Visual Computer*, 25:469–477, 2009. 10.1007/s00371-009-0341-5.

[54] E.O. Brigham and R.E. Morrow. The fast Fourier transform. *Spectrum, IEEE*, 4(12):63–70, 1967.

[55] H.J. Nussbaumer. Fast Fourier transform and convolution algorithms. *Berlin and New York, Springer-Verlag(Springer Series in Information Sciences.*, 2, 1982.

[56] Donald Fraser. Array Permutation by Index-Digit Permutation. *J. ACM*, 23(2):298–309, April 1976.

[57] G.U. Ramos. Roundoff error analysis of the fast Fourier transform. *Math. Comp*, 25:757–768, 1971.

[58] R. N. Bracewell. *The Fourier Transform and Its Applications*. McGraw-Hill, 3rd edition, 2000.

[59] F.J. Harris. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE*, 66(1):51–83, 1978.

[60] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput*, 19(90):297–301, 1965.

[61] M. Frigo and S.G. Johnson. The Fastest Fourier Transform in the West. 1997.

[62] M. Frigo and S.G. Johnson. benchFFT. http://www.fftw.org/benchfft/, 2012.

[63] Intel Integrated Performance Primitives. http://software.intel.com/en-us/articles/intel-ipp/, 2012.

[64] Intel Integrated Performance Primitives. http://software.intel.com/en-us/articles/intel-mkl/, 2012.

[65] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[66] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[67] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki. The OpenCL Programming Book. *Group*, 2009.

[68] Z. Li, H. Sorensen, and C. Burrus. FFT and convolution algorithms on DSP microprocessors. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'86.*, volume 11, pages 289–292. IEEE, 1986.

[69] I.S. Uzun, A. Amira, and A. Bouridane. FPGA implementations of fast Fourier transforms for real-time signal and image processing. In *Vision, Image and Signal Processing, IEE Proceedings-*, volume 152, pages 283–296. IET, 2005.

[70] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast Fourier transform. *ASSP Magazine, IEEE*, 1(4):14–21, Oct 1984. ISSN: 0740-7467.

[71] P. Karas, D. Svoboda, and P. Zemčík. GPU Optimization of Convolution for Large 3-D Real Images. In *Advanced Concepts for Intelligent Vision Systems (ACIVS), 2012*. Springer, 2012. Accepted.

[72] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.