SN Computer Science (2022) 3:78
https://doi.org/10.1007/s42979-021-00960-7

ORIGINAL RESEARCH

GPU-CUDA Implementation of the Third Order Gaussian Recursive Filter

Pasquale De Luca[1,2] · Ardelio Galletti[2] · Livia Marcellino[2]

Received: 3 December 2020 / Accepted: 6 September 2021 / Published online: 19 November 2021

Abstract
Gaussian convolution operation is a fundamental procedure in several data analysis tasks and scientific fields. For example, Gaussian convolution is a central step in data assimilation and machine learning and it is also frequently used in image and signal processing. Gaussian recursive filters are a class of methods designed to approximate Gaussian convolutions in a fast way. In De Luca et al. (2019 15th international conference on signal-image technology and internet-based systems (SITIS), pp 941–648, 2019), we presented a parallel implementation of the K-iterated first-order Gaussian recursive filter. This algorithm has been proved to be very efficient and accurate. Here, we provide a new GPU-parallel implementation which is based on the third order recursive filter. This filter guarantees larger accuracy and a lower computational cost with respect to the first-order one. To manage an efficient memory strategy access, and to achieve better performance results, our algorithm exploits the CUDA capabilities available on the GPU environment. Results in terms of performance and accuracy are provided in tests and experiments.

Keywords Gaussian convolution · Recursive filter · GPU · CUDA

Introduction

It is well-known that convolution operations plays a significant role in the computational process of most big-data analysis problems [13]. In particular, the Gaussian convolution, that is the application of the Gaussian filter, can be considered needful for many procedures, for this reason it represents a pre-processing that rarely can be avoided. Some application fields are, for example, in Data Assimilation and Machine Learning, for solving three-dimensional variational analysis schemes [5] and in advanced image and signal processing [1, 10]. Significant efforts have been made to speedup the operation of Gaussian filter, since for large datasets it requires a considerable computation complexity. In fact, whatever the field of application, such a basic preprocessing operation involves for large input sizes [7, 8, 11, 12] too many operations and memory accesses. This is unacceptable, for a basic step of data analysis software [15, 17, 19]. Indeed, faster methods, parallel approaches and High Performance Computing (HPC) architectures, as multicore or Graphics Processing Units (GPUs), are strongly helpful for this kind of problem and many parallel implementations have been presented to this purpose (see survey in [2, 3, 6]).

More in details, if we denote by $s^{(0)}$ an input signal, that is a complex valued function:

$$s^{(0)} = \left\{s_j^{(0)}\right\}_{j\in\mathbf{Z}}, \qquad \text{with} \quad s_j^{(0)} \in \mathbf{C}, \tag{1}$$

the discrete Gaussian convolution $s^{(g)}$ of $s^{(0)}$ has entries defined as:

$$s_j^{(g)} = \left(g * s^{(0)}\right)_j = \sum_{t\in\mathbf{Z}} g_{j-t} s_t^{(0)}, \qquad \forall j \in \mathbf{Z}, \tag{2}$$

This article is part of the topical collection "Advances on Signal Image Technology and Internet-based Systems" guest edited by Albert Dipanda, Luigi Gallo and Kokou Yetongnon.

✉ Ardelio Galletti
ardelio.galletti@uniparthenope.it

Pasquale De Luca
deluca@ieee.org

Livia Marcellino
livia.marcellino@uniparthenope.it

1 Department of Computer Science, University of Salerno, via Giovanni Paolo II, Fisciano, Italy

2 Department of Science and Technology, University of Naples "Parthenope", Centro Direzionale C4, Naples, Italy

where

$$g_t \equiv g(t) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{t^2}{2\sigma^2}\right), \tag{3}$$

and g represents the Gaussian kernel with zero mean and standard deviation $\sigma > 0$. The recursive filters (RFs) have widely used as computational tool to provide an efficient and fast approximation of Gaussian convolution. The basic idea of RFs consists in defining recursive formulas involving subsequent entries of the output in a recursive way. This results in a series of linear equations to be solved to obtain the required approximation of the convolution output. A general description of RFs is the following. Given an input signal as in (1), a K-iterated n-order Gaussian RF filter computes the output $s^{(K)}$, i.e. the K-iterate approximation of $s^{(g)}$, whose entries solve the infinite sequences of equations:

$$p_j^{(k)} = \beta s_j^{(k-1)} + \sum_{t=1}^{n} \alpha_t p_{j-t}^{(k)}, \qquad \forall j \in \mathbf{Z}, \tag{4}$$

$$s_j^{(k)} = \beta p_j^{(k)} + \sum_{t=1}^{n} \alpha_t s_{j+t}^{(k)}, \qquad \forall j \in \mathbf{Z}. \tag{5}$$

K represents the number of filter iterations, while k denotes the filter iteration counter. Different RFs have been proposed in literature, by setting the smoothing coefficients α_t and β (see [4] and references therein). In particular, starting from their formulation by [18, 20], RFs have been exploited for solving three-dimensional variational analysis schemes [5], as well as, in signal processing. The order of the filter characterises the accuracy that is offered in approximating the convolution. The repeated application of a n-order filter, that is the application of a K-iterated filter, can improve that accuracy. However, when these filters are applied to signals with support in a finite domain, they continue to generate distortions and artifacts, mostly localised at the boundaries of the output. This issue is addressed, partially, by introducing the so-called boundary conditions [18] and a suitable extending-resizing strategy.

In [9], we presented an accelerated implementation, on GPU environment, of the K-iterated first-order Gaussian RF for 1D signals. The implementation is performed together with a suitable memory strategy which involves the strong usage GPU memory. More specifically, to reduce the memory access time, the redundant operations have been addressed in GPU L2 cache memory. This implementation has been proved to be very efficient and quite accurate, depending on the number of filter iterations K. Greater efficiency, given the same accuracy level, can be reached by increasing the order of the basic RF. To this aim, in this work, we use the third order Gaussian RF. The choice to involve this filter is strongly based on the will to achieve a faster and more accurate Gaussian convolution approximation. In particular, we first give a description about the third order Gaussian RF, then, in addiction, we propose an accelerated implementation of it in a GPU environment to retrieve good results in a lower time. Despite the high numerical stability of the method, for large input, a huge execution time is required. For this issue, here it is proposed a related GPU implementation based on the Compute Unified Device Architecture (CUDA) framework combined with an ad-hoc memory strategy [16]. More in details, the memory strategy forces the storing into local CUDA threads block's register so that the memory access time is reduced. Experimental results confirm our contribution.

The rest of paper is organised as follows: "Mathematical details" gives preliminaries about recursive filters to approximate the discrete Gaussian convolution, with more details and features about the third order one. Moreover, the issues related to the boundary conditions are described. "GPU-parallel software for the third order RF" exhibits a detailed description of our GPU implementation, highlighting both the domain decomposition and memory management strategy. In "Experimental results", tests and experiments allow us to give interesting considerations about the reliability of our contribution and the related performance analysis. Finally, in "Conclusions" the conclusions are drawn.

Mathematical Details

This section deals with the description of the third order Gaussian recursive filter. Mathematical details about the derivation of the boundary conditions are shown. In particular, the comparison with the first-order filter, in terms of impulse response, proves the larger accuracy provided by the third order one.

From now on, we just consider $K = 1$ in Eqs. (4) and (5), that is we use the third order filter with a single iteration. Then, by considering the lighter notation:

$$s \equiv s^{(K)} = s^{(1)}, \qquad p \equiv p^{(K)} = p^{(1)}.$$

The general form of such a filter becomes:

$$p_j = \beta s_j^{(0)} + \alpha_1 p_{j-1} + \alpha_2 p_{j-2} + \alpha_3 p_{j-3}, \qquad \forall j \in \mathbf{Z}, \tag{6}$$

$$s_j = \beta p_j + \alpha_1 s_{j+1} + \alpha_2 s_{j+2} + \alpha_3 s_{j+3}, \qquad \forall j \in \mathbf{Z}. \tag{7}$$

The equations in (6) are suitably referred to as the advancing filters while Eq. (7) are called backing filters. This is because, when implementing a Gaussian RF as an algorithm, to a finite size input signal $s^{(0)}$, the index j must increase in (4) and decrease in (5).

In a general setting, the smoothing coefficients $\alpha_1, \alpha_2, \alpha_3$ and β depend on σ, the order n and the number of iterations K. In this case, their value is only a function σ. To fix them, following the approach described in [20], let us first introduce a value q, related to σ through the rules:

$$q = \begin{cases} 0.98711\,\sigma - 0.96330 & \text{for } \sigma > 2.5 \\ 3.97156 - 4.14554\sqrt{1 - 0.26891\sigma} & \text{for } 0.5 \le \sigma \le 2.5 \\ \sigma & \text{for } \sigma < 0.5 \end{cases} \tag{8}$$

Then, by setting the value

$$\psi = 3.738128 + 5.788982\,q + 3.382472\,q^2 + q^3,$$

the third order smoothing coefficients are set as:

$$\begin{aligned} \alpha_1 &= (5.788982\,q + 6.764946\,q^2 + 3\,q^3)/\psi \\ \alpha_2 &= (-3.382472\,q^2 - 3\,q^3)/\psi \\ \alpha_3 &= q^3/\psi \\ \beta &= 1 - (\alpha_1 + \alpha_2 + \alpha_3). \end{aligned} \tag{9}$$

When applying this filter, a good accuracy in approximating Gaussian convolution is guaranteed as it is shown in the experiments section. However, as already discussed in [4], this usually generates, in the algorithmic setting, distortions localised into the right boundary output entries. An example of this phenomenon, which is usually known as *edge effect*, it is shown in Fig. 1 where the output of the third order Gaussian filter (red marker), and the actual Gaussian convolution of the input signal (black solid line), are compared.

The edge effect can be attributed to the fact that to use the filter as algorithm, the right off-grid points $N+1, N+2, N+3$ are set to 0, while this property in almost never true for the convolution output $s^{(g)}$. To fix this drawback, a kind of modification of (6) and (7) for the more general case (n-order filters), which introduces the so-called *boundary conditions* or *end conditions*, has been proposed by Triggs and Sdika [18]. This approach is here described for the third order RF case.

Let us introduce the following 3×3 matrices:

$$A = \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad C = \begin{pmatrix} \beta & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{10}$$

and, for all $t \in \mathbf{Z}$, the column arrays:

$$\begin{aligned} \mathbf{p}_t &= \left(p_t, p_{t-1}, p_{t-2}\right)^T, \\ \mathbf{s}_t &= \left(s_t, s_{t+1}, s_{t+2}\right)^T, \\ \mathbf{s}_t^{(0)} &= \left(s_t^{(0)}, s_{t+1}^{(0)}, s_{t+2}^{(0)}\right)^T. \end{aligned} \tag{11}$$

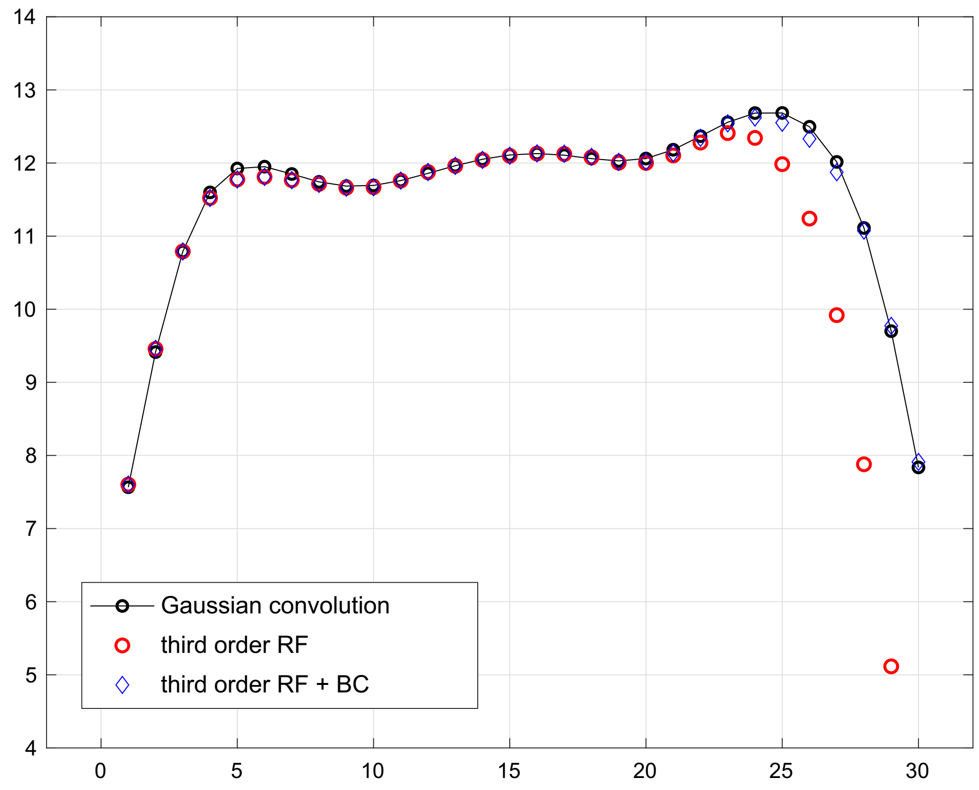


Fig. 1 Edge effect: third order RF (red line) and boundary condition correction (blue line)

With this notation, forward and backward filters in (6) and (7) can be rewritten in matrix form as:

$$\mathbf{p}_{t+1} = C\mathbf{s}_{t+1}^{(0)} + A\mathbf{p}_t, \tag{12}$$

$$\mathbf{s}_t = C\mathbf{p}_t^{(0)} + A\mathbf{s}_{t+1}. \tag{13}$$

Then, using repeatedly (12) in itself, and (13) in itself, we derive respectively:

$$\mathbf{p}_{t+k} = \sum_{l=0}^{k-1} A^l C\mathbf{s}_{t+k-l}^{(0)} + A^k\mathbf{p}_t, \tag{14}$$

$$\mathbf{s}_t = \sum_{l=0}^{k-1} A^l C\mathbf{p}_{t+l}^{(0)} + A^k\mathbf{s}_{t+k}. \tag{15}$$

If we assume the entries $s_j^{(0)}$ of the input signal are zero for $j \neq 1, 2, \dots, N$, the Eq. (14), for $k > 0$ and $t \geq N$, becomes:

$$\mathbf{p}_{t+k} = A^k\mathbf{p}_t. \tag{16}$$

Finally, by assuming the output s is bounded, substituting (16) in (15), and taking the limit for $l \to \infty$, we get:

$$\mathbf{s}_t = M\mathbf{p}_t, \qquad \text{with} \qquad M \overset{\text{def}}{=} \sum_{l=0}^{\infty} A^l C A^l. \tag{17}$$

For $t = N$, Eq. (17) takes the scalar form:

$$\begin{aligned} s_N &= M_{1,1}p_N + M_{1,2}p_{N-1} + M_{1,3}p_{N-2} \\ s_{N+1} &= M_{2,1}p_N + M_{2,2}p_{N-1} + M_{2,3}p_{N-2}, \\ s_{N+2} &= M_{3,1}p_N + M_{3,2}p_{N-1} + M_{3,3}p_{N-2} \end{aligned} \tag{18}$$

and it represents a way for priming the backing filter. In other words, the above described approach behaves as a sort of *turning* condition which takes in account all neglected equations when implementing the method in a finite setting. So, the matrix M, which only depends on σ, has to be pre-computed once in advance. This can be done, for example, by taking as an approximation of M, the partial sum:

$$M_k = \sum_{l=0}^{k} A_l, \tag{19}$$

where $A_0 = C$ and $A_l = A A_{l-1} A$ (for $l \geq 1$), provided that the norm of A_l is negligible. With this modification the third order Gaussian recursive filter can be implemented in the following Algorithm 1 where, to prime the advancing filter, we assign zero left end conditions, that is we assume to have $s_0^0 = s_{-1}^0 = s_{-2}^0 = 0$.

Algorithm 1 Third order RF with boundary conditions

```
Input: $s^{(0)}$, $\sigma$
Output: $s$
 1: set $\beta, \alpha_1, \alpha_2, \alpha_3$ as in (9);
 2: pre-compute $M$;
 3: set $N := \text{size}(s^{(0)})$
 4:     compute $p_1, p_2, p_3$     % zero left end condition
 5:         $p_1 := \beta s_1^{(0)}$
 6:         $p_2 := \beta s_2^{(0)} + \alpha_1 s_1^{(0)}$
 7:         $p_3 := \beta s_3^{(0)} + \alpha_1 s_2^{(0)} + \alpha_2 s_1^{(0)}$
 8:     for $j = 4, \dots, N$               % advancing filter
 9:         $p_j := \beta s_j + \alpha_1 p_{j-1} + \alpha_2 p_{j-2} + \alpha_3 p_{j-3}$
10:     endfor
11:     compute $s_N, s_{N+1}, s_{N+2}$ using (18)  % right end condition
12:     $s_N := M_{1,1}p_N + M_{1,2}p_{N-1} + M_{1,3}p_{N-2}$
13:     $s_{N+1} := M_{2,1}p_N + M_{2,2}p_{N-1} + M_{2,3}p_{N-2}$
14:     $s_{N+2} := M_{3,1}p_N + M_{3,2}p_{N-1} + M_{3,3}p_{N-2}$
15:     for $j = N-1, \dots, 1$             % backing filter
16:         $s_j := \beta p_j + \alpha_1 s_{j+1} + \alpha_2 s_{j+2} + \alpha_3 s_{j+3}$
17:     endfor
```

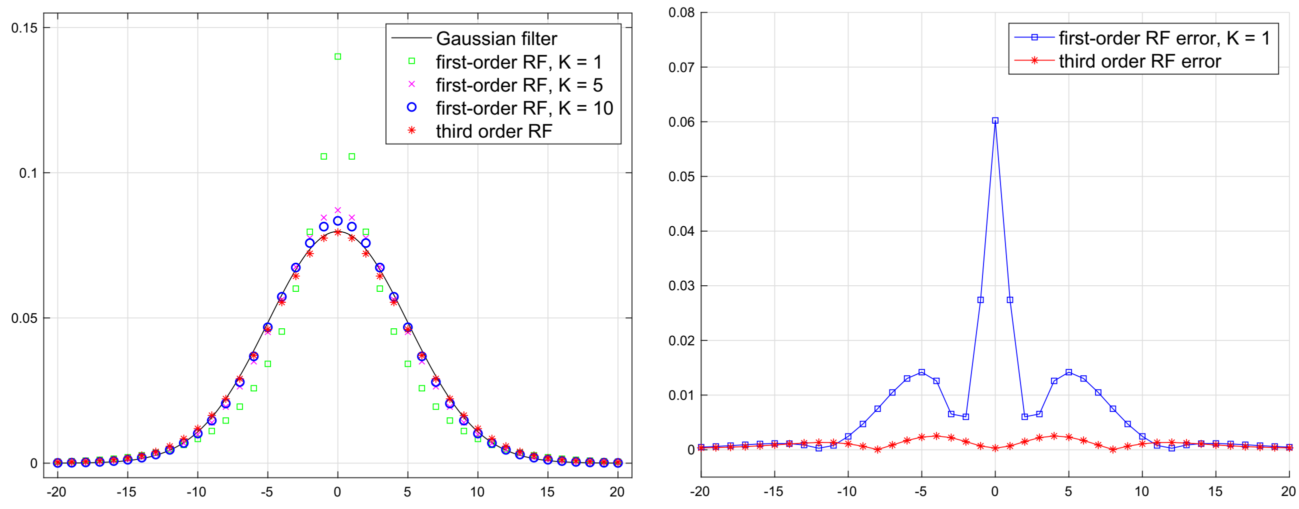


Fig. 2 Left: impulse response comparison. Right: component-wise errors

Let us observe in Fig. 1 how the boundary conditions improve the accuracy on the right boundary (blue diamond marker). In general, a way for establishing the accuracy provided by a Gaussian RF is to compare its impulse response to the Gaussian function. The impulse response represents the output of the filter when it is applied to the *unit-sample* $\delta = \{\delta_j\}_{j\in\mathbf{Z}}$, where:

$$\delta_j = \begin{cases} 1 & \text{if } \; j=0, \\ 0 & \text{otherwise} \, . \end{cases} \tag{20}$$

Figure 2 (left hand-side) compares the impulse responses of third order Gaussian RF and of the first-order one with three different iteration values ($K = 1, 5, 10$). We can observe how the impulse response of the third order is closer to the ideal Gaussian response ($\sigma = 5$) with respect to all first-order curves. In other words, the accuracy provided by this filter is larger than the one obtained by the first-order RF for all K values. This assertion is confirmed looking at a more precise measure of the error which is highlighted in Fig. 2 on the right: here the component-wise error is plotted and the red curve, related to the third order RF, is mostly below the blue curve related to the first-order RF. We remark that taking a larger K can improve the accuracy of the first-order filter to the point of to overcome the accuracy offered by the third order one. However this choice would imply a very large cost in terms of computational load. A deeper comparison is treated in Sect. "Experimental results".

GPU-Parallel Software for the Third Order RF

Despite the high numerical stability of the described method, for large input, a huge execution time is required for computing the third order RF. For this issue, we propose a GPU-CUDA implementation of Algorithm 1 described in the previous section, based on ad-hoc memory access and a suitable domain decomposition approach. More in details, our parallel strategy exploits an efficient memory strategy access, forcing the storing of data into the local CUDA threads block's register. Moreover, to obtain a reliable and performing computation, the algorithm is organised as follows.

In the first phase, `step 1`, we use a domain decomposition approach: this consists in splitting the input signal $s^{(0)}$ into $\mathbf{t}$ local blocks $s_{\mathbf{j}}^{(0),m}$ ($\mathbf{j} = 0, \dots, \mathbf{t} - 1$), one for each thread. Therefore, the input signal, of size N, is distributed at each thread $\mathbf{j}$, using the standard balanced approach, so that the local input size for each thread is $d = \left\lfloor \frac{N}{\mathbf{t}} \right\rfloor$. However, to avoid a distortion effect, similar to the one discussed in [9], that is a large perturbation on the blocks boundary entries due to the application of the third order RF to each block, an overlapping procedure is also introduced. That is, each block $s_{\mathbf{j}}^{(0),m}$ includes further $2m$ entries of the input signal, by creating overlapping areas shared by all couples of subsequent blocks. The whole procedure can be seen as a domain decomposition with overlapping strategy and m denotes the overlapping size.

In this way, each thread $\mathbf{j}$ loads in its own local memory the block $s_{\mathbf{j}}^{(0),m}$, whose size is $d + 2m$ or $d + 1 + 2m$ (depending on $\mathbf{j}$). More precisely, by using the remainder

$r = mod(N, \mathbf{t})$, the entries of the $\mathbf{j}$-th local block are formally defined as:

$$\left(s_{\mathbf{j}}^{(0),m}\right)_i = \begin{cases} s^{(0)}_{\mathbf{j}d+\mathbf{j}+i-m+1}, \; i = 0, \dots, d+2m & \text{if} \quad \mathbf{j} < r, \\ s^{(0)}_{\mathbf{j}d+r+i-m+1}, \; i = 0, \dots, d+2m-1 & \text{otherwise}, \end{cases} \tag{21}$$

where the input signal entries are set to zero, when not available ($s_i^{(0)} = 0$ for $i \le 0$ and $i > N$). Therefore, considering the overlapping entries, the local size becomes $n_{\text{loc}} + 2m$ ($n_{\text{loc}} = d$ or $n_{\text{loc}} = d + 1$).

The next phase, `step 2`, performs the third order Gaussian RF. In this step the output blocks, denoted by $s_{\mathbf{j}}^m$, are computed locally, for each thread applying Algorithm 1 to $s_{\mathbf{j}}^{(0),m}$.

Finally, in the last phase, `step 3`, the local results are collected by loading them into a global output signal. To do this, a resizing operation is performed before: that is the first and last m entries of $s_{\mathbf{j}}^m$ are removed to obtain the local outputs $s_{\mathbf{j}}$. Algorithm 2 shows the procedure in detail.

Algorithm 2 Parallel main scheme for Gaussian recursive filter based on domain decomposition with overlapping

```
Input: $s^{(0)}$, $\sigma$, $m$, $\mathbf{t}$
Output: $s$
1: FOR ALL THREAD $\mathbf{j}$
2: save in the private memory of thread the extended input signal $s_{\mathbf{j}}^{(0),m}$ as
   described in step 1 (domain decomposition with overlapping)
3: apply Algorithm 1 to $s_{\mathbf{j}}^{(0),m}$ with parameter $\sigma$ as described in step 2, to obtain
   $s_{\mathbf{j}}^m$
4: resize $s_{\mathbf{j}}^m$ to recover $s_{\mathbf{j}}$ and copy it in the shared memory in order to obtain
   the global output $s$ as described in step 3
5: ENDFOR ALL THREAD $\mathbf{j}$
```

Since the Algorithm 2 is implemented in the GPU-CUDA environment, this entails, first of all, that the input data are transferred from host to the device global memory and, after the computation, a new transfer of data from the device to host is required. But, despite the unavoidable transfer times, the overall algorithm can be executed by all threads in a fully-parallel way and this makes the execution really performing. In Algorithm 3, a more detailed version of the procedure is shown.

Algorithm 3 GPU-parallel implementation for the third order RF

```
Input: N, input_data[ ]        Output: results[ ]
1: set overlapping size value m
2: compute local size value n_loc
3: define the extended local size:
   length := 2m + n_loc
4: define the index of each thread:
   index := threadIdx.x+(blockDim.x × blockIdx.x)
5: define the local chunk interval:
   chunk_idx := (index × n_loc)+((index+1) × n_loc)
   % parallel work: begin
6: for ALL threads do
7:    set a local register memory:
      x_local[index] := input_data[chunk_idx + length]
8:    copy into local memory the selected input data chunk
9:    compute forward & backward filters on x_local[index] using Algorithm 1
10:   copy local results to global memory:
      results[n_loc] := x_local
11: end for
12: % parallel work: end
```

The parallel algorithm uses a suitable memory allocation for each thread to manage the synchronous operations. In fact, after the input signal vector of data is loaded in the global device memory, local stacks are set (see lines 1–5) for each thread by considering the padding pieces related to the overlapping value m. Each thread performs a preliminary check of the local chunk by means of the local index `chunk_idx`. Therefore, following the overlapping procedure, if the left and the right sides of the input data are provided, these values are added, otherwise m values, set to zero, are inserted on the off-grid positions. In practice, a forced usage of local register for each CUDA block is performed, by setting the CUDA kernel by means of the `__launch_bounds__` keyword, which indicates how many registers the compiler could use, for the `__global__` function. This trick was used because a good choice of the number of used registers guarantees an atomic access of each thread, per block, to its own register. Moreover, to ensure a coalescing memory access and a minimization of memory transaction, a local *array* is defined and stored into the local register. The use of the `__launch_bounds__` keyword, combined with this special array definition, forces the Parallel Thread Execution (PTX) to store the data into the local register of each block. Lines 6–11 of Algorithm 3 perform the parallel work on the GPU device and, in particular, at line 9 the call of third order RF kernel on the local `x_local[index]` input signal is carried out. Here, each thread, according to SIMT [14] paradigm, applies the filter by guaranteeing the total independent work and keeping in sure the memory transactions. While, line 11 is related to gathering of the local results of each thread in the global output. The copy operation, of data from local register to the global memory, is obtained by erasing the $2m$ overlapped values and according to avoid memory contention. Since each thread writes only the `n_loc` central elements of own local result the procedure is memory-safe. In next section, experimental results will prove the accuracy of the third order RF and the efficiency of the proposed parallel implementation.

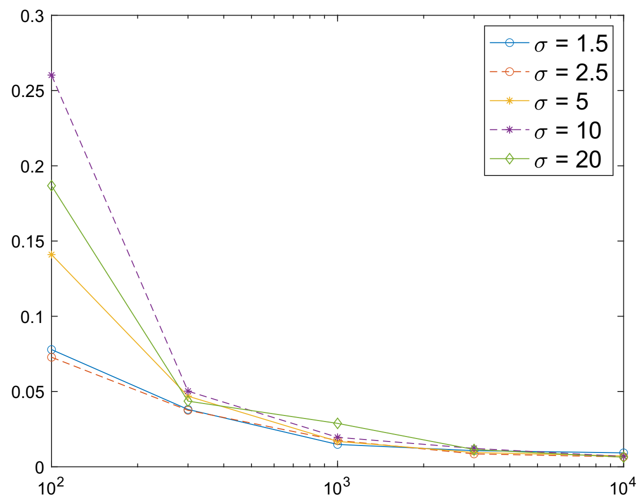
Table 1 Euclidean norm relative error $E_{3,\sigma,N}$

σ	N				
	100	316	1000	3162	10000
1.5	0.0259	0.0212	0.0198	0.0197	0.0199
2.5	0.0122	0.0101	0.0096	0.0100	0.0099
5	0.0066	0.0048	0.0047	0.0047	0.0045
10	0.0060	0.0032	0.0031	0.0026	0.0022
20	0.0077	0.0030	0.0020	0.0013	0.0011

Experimental Results

In this section, we propose several tests and experiments to prove the reliability and the main features of our implementation. In particular, the experiments are carried out in the following high-performance computer with technical specifications:

– 2 CPU Intel Xeon with 6 cores, E5-2609v3, 1.9 Ghz, 32 GB of RAM, 4 channels 51 Gb/s memory bandwidth



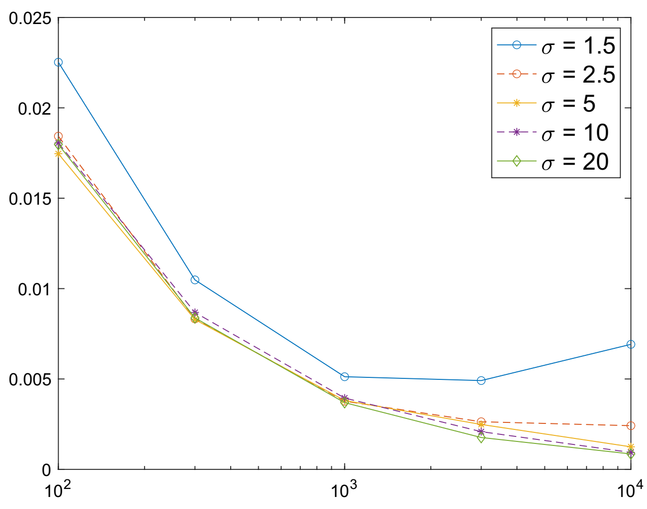


Fig. 3 Maximum (left) and mean (right) relative point-wise errors

Table 2 Comparison of GPU (more configurations) and CPU execution times for different input sizes

N	CPU	GPU configuration (block × threads)			
		1×128	1×256	1×512	1×1024
1×10^4	115.61	0.067	0.039	0.029	0.031
5×10^4	123.77	0.175	0.084	0.069	0.140
1×10^5	504.17	0.892	0.231	0.187	0.489
5×10^5	1499.42	1.535	0.699	0.431	0.913
1×10^6	4090.68	3.951	1.602	1.100	2.350

– 2 NVIDIA GeForce GTX TITAN X, 3072 CUDA cores, 1 Ghz clock for core, 12 GB DDR5, 336 Gb/s as bandwidth.

To take full advantage of the massive parallelism of the GPU environment, our implementation of Algorithm 3 is designed for CUDA, the best recent framework for exploiting all GPUs' features. The implementation aims to obtain a performance improvement by adopting specific memory management techniques. More in details, the memory strategy management provides to increase the size available of local stack and heap memory for each thread and for each threads' block. Thanks to this enhancement, in a big data context, the memory access time is reduced. To make effectively fair computations, i.e. avoiding any memory contention, we use the CUDA routine `cudaDeviceSetLimit`, by setting as:

– first parameter `cudaLimitMallocHeapSize`;
– second parameter `cudaLimitStackSize`;
– size, according to hardware architecture, the value $1024 \times 1024 \times 1024$.

By exploiting this procedure and using the `malloc` system-call, the dynamic allocation of the memory device is possible.

The following tests, executed on the aforementioned hardware, have been designed to prove the accuracy of the Algorithm 3, its performance in terms of execution time, and to compare it with a similar code which implements the first-order Gaussian recursive filter [9]. All execution times reported in the following tests are taken as averages of 10 runs.

Third Order Performance

In this section we present the tests for the third order RF GPU-parallel algorithm: the first test shows the accuracy obtained; the second test analyses the execution times and it establishes the impact of the GPU configuration; the third test measures the performance in terms of floating point operations.

Table 3 Gflops comparison: CPU vs GPU (block × thread $s = 1 \times 512$)

Input size	CPU	GPU
1×10^4	2.49	200.3
5×10^4	5.65	411.7
1×10^5	2.77	157.3
5×10^5	4.66	457.5
1×10^6	3.42	354.4

Test 1. Accuracy. In this test we measure the truncation error due to the application of the third order filter instead of the Gaussian one. To do this, we use different values of σ and we increase the input size N. Input signals s_0 are randomly distributed, uniformly in the interval [0, 1], and the error shown in Table 1 is the two-norm relative error:

$$E_{3,\sigma,N} = \frac{\|s_g - s_3\|_2}{\|s_g\|_2}, \tag{22}$$

where s_g is the Gaussian convolution of s_0 while s_3 is the output of the third order RF obtained by our parallel software.

Observe that, for all fixed value σ, the error remains almost constant as the size N increases. That is the method is accurate (below 1%) for all tests executed. Moreover, for all fixed size the error generally decreases as σ grows. To also include a point-wise analysis, we reported two images (Fig. 3) related to the component-wise error behaviour. The errors we consider are:

– the maximum relative point-wise error:

$$\text{Max\_}PE_{3,\sigma,N} = \max_{i=1,\dots,N} \left| \frac{(s_g)_i - (s_3)_i}{(s_g)_i} \right|;$$

– the mean relative point-wise error:

$$\text{Mean\_}PE_{3,\sigma,N} = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{(s_g)_i - (s_3)_i}{(s_g)_i} \right|.$$

Figure 3 shows that all components of the output have a low relative error, for all sizes and σ values (see Max_$PE_{3,\sigma,N}$ on the left). Of course, by comparing the two sub-figures, we observe that the mean error level (right curves) is always lower than the corresponding maximum value (left curves). Finally, right sub-figure highlights that all error values are, again, almost constant for all outcomes.

Test 2. Execution times acceleration and GPU configuration. In this experiment we compare execution

Table 4 Error analysis between first-order and third order Gaussian RF by varying both sigma values and iterations number for first-order

σ	$E_{3,\sigma,N}$	$E^{(K)}_{1,\sigma,N}$					
		$K = 1$	$K = 3$	$K = 5$	$K = 8$	$K = 10$	$K = 12$
1.5	0.0198	0.1031	0.0473	0.0344	0.0270	0.0246	0.0229
2.5	0.0096	0.0727	0.0254	0.0164	0.0115	0.0099	0.0088
5	0.0047	0.0465	0.0150	0.0091	0.0059	0.0048	0.0041
10	0.0031	0.0294	0.0094	0.0056	0.0035	0.0029	0.0024
20	0.0020	0.0266	0.0095	0.0058	0.0037	0.0030	0.0026

Table 5 Execution times comparison by varying input data size and σ values between first-order RF ($K = 10$) and third order RF

N	GPU time (s)					
	$\sigma = 2.5$		$\sigma = 5$		$\sigma = 10$	
	First-order	Third order	First-order	Third order	First-order	Third order
1×10^4	0.138	0.029	0.142	0.023	0.123	0.025
5×10^4	0.648	0.069	0.649	0.061	0.622	0.086
1×10^5	1.239	0.187	1.219	0.139	1.209	0.214
5×10^5	5.369	0.431	5.743	0.402	5.110	0.489
1×10^6	11.893	1.100	12.122	1.083	11.219	1.193

times achieved by using both sequential and parallel versions of the algorithm. In particular, we first execute the CPU sequential algorithm by varying several input sizes. Hence, we perform the same experiments, by using the proposed parallel implementation with several CUDA threads configurations (block $\times$ threads).

Table 2 highlights the gain of performance obtained by the GPU implementation. In particular we observe the growing of CPU times with the problem dimension and the same thing happens for all GPU configurations. However, the increase of performance of GPU implementation is noticeable and confirmed by the saved time (at least $99,8\%$ for all executions). Table 2 also reveals the best CUDA configuration for our implementation. The setting

$$\text{block} \times \text{threads} = 1 \times 512,$$

guarantees the best performance and allows us to execute a fair and thread-safe execution. We notice that outcomes get worse as the configuration thread number gets higher values (1×1024). This is because a large threads number would return an overload for each dedicated blocks' register with a related interleaving problem and a subsequent increased overhead.

Test 3. GPU Gflops analysis. In this test, to confirm the gain of performance with respect to the sequential implementation, an addiction theoretical metric, i.e. the performance analysis in Giga floating point operations per second (Gflops), is analysed. Results obtained are referred to the same experiments of previous test and to the best observed CUDA configuration.

We observe in Table 3 an appreciable enhancement of performance obtained by exploiting the GPU architecture: the performance increase, in terms of Gflops, for all executions goes from a minimum of 75× to a maximum of about 103×. This high gain is achieved thanks to the chosen memory strategy management.

Performance Comparison With the First-Order Recursive Filter

In this section we present two tests to compare performance between first-order Gaussian RF implementation proposed in [9] and the version here implemented: the first test aims to establish the number of iterations needed to obtain the same accuracy for both software; the last test compares the time executions, under the constraint of the same accuracy level.

Test 4. Accuracy comparison. In this test, we measure accuracy obtained using the third order RF and the first-order one. The test involves several σ values and seeks for the suitable number of iterations, for the first-order implementation, needed to reach the same accuracy provided by the third order one. We limit the discussion to the same input signals of Test 1, with size $N = 1000$ and compute the error by using the analogous metric defined in (22), that is we denote by $E^{(K)}_{1,\sigma,N}$ the error of the first-order RF. Notice that the error behaviour is almost constant as the input size varies (see Test 1) and this allows us to consider just a fixed value of N.

Table 4 reveals the following issues: the error decreases as σ grows for both filters; at the same time, for all fixed σ the first-order RF error falls as K increases; finally, by comparing the two filters, we obtain a similar accuracy level always starting from about $K = 10$ iterations of the first order RF, regardless of the σ value.

Test 5. Time comparison. In this test we compare the proposed implementation with the first-order one presented in [9], in terms of execution time. Since observations of previous test, to get a fair comparison, we set $K = 10$ for all executions of the first-order RF implementation. This choice guarantees that the two software give the same accuracy level. The test involves several executions by varying both the σ value and the input size value N. The CUDA configuration is set as 1×512 for both implementations. Table 5 shows that the execution time, for both implementations, seems do not depend on σ, but only on the filter and, in particular, on the data size. However, for all σ and N values, a straight comparison shows that the third order execution times are always smaller (from about 4 to 10 times) than the first-order ones. This confirms that, under the constraint of the same accuracy level, the third order RF can be considered the most efficient.

Conclusions

In this work, we presented a new GPU-parallel implementation which is based on the third order recursive filter, to approximate the Gaussian convolution operation. Starting by some preliminary results achieved by the use of the first-order Gaussian RF, here we described how the third order one can guarantee a larger accuracy at a lower computational cost. We outlined the related sequential CPU algorithm and, in addition, we proposed to accelerate it in a GPU environment. In fact, using an ad-hoc memory management of device, we exploited the massive parallelism of GPUs using a well-established strategy for data decomposition through suitable routines of the CUDA framework. This has enabled us to provide a very fast tool to approximate the Gaussian convolution, as shown in the experimental section.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Aprovitola A, Gallo L. Edge and junction detection improvement using the canny algorithm with a fourth order accurate derivative filter. In: 2014 tenth international conference on signal-image technology and internet-based systems, IEEE. 2014. pp. 104–11.
2. Chaurasia G, Ragan-Kelley J, Paris S, Drettakis G, Durand F. Compiling high performance recursive filters. In: Proceedings of the 7th conference on high-performance graphics. 2015. pp. 85–94.
3. Cuomo S, De Michele P, Galletti A, Marcellino L. A GPU parallel implementation of the local principal component analysis overcomplete method for DW image denoising. In: 2016 IEEE symposium on computers and communication (ISCC), IEEE. 2016. pp. 26–31.
4. Cuomo S, Farina R, Galletti A, Marcellino L. A K-iterated scheme for the first-order Gaussian recursive filter with boundary conditions. In: 2015 federated conference on computer science and information systems (FedCSIS), IEEE. 2015. pp. 641–47.
5. Cuomo S, Galletti A, Giunta G, Marcellino L. Numerical effects of the Gaussian recursive filters in solving linear systems in the 3Dvar case study. Numer Math Theory Methods Appl. 2017;10:3.
6. Cuomo S, Galletti A, Marcellino L. A GPU algorithm in a distributed computing system for 3D MRI denoising. In: 2015 10th international conference on P2P, parallel, grid, cloud and internet computing (3PGCIC), IEEE. 2015. pp. 557–62.
7. De Luca P, Fiscale S, Landolfi L, Di Mauro A. Distributed genomic compression in MapReduce paradigm. In: International conference on internet and distributed computing systems (2019), New York: Springer; 2019. pp. 369–78.
8. De Luca P, Galletti A, Giunta G, Marcellino L. Accelerated Gaussian convolution in a data assimilation scenario. In: International conference on computational science . New York: Springer; 2020. pp. 199–211.
9. De Luca P, Galletti A, Marcellino L. A Gaussian recursive filter parallel implementation with overlapping. In: 2019 15th international conference on signal-image technology & internet-based systems (SITIS) (2019), IEEE. 2019. pp. 641–48.
10. Gonzales RC, Woods RE. Digital image processing, 2002.
11. Gutiérrez PD, Lastra M, Benítez JM, Herrera F. SMOTE-GPU: big data preprocessing on commodity hardware for imbalanced classification. Prog Artif Intell. 2017;6(4):347–54.
12. Hewer G, Martin R, Zeh J. Robust preprocessing for Kalman filtering of glint noise. IEEE Trans Aerosp Electron Syst. 1987;1:120–8.
13. Liu H, Ong YS, Shen X, Cai J. When Gaussian process meets big data: A review of scalable GPs. IEEE Trans Neural Netw Learn Syst. 2020;31(11):4405–23.
14. László E, Giles MB, Appleyard J, Szolgay P. Methods to utilize SIMT and SIMD instruction level parallelism in tridiagonal solvers. In 2014 14th international workshop on cellular nanoscale networks and their applications (CNNA). 2014. pp. 1–2.
15. Marcellino L, Montella R, Kosta S, Galletti A, Di Luccio D, Santopietro V, Ruggieri M, Lapegna M, D'Amore L, Laccetti G. Using GPGPU accelerated interpolation algorithms for marine bathymetry processing with on-premises and cloud based computational resources. Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics) 10778 LNCS (2018), pp. 14–24.
16. NVIDIA. 2020. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
17. Steinkraus D, Buck I, Simard P. Using GPUs for machine learning algorithms. In: Eighth international conference on document

analysis and recognition (ICDAR'05) (2005), IEEE. 2005. pp. 1115–20.

18. Triggs B, Sdika M. Boundary conditions for young-van vliet recursive filtering. IEEE Trans Signal Process. 2006;54(6):2365–7.
19. Yip H-M, Ahmad I, Pong T-C. An efficient parallel algorithm for computing the gaussian convolution of multi-dimensional image data. J Supercomput. 1999;14(3):233–55.
20. Young IT, Van Vliet LJ. Recursive implementation of the gaussian filter. Signal Process. 1995;44(2):139–51.