Accelerated Gaussian Convolution in a Data Assimilation Scenario

Pasquale De Luca[1](✉), Ardelio Galletti[2], Giulio Giunta[2], and Livia Marcellino[2]

[1] Department of Computer Science, University of Salerno, Fisciano, Italy
p.deluca16@studenti.unisa.it

[2] Department of Science and Technology, University of Naples Parthenope, Naples, Italy
{ardelio.galletti,giulio.giunta,livia.marcellino}@uniparthenope.it

Abstract. Machine Learning algorithms try to provide an adequate forecast for predicting and understanding a multitude of phenomena. However, due to the chaotic nature of real systems, it is very difficult to predict data: a small perturbation from initial state can generate serious errors. Data Assimilation is used to estimate the best initial state of a system in order to predict carefully the future states. Therefore, an accurate and fast Data Assimilation can be considered a fundamental step for the entire Machine Learning process. Here, we deal with the Gaussian convolution operation which is a central step of the Data Assimilation approach and, in general, in several data analysis procedures. In particular, we propose a parallel algorithm, based on the use of Recursive Filters to approximate the Gaussian convolution in a very fast way. Tests and experiments confirm the efficiency of the proposed implementation.

Keywords: Gaussian convolution · Recursive filters · Parallel algorithms · GPU

1 Introduction

Data Assimilation (DA) is a *prediction-correction* method for combining a physical model with observations. The Data Assimilation and the Machine Learning (ML) fields are closely related to each other. Machine Learning process is used to perform a specific task without using explicit instructions and it can be seen as a subset of the Artificial Intelligence (AI) field, because it creates new methods and applications for analyze and classify many natural phenomena (see for example [1]). In general, this process consists in two main phases: the *analysis phase* - some collected data are analyzed to detect patterns that help to create explicit features or parameters; the *training phase* - data parameter generated in the previous phase are used to create Machine Learning models.

However, the learning part of the training phase relies on a relevant training data-set, containing samples of spatio-temporal dependent structures. In many fields, there is an absence of direct observations of the random variables, and therefore, learning techniques cannot be readily deployed.

Therefore, a correct training dataset is a basic need to get a right learning. This is because in order to perform a correct ML approach, often a classifier is used in the analyze phase. Each classifier is composed by a kernel which aims to correctly predict the classes by using a higher-dimension feature space to make data almost linearly separable. In order to compute a fair classification and accurate prediction a suitable method could be chosen [23].

The variational approach of the Data Assimilation process, characterized by a cost function minimization, is a good choice for classification. Numerically, this means to apply an iterative procedure using a covariance matrix defined by measuring the error between predictions and observed data. Here, we are interested in those numerical issues. In particular, since the error covariance matrix presents a Gaussian correlation structure, the Gaussian convolution process plays a key role in such a problem. Furthermore, it should be noted that, beyond its fundamental role in the Data Assimilation field, the convolution operation is always significant in the computational process of most big-data analysis problems. Hence, a correct Machine Learning process can use it as a basic step in the analysis phase. Moreover, because of the need to process large amount of data, parallel approaches and High Performance Computing (HPC) architectures, as multicore or Graphics Processing Units (GPUs), are mandatory [2–4]. In this direction, some recent papers deal with parallel data assimilation [5–7] but we just limit our attention to the basic step represented by a parallel implementation for the Gaussian Convolution. In particular, we propose an accelerated procedure to approximate the Gaussian convolution which is based on Recursive Filters (RFs). In fact, Gaussian RFs have been designed to provide an accurate and very efficient approximated Gaussian convolution [8–11]. Since the use of RFs is mainly suitable to overcome a large execution time, when there is a lot of data to analyze, many parallel implementations have been presented (see survey in [12]). Here, we propose a novel implementation that exploits the computational power of the GPUs which are very useful for solving numerical problems in several application fields [13,14].

More precisely, to manage big size input data, the parallelization strategy is based on a domain decomposition approach with overlapping, so that all possible interactions between forecasts and observations are included. In this way, this computational step becomes a very fast kernel specifically designed for exploiting the dynamic parallelism [15] approach available on the Compute Unified Device Architecture (CUDA) [16].

The paper is organized as follows. Section 2 recalls the variational Data Assimilation problem, and the use of the Recursive Filter to approximate the discrete Gaussian convolution. In Sect. 3, the underlying domain decomposition strategy and the GPU-CUDA parallel algorithm are provided. The experiments in Sect. 4 confirm the efficiency of the proposed implementation in terms of performance. Finally conclusions are drawn in Sect. 5.

2 Gaussian Convolutions in Data Assimilation

In this section, we show how the Gaussian convolution is involved in a Data Assimilation scenario. In particular, let us consider a three-dimensional variational data assimilation problem [17]: the objective is to give a best estimate of x, that is called the analysis or state vector, once a prior estimate vector x^b (background), usually provided by a numerical forecasting model, and a vector $y = \mathcal{H}(x) + \delta y$ of observations, related to the nonlinear model $\mathcal{H}$, are given. The unknown x solves the regularized constrained least-squared problem:

$$\min_x J(x) = \min_x \left[\|y - \mathcal{H}(x)\|^2 + \|x - x^b\|^2 \right], \tag{1}$$

where J denotes the objective function to minimize. Here, $\|x - x^b\|^2$ is a penalty term and $\|y - \mathcal{H}(x)\|^2$ is a quadratic data-fidelity term which compares measured data and solution obtained by the nonlinear model $\mathcal{H}$ [10]. In this scheme, the background error $\delta x = x^b - x$ and the observational error $\delta y = y - \mathcal{H}(x)$ are assumed to be random variables with zero mean and covariance matrices

$$\mathbf{B} = <\delta x, \delta x^T> \qquad \text{and} \qquad \mathbf{R} = <\delta \mathrm{y}, \delta \mathrm{y}^{\mathrm{T}}>,$$

respectively. Following description in [9], let the matrix $\mathbf{H}$ be a first-order approximation of the Jacobian of $\mathcal{H}$ at x^b and denote by

$$d = y - \mathcal{H}(x^b)$$

the so-called *misfit*. Denoting by $\mathbf{V}$ the unique symmetric Gaussian matrix such that $\mathbf{V^2} = \mathbf{B}$, and by introducing the variable $v = \mathbf{V}^{-1}\delta x$, the problem (1) can be proven to be equivalent to [9,18,19]:

$$\min_v \widetilde{J}(v) = \min_v \frac{1}{2}(d - \mathbf{HV}v)^T \mathbf{R}^{-1}(d - \mathbf{HV}v) + \frac{1}{2} v^T v. \tag{2}$$

The minimization of the cost function $\widetilde{J}(v)$ leads to the linear system:

$$(I + \mathbf{V}\Psi\mathbf{V})v = \mathbf{VH}^T\mathbf{R}^{-1}d. \tag{3}$$

Since $I + \mathbf{V}\Psi\mathbf{V}$ is symmetric, the linear system (3) that can be handled by means of the CG method, whose basic operation is the matrix-vector multiplication:

$$(I + \mathbf{V}\Psi\mathbf{V})\rho = \rho + \mathbf{V}\Psi\mathbf{V}\rho.$$

Here, $\Psi = \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}$ is a diagonal matrix and ρ denotes the residual at the current step of the CG algorithm. More precisely, it turns out that such an operation involves three discrete Gaussian convolutions:

$$\mathbf{V}\rho, \qquad \mathbf{V}(\Psi\mathbf{V}\rho), \qquad \mathbf{V}(\mathbf{H}^T\mathbf{R}^{-1}d). \tag{4}$$

In conclusion, previous analysis shows that Gaussian convolution becomes a main kernel for Data Assimilation. From here comes the need to implement accurate and fast methods to perform it. In fact, in the described context, the matrix $\mathbf{V}$ is neither effectively used nor even assembled, and the matrix-vector multiplications in (4) are computed by introducing the so-called Gaussian Recursive Filters. It has been proved that these tools offer good accuracy and bring down the computational cost in time and space [20,21].

In particular, in this work we just consider K-iterated first-order Gaussian RFs and follow the approach and notation used in [8]. Let:

$$s^{(0)} = \{s_j^{(0)}\}_{j\in\mathbf{Z}} = \left(\dots, s_{-1}^{(0)}, s_0^{(0)}, s_1^{(0)}, \dots\right)$$

be an input signal and let g denote the Gaussian function with zero mean and standard deviation σ. The Gaussian filter is a filter whose response to the input $s^{(0)}$ is given by the discrete Gaussian convolution:

$$s_j^{(g)} = \left(g * s^{(0)}\right)_j = \sum_{t\in\mathbf{Z}} g_{j-t} s_t^{(0)}, \qquad \forall j \in \mathbf{Z}, \tag{5}$$

where $g_t \equiv g(t)$. A K-iterated first-order Gaussian recursive filter generates an output signal $s^{(K)}$, the so-called K-iterate approximation of $s^{(g)}$, whose entries solve the $2K$ recurrence relations:

$$p_j^{(k)} = \beta s_j^{(k-1)} + \alpha p_{j-1}^{(k)}, \qquad \forall j \in \mathbf{Z}, \tag{6}$$

$$s_j^{(k)} = \beta p_j^{(k)} + \alpha s_{j+1}^{(k)}, \qquad \forall j \in \mathbf{Z}. \tag{7}$$

$(k = 1, \dots, K)$ where values α and $\beta = 1 - \alpha$ are called *smoothing coefficients* and verify:

$$\alpha = 1 + E_\sigma - \sqrt{E_\sigma(E_\sigma + 2)}, \qquad \beta = \sqrt{E_\sigma(E_\sigma + 2)} - E_\sigma, \tag{8}$$

with $E_\sigma = K\sigma^{-2}$. It has been proved that as $K \to \infty$ the filter converges to the Gaussian filter [22]. If we consider a finite size input signal $s^{(0)}$ (i.e. with support in the grid $\{0, 1, 2, \dots, N-1\}$) then the index j has to be used in increasing order in (6) and decreasing order in (7). Hence, relations (6) and (7) are suitable called advancing and backing filters, respectively [8]. We highlight that to prime the algorithm these filters requires to set values $p_0^{(k)}$ and $s_{N-1}^{(k)}$. This can be done using the boundary conditions [24]:

$$p_0^{(k)} = \frac{1}{1+\alpha} s_0^{(k-1)}, \qquad s_{N-1}^{(k)} = \frac{1}{1+\alpha} p_{N-1}^{(k)}$$

which are derived to simulate the effect of the neglected entries when using finite size input signals. Typically a well-known edge effect, i.e. a large perturbation error, can be seen on the boundary entries of the output. In [8], provided that the input support is in $[0, N-1]$, this effect can be mitigated by increasing the input size including and putting artificial zero entries at the left and right boundaries of the input. Algorithm 1 describes a K-iterated first-order Gaussian RF straight implementation.

Algorithm 1. K-iterated first-order RF with boundary conditions

```
Input: $s^{(0)}$, $\sigma$, $K$
Output: $s^{(K)}$
 1: set $\beta, \alpha$ as in (8); $M := 1/(1+\alpha)$
 2: set $N :=$ size($s^{(0)}$)
 3: for $k = 1, 2, \ldots, K$                          % filter loop
 4:     compute $p_0^{(k)} := M s_0^{(k-1)}$     % left end condition
 5:     if $k = 1$ then
 6:         $p_0^{(k)} := \beta s_0^{(k-1)}$
 7:     end
 8:     for $j = 1, \ldots, N-1$                      % advancing filter
 9:         $p_j^{(k)} := \beta s_j^{(k-1)} + \alpha p_{j-1}^{(k)}$
10:     endfor
11:     compute $s_{N-1}^{(k)} := M p_{N-1}^{(k)}$   % right end condition
12:     for $j = N-2, \ldots, 0$                      % backing filter
13:         $s_j^{(k)} := \beta p_j^{(k)} + \alpha s_{j+1}^{(k)}$
14:     endfor
15: endfor
```

3 Parallel Approach and GPU Algorithm

In this section we give a description of our parallel algorithm, and the related strategy, to implement a fast and accurate version of the K-iterated first-order Gaussian RF. This approach exploits the main features of the GPU environment. The main idea relies on several macro steps in order to obtain a reliable and performing computation. The whole process can be partitioned in three steps. In the first phase, `step 1`, in order to perform a fair workload distribution, we use a Domain Decomposition (DD) approach with overlapping. More specifically, the strategy consists in splitting the input signal $s^{(0)}$ into $\mathbf{t}$ local blocks, one for each thread:

$$s_{\mathbf{0}}^{(0),m},\ s_{\mathbf{1}}^{(0),m}, \ldots,\ s_{\mathbf{t}-1}^{(0),m}. \tag{9}$$

Here, N denotes the problem size, while:

$$d = \left\lfloor \frac{N}{\mathbf{t}} \right\rfloor \quad \text{and} \quad \mathrm{r} = \mathrm{mod}(\mathrm{N}, \mathbf{t}) \tag{10}$$

are the quotient and the remainder when dividing N by $\mathbf{t}$, respectively. Moreover, the parameter m denotes the overlapping size. To be specific, each thread $\mathbf{j}$ loads in own local memory the block $s_{\mathbf{j}}^{(0),m}$, whose size is $d + 2m$ or $d + 1 + 2m$ (depending on $\mathbf{j}$). The entries of the $\mathbf{j}$-th local block are formally defined using the subdivision:

$$\left(s_{\mathbf{j}}^{(0),m}\right)_i = \begin{cases} s^{(0)}_{\mathbf{j}d+\mathbf{j}+i-m},\ i = 0,\ldots,d+2m & \text{if } \mathbf{j} < \mathrm{r} \\ s^{(0)}_{\mathbf{j}d+r+i-m},\ i = 0,\ldots,d+2m-1 & \text{if } \mathbf{j} \geq \mathrm{r} \end{cases} \tag{11}$$

where the input signal entries are set to zero, when not available ($s_i^{(0)} = 0$ for $i < 0$ and $i \geq N$).

In other words, this partitioning consists in assigning to each thread a part of the signal, so that two consecutive threads have consecutive signal blocks and those blocks overlap on the edges by sharing exactly $2m$ entries. The reason of overlapping is because, to perform a good approximation of the convolution, block edge values need to use close values that lie in the neighboring blocks. We notice that by setting $m = 0$, i.e. by excluding the overlapping areas, could create possible perturbation errors and generate a bad accuracy close to the boundaries of the local output signals.

The `step 2` deals with the approximated local Gaussian convolution for each block. More precisely, each thread $\mathbf{j}$ performs the K-iterated first-order Gaussian RF to $s_{\mathbf{j}}^{(0),m}$, by applying Algorithm 1, and computes $s_{\mathbf{j}}^{(K),m}$.

The last phase, `step 3`, is related to collect the local approximated results by loading them into a global output signal. Therefore, in order to remove the first and last m entries, a *resizing* operation is firstly performed for each local output. More in details, each thread $\mathbf{j}$ resizes the local computed signal $s_{\mathbf{j}}^{(K),m}$, by removing its first and last m entries, and it generates the local output $s_{\mathbf{j}}^{(K)}$. Finally, a gathering of local resized outputs into the global output signal is done.

A very important consequence of our strategy is that all previous steps, which are summarized in the following parallel algorithm, can be computed by all threads in a fully-parallel way.

Algorithm 2. Parallel K-iterated first-order Gaussian recursive filter based on domain decomposition with overlapping

```
Input: $s^{(0)}$, $\sigma$, $m$, $K$, t
Output: $s^{(K)}$
1: FOR ALL THREAD 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 parameters $\sigma$, $K$ as described in step 2,
       to obtain $s_{\mathbf{j}}^{(K),m}$
4:     resize $s_{\mathbf{j}}^{(K),m}$ to recover $s_{\mathbf{j}}^{(K)}$ and copy it in the shared memory in order
       to obtain the global output $s^{(K)}$ as described in step 3
5: ENDFOR ALL THREAD j
```

Now, we discuss how the Algorithm 2 is implemented in a CUDA environment. Firstly, input data are transferred to device global memory. Hence, in order to guarantee a reliable workload distribution, the described domain decomposition in `step 1` is performed. More in detail, we set for each thread the local size $n_{loc} = d$ or $n_{loc} = d + 1$, depending on the threads number **j** and the value r in (10). This confirms that if the input size value n is not divisible by **t**, according to (11) a suitable workload distribution is done. By also considering the overlapping entries, the block length becomes $n_{loc} + 2m$, and each thread can retrieve from the global array the required amount of data needed for its local computation.

Moreover, an adequate access to the global memory is performed by means of a suitable indexing, i.e. every thread loads data from global memory and stores them in its own local memory in order to perform each operation independently. Thanks to this operation, any overhead due to the contention and synchronization of the global memory is avoided. In the following, the overall GPU parallel algorithm is shown.

Algorithm 3. GPU parallel implementation

```
    Input: N, input_data[], K
 1: % set overlapping size value m
 2: % compute local size value n_loc
 3: % define the extended local size
 4: length = 2m + n_loc
 5: % define the index of each thread
 6: index = threadIdx.x+(blockDim.x × blockIdx.x)
 7: % define the local chunk interval
 8: chunk_idx = (index × n_loc)+((index+1) × n_loc)
 9: % parallel work: begin
10: for each thread do
11:     % compute the chunk overlapped from input
12:     x_local[index] = input_data[chunk_idx + length]
13:     % start the dynamic parallelism region, by setting the threads number using the
        iteration number K
14:     for each thread in dynamic region do
15:         % compute forward & backward filter
16:         x_local[index]
17:     end for
18:     %end the dynamic parallelism region
19:     % collect local results
20:     results[n_loc] = x_local
21: end for
22: % parallel work: end
    Output: results[ ]
```

Shortly, in Algorithm 3, starting from the input data size N, the iteration number K and the input signal vector `input_data`, which are loaded in the global device memory, the procedure returns the approximated Gaussian convolution by the signal vector `results` which is computed in a GPU-parallel way. More in detail, Algorithm 3 highlights several memory and computation strategies.

To be specific, first operations provide, lines 1–8 to set the local stacks for each thread by considering the padding pieces related to the overlapping value m. Hence, according to `step 1` each thread performs a preliminary check of the local chunk by means of the local index `chunk_idx`. Therefore, if the left and the right side of the input data are provided, these values are added, otherwise m values, set to zero, are inserted on the overlapped positions. In lines 9–18 the computation phase is performed and a *dynamic parallelism* approach [15] has been applied, when possible. CUDA allows us to exploit the dynamic parallelism which is an extension to the CUDA programming model by enabling a CUDA kernel to configure new thread grids in order to launch new kernels for reducing the computational time. The aim of dynamic parallelism in our implementation consists in to the assignment, by each thread corresponding to each input portion, therefore for every CUDA kernel, to K threads by scheduling each thread in order to perform the forward and the backward filter operations as described in Algorithm 1, in synchronous way. More in details, K different threads perform the operations on each element following a pipeline modality. The usage of dynamic parallelism is able to obtain very low execution times, despite the predictable start-up and end-up times. The lines 19–22 are related to gathering of the local results of each thread in the global output. The copy operation is designed according to avoid memory contention, so that it is memory-safe because each thread carries out only the `n_loc` central elements of own local result by removing the $2m$ boundary values. This property guarantees a strong memory consistency.

4 Experimental Results

In this section, several experimental results highlight and confirm the reliability and the efficiency of the proposed software. Following, the technical specifications where the GPU-parallel algorithm has been implemented, are shown:

– two CPU Intel Xeon with 6 cores, E5-2609v3, 1.9 GHz, 32 GB of RAM, 4 channels 51 Gb/s memory bandwidth
– two NVIDIA GeForce GTX TITAN X, 3072 CUDA cores, 1 GHz Core clock for core, 12 GB DDR5, 336 GBs as bandwidth.

Thanks to GPUs' computational power, our algorithm exploits the CUDA framework in order to take best advantage of parallel environment. Our approach relies on an ad-hoc memory strategy which provides to increase the size of local stack heap memory for each thread and for each thread blocks'. Exploiting this technique, when a large amount of input data will be loaded, the memory access time is reduced. Previous operations are executed by using the following CUDA routine: `cudaDeviceSetLimit`, by setting as first parameter `cudaLimitMallocHeapSize` and second `cudaLimitStackSize`; while as size, according to hardware architecture, the value $1024 \times 1024 \times 1024$ is fixed.

This trick allows us to allocate the dynamic memory, by using `malloc` system-call, directly on the device.

Therefore, in order to increase the performances an additional memory-based operation has been done. More precisely, this operation relies on L2 cache obtaining a gain of performance by varying dynamically the fetch granularity. More in details, after that each thread blocks computation is completed, we perform a dynamic fetch granularity by using the CUDA routine `cudaDeviceSetLimit` and setting as parameters: `cudaLimitMaxL2FetchGranularity` and `128*sizeof(int)`. The value 128 is related to hardware architecture that can be support this range of data loading. Applying this approach an appreciable increasing of performance has been obtained by exploiting the memory cache's property that can recover the most used data and instructions during the execution. Accordingly, due to the canonical operations of Recursive Filters during their execution, a reduced memory access time is obtained by increasing the fetch granularity.

In other words, in a classical execution each thread accesses to the global memory to retrieve the required data for the computation. In this case, according to the memory hierarchy and the L2 memory strategy, each thread accesses first in the cache, then in the local stack, and finally in the heap/global memory. With this procedure a considerable gain in terms of performance has been achieved. In the following tests we set $\sigma = 2$ and input signals randomly distributed (Gaussian or uniform). The choice $m = 2.5\sigma = 5$ guarantees a good accuracy level, as shown in [8,20].

Test 1. Here, in order to highlight the performance gain, we set as input: $N = 10^5$, $m = 5$, $K = 10$ and the thread number $\mathtt{t} = 100$. Averaged times, related to 10 executions, achieved are:

- 7.24 s, without increasing fetch granularity,
- 6.93 s, by varying dynamic fetch granularity.

The first test highlights a small time difference but, if we give a large dataset input, which requires a large execution time even on GPU, thanks to the granularity of the dynamic recovery a significant performance gain can be obtained. Thus, the dynamic operations are closely related to the size of the input, i.e. according to cache granularity size chosen as parameter into function `cudaDeviceSetLimit`, where in this case the maximum value is fixed to 128 bytes. This experiment provides a comparison among serial and GPU parallel execution times. More precisely, in Table 1 the execution times for both serial (CPU) and parallel version (GPU) by choosing different input sizes and the iteration numbers are shown. The input parameters are set as: `Blocks` $\times$ `Threads` $= 10 \times 100$ and $m = 5$.

Table 1. Execution times (in seconds), `Blocks` $\times$ `Threads` $= 10 \times 100$, $m = 5$.

K	N			
	1×10^4		5×10^4	1×10^5
10	GPU	0.274	3.243	10.219
	CPU	5.892	129.790	403.871
50	GPU	0.483	3.592	11.883
	CPU	6.293	149.592	518.560
100	GPU	0.596	4.711	12.703
	CPU	8.431	172.197	650.195
500	GPU	0.778	5.719	13.177
	CPU	9.324	195.150	757.912
1000	GPU	0.984	6.135	14.297
	CPU	129.790	223.542	875.442

Test 2. This experiment confirms the reliability effects when choosing different CUDA thread configurations. Here, we emphasize the different input sizes given, while the iteration number and the overlapping value are set to $K = 500$ and $m = 5$, respectively. Indeed, reduction of execution times has been achieved by decreasing the `Blocks` number, this holds true for all given input sizes. This phenomenon is related to a good synchronization applied during the access to the global memory from each thread, which reduces the access time and consequently the overall execution time. These results are confirmed and verified also by choosing any possible CUDA configuration in the range 1000–3072 threads (3072 is the maximum threads number available for our hardware). Table 2 confirms the reliability of the parallelization strategy by highlighting the access time

Table 2. Execution times (in seconds), iteration number $K = 500$. $m = 5$.

CUDA configuration	N		
	1×10^4	5×10^4	1×10^5
10×100	0.77	5.72	13.17
4×250	0.59	5.01	11.88
2×512	0.41	4.07	10.91
1×1024	0.38	3.76	9.68
20×100	0.55	6.24	11.57
8×250	0.47	4.92	10.71
4×512	0.39	3.67	10.42
2×1024	0.31	3.10	8.96
30×100	0.21	3.24	6.58
12×250	0.17	3.12	5.70
6×512	0.14	3.02	4.99
3×1024	0.12	2.99	4.07

to the global memory. In particular, the results allow us to find the best CUDA thread configuration, `Blocks` $\times$ `Threads` $= 3 \times 1024$, obtained in correspondence of the best execution times.

Test 3. This experiment is referred to the optimal CUDA configuration and aims to investigate the behaviour of the algorithm by varying both iteration number value K and the input size N. Figure 1 shows an appreciable gain of performance and, in particular, a sub-linear increase of execution time with respect to the problem size (which is linear in $N \times K$), typical for GPUs architectures.

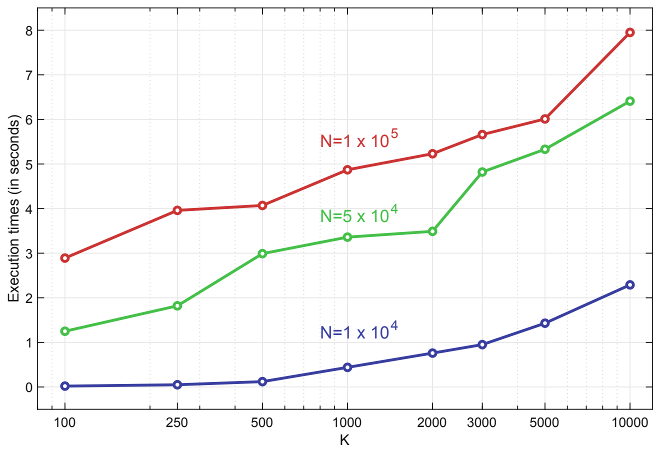

Fig. 1. Execution times by varying K and N (`Blocks` $\times$ `Threads` $= 3 \times 1024$, $m = 5$)

Test 4. Here, we show a further improvement of the performance due to the use the power of dynamic parallelism approach. Table 3 exhibits the best execution times achieved by using the dynamic parallelism and choosing an ad-hoc, i.e. limited by our machine available resources, CUDA configuration. Comparison with Table 2 (first 4 lines) confirms the improvement for all data sizes. However, we underline that, because the hardware limits available, if we set a too large threads number, a big portion of them cannot work and, from a numerical point of view, the output result becomes completely unreliable. In other words, a fair CUDA configuration avoids a failed computation. For this reason, we have no results by using a greater number of threads. Finally, behaviour of results in Table 3 seems to suggest that an improving of performance should be obtained by exploiting a machine with higher computational resources.

Table 3. Execution times (in seconds) with dynamic parallelism, iteration number $K = 500$. $m = 5$.

CUDA configuration	N		
	1×10^4	5×10^4	1×10^5
10×100	0.28	4.64	10.09
4×250	0.21	3.86	9.71
2×512	0.19	3.60	8.15
1×1024	0.13	3.09	6.92

5 Conclusions

In this paper, we proposed a GPU-parallel algorithm that provides a fast and accurate Gaussian convolution, which is a fundamental step in both Data Assimilation and Machine Learning fields. The algorithm relies on the K-iterated first-order Gaussian Recursive filter. The parallel algorithm is designed by exploiting dynamic parallelism available in CUDA environment. The experimental results confirm the reliability and the efficiency of the proposed algorithm.

References

1. De Luca, P., Fiscale, S., Landolfi, L., Di Mauro, A.: Distributed genomic compression in MapReduce paradigm. In: Montella, R., Ciaramella, A., Fortino, G., Guerrieri, A., Liotta, A. (eds.) IDCS 2019. LNCS, vol. 11874, pp. 369–378. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34914-1_35
2. De Luca, P., Galletti, A., Giunta, G., Marcellino, L., Raei, M.: Performance analysis of a multicore implementation for solving a two-dimensional inverse anomalous diffusion problem. In: Sergeyev, Y.D., Kvasov, D.E. (eds.) NUMTA 2019. LNCS, vol. 11973, pp. 109–121. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39081-5_11
3. De Luca, P., Formisano, A.: Haptic data accelerated prediction via multicore implementation. In: Kohei, A. (ed.) Proceedings of the 2020 Computing Conference, CompCom. Advances in Intelligent Systems and Computing. Springer, Cham (2020)
4. Giunta, G., Montella, R., Mariani, P., Riccio, A.: Modeling and computational issues for air/water quality problems: a grid computing approach. Nuovo Cimento C Geophys. Space Phys. C **28**, 215 (2005)
5. Rao, V., Sandu, A.: A time-parallel approach to strong-constraint four-dimensional variational data assimilation. J. Comput. Phys. **313**, 583–593 (2016)
6. Bousserez, N., Guerrette, J.J., Henze, D.K.: Enhanced parallelization of the incremental 4D-Var data assimilation algorithm using the randomized incremental optimal technique (RIOT). Q. J. R. Meteorol. Soc. **146**, 1351–1371 (2020)
7. Fisher, M., Gürol, S.: Parallelization in the time dimension of four-dimensional variational data assimilation. Q. J. R. Meteorol. Soc. **143**(703), 1136–1147 (2017)
8. Galletti, A., Giunta, G.: Error analysis for the first-order Gaussian recursive filter operator. In: 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), pp.673–378. IEEE (2016). APA

9. 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., 520–540 (2017). https://doi.org/10.4208/nmtma.2017.m1528
10. D'Amore, L., Arcucci, R., Marcellino, L., Murli, A.: A parallel three-dimensional variational data assimilation scheme. In: AIP Conference Proceedings, vol. 1389, no. 1, pp. 1829–1831. American Institute of Physics, September 2011
11. 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), Sorrento, Italy, pp. 633–640 (2019)
12. Chaurasia, G., Kelley, J.R., Paris, S., Drettakis, G., Durand, F.: Compiling high performance recursive filters. In: Proceedings of the 7th Conference on High-Performance Graphics, pp. 85–94 (2015)
13. De Luca, P., Galletti, A., Ghehsareh, H.R., Marcellino, L., Raei, M.: A GPU-CUDA framework for solving a two-dimensional inverse anomalous diffusion problem. In: Foster, I., Joubert, G.R., Kučera, L., Nagel, W.E., Peters, F. (eds.) Parallel Computing: Technology Trends, Advances in Parallel Computing, vol. 36, pp. 311–320. IOS Press, Amsterdam (2020). https://doi.org/10.3233/APC200056
14. Cuomo, S., Michele, P. D., Galletti, A., Marcellino, L.: A GPU-parallel algorithm for ECG signal denoising based on the NLM method. In: 2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA), Crans-Montana, pp. 35–39 (2016)
15. Jones, S.: Introduction to dynamic parallelism. In: GPU Technology Conference Presentations, vol. 338, p. 2012, May 2012
16. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
17. Ghil, M., Malanotte-Rizzoli, P.: Data assimilation in meteorology and oceanography. In: Dmowska, R., Saltzman, B. (eds.) Advances in Geophysics, vol. 33, pp. 141–266. Elsevier, New York (1991)
18. Lorenc, A.C.: Development of an operational variational assimilation scheme. J. Meteorol. Soc. Jpn **75**, 339–346 (1997)
19. Hayden, C., Purser, R.J.: Recursive filter objective analysis of meteorological field: applications to NESDIS operational processing. J. Appl. Meteorol. **34**, 3–15 (1995)
20. Cuomo, S., Farina, R., Galletti, A., Marcellino, L.: An error estimate of Gaussian recursive filter in 3Dvar problem. In: 2014 Federated Conference on Computer Science and Information Systems, Warsaw, pp. 587–595 (2014). https://doi.org/10.15439/2014F279
21. Young, I.T., van Vliet, L.J.: Recursive implementation of the Gaussian filter. Signal Process. **44**, 139–151 (1995)
22. Wells, W.M.: Efficient synthesis of Gaussian filters by cascaded uniform filters. IEEE Trans. Pattern Anal. Mach. Intell. **2**, 234–239 (1986)
23. Gilbert, R.C., Richman, M.B., Trafalis, T.B., Leslie, L.M.: Machine learning methods for data assimilation. In: Computing Intelligence Architecturing Complex Engineering Systems, pp. 105–112 (2010)
24. Triggs, B., Sdika, M.: Boundary conditions for Young-van Vliet recursive filtering. IEEE Trans. Signal Process. **54**(6 I), 2365–2367 (2006)