

On The Performance of Prefix-Sum Parallel Kalman Filters and Smoothers on GPUs

Simo Särkkä, *Senior Member, IEEE*, Ángel F. García-Fernández

Abstract—This paper presents an experimental evaluation of parallel-in-time Kalman filters and smoothers using graphics processing units (GPUs). In particular, the paper evaluates different all-prefix-sum algorithms, that is, parallel scan algorithms for temporal parallelization of Kalman filters and smoothers in two ways: by calculating the required number of operations via simulation, and by measuring the actual run time of the algorithms on real GPU hardware. In addition, a novel parallel-in-time two-filter smoother is proposed and experimentally evaluated. Julia code for Metal and CUDA implementations of all the algorithms is made publicly available.

Index Terms—Kalman filtering, smoothing, parallel computation, all-prefix sums, GPU, Julia.

I. INTRODUCTION

KALMAN filters and smoothers [1]–[4] are classical algorithms for state estimation in stochastic dynamic systems. They are widely used in target tracking, process control, mobile computing, and other sensor fusion applications [5], [6]. The classical Kalman filter and smoother algorithms [1]–[6] are sequential algorithms that loop over the measurement data (or length T) in forward and backward directions, respectively. Although they have linear time complexity $O(T)$ and are time-optimal in the sequential sense, they are suboptimal on parallel hardware such as graphics processing units (GPUs). Furthermore, in many applications, such as sensor fusion systems in mobile phones, it would be desirable to run the Kalman filters and smoothers on the GPU, thereby freeing the main central processing unit (CPU) for other tasks. Unfortunately, Kalman filters and smoothers in their classical form are very slow and inefficient in terms of resource utilization when run on a GPU.

Särkkä & García-Fernández [7] developed parallel-in-time algorithms for Kalman filtering and smoothing, which provide a solution to the problem above. The algorithms convert the sequential computation of $O(T)$ steps into $O(\log T)$ parallel steps, which utilize the parallel GPU hardware more effectively. The algorithms are based on reformulating the filtering and smoothing problems as the computation of generalized prefix sums for suitably defined associative operators. The ideal time complexity $O(\log T)$ results from the span complexity of the underlying prefix-sum (i.e., scan) algorithm [8], [9], which, though, is only realized in practice when the number of computational cores is high enough. These algorithms

enable the execution of Kalman filters and smoothers on GPUs significantly faster, utilizing the GPU’s parallel computational resources more effectively. Alternatively, or additionally, it would be possible to parallelize the Kalman filter equations themselves [10]–[12], but in this paper, the focus is on the temporal parallelization.

Although the reference [7] only analyzed the computational advantage of the parallel algorithms by using simulated parallel hardware, supplemental JAX [13] and TensorFlow [14] implementations for GPUs were later provided¹. Furthermore, the algorithms, or actually their extensions, have been experimentally evaluated on GPUs in subsequent references [15], [16] using the JAX framework [13].

Although the aforementioned experimental results already demonstrate the advantage of temporal parallelization of Kalman filters and smoothers, the experimental results are still limited. The experiments were implemented using the relatively high-level JAX [13] and TensorFlow [14] frameworks, and the prefix-sum algorithms used were the default implementations provided by the frameworks. The results still leave it uncertain what the overhead of the frameworks themselves is and what the effect of the underlying prefix-sum algorithm is on the practical GPU performance. Also, the impact of the particular GPU choice remains unknown. In this paper, the aim is to fill these gaps by evaluating different prefix-sum algorithms on two different GPUs using a lower-level cross-platform implementation in Julia [17].

Another limitation in the reference [7] and the subsequent works is that they are based on the so-called Rauch–Tung–Striebel (RTS) formulation [2], [6] of the smoother. The RTS formulation has the disadvantage that the backward pass depends on the forward pass, and hence, they cannot be performed simultaneously, even when multiple GPUs are available. An alternative formulation of the smoother is the so-called two-filter smoother [3], which has independent forward and backward passes that can be run simultaneously on two GPUs. Although such two-filter smoothing in the context of hidden Markov models (HMMs) was discussed in [18] and an analogous parallel linear quadratic control algorithm was developed in [19], this kind of algorithm has not yet appeared in the literature. In this paper, we develop the parallel two-filter smoother and test its performance in a two-GPU setup.

As discussed above, the selection of the underlying prefix-sum algorithm also affects the performance of the parallel Kalman filter and smoother algorithms. Although, for example,

Simo Särkkä (simo.sarkka@aalto.fi) is with Dept. of Electrical Engineering and Automation, Aalto University, Finland, and Ángel F. García-Fernández (angel.garcia.fernandez@upm.es) is with IPTC, ETSI Telecomunicación, Universidad Politécnica de Madrid, Spain.

¹[https://github.com/EEA-sensors/sequential-parallelization-examples/tree/main/0000-0000/00\\$00.00-on-2 GPUs temporal-parallelization-bayes-smoothers](https://github.com/EEA-sensors/sequential-parallelization-examples/tree/main/0000-0000/00$00.00-on-2 GPUs temporal-parallelization-bayes-smoothers)

JAX and TensorFlow frameworks [13], [14] use the associative scan algorithm of Blelloch [9] for parallel prefix sum computation, there exist various alternative prefix sum methods. An early example is the Hillis–Steele algorithm [20], even earlier is the Ladner & Fischer circuit [21], [22], and various alternative and improved algorithms have been proposed [23]–[26]. Reviews of prefix-sum algorithms, historical remarks, and discussion on their performance in CPU and GPU contexts can be found in [26], [27]. The prefix-sum methods have various other applications, including sorting, recurrence equations, computer graphics, and optimal control [8], [9], [19], [20], [28], [29].

The contributions of the paper are:

- Analysis of the number of floating point operations required by Kalman filtering/smoothing methods with different prefix-sum algorithms.
- Experimental evaluation of Kalman filters and smoothers on two GPUs using dedicated Julia Metal.jl/CUDA.jl implementations with different all-prefix-sum algorithms.
- Two-filter smoother version of the parallel Kalman smoother and its evaluation using 2 GPUs.
- We also provide an open source code for the implementation of the algorithms.

The structure of the paper is the following. Section II reviews relevant parallel all-prefix sum algorithms. Section III reviews sequential Bayesian and Kalman filtering and smoothing algorithms. Section IV reviews the parallel versions of these algorithms and presents a novel two-filter form of the Bayesian and Kalman smoother. The implementations of the algorithms in Julia are explained in Section V. Experimental results are analyzed in Section VI. Finally, conclusions are drawn in Section VII.

II. PARALLEL ALL-PREFIX-SUM ALGORITHMS

This section reviews relevant parallel all-prefix sum algorithms required for the parallel implementation of filters and smoothers. The input size of the algorithms is denoted as T , which is also the number of measurements in the corresponding state estimation problem. All the algorithms presented assume that T is a power of 2, but they can be easily generalized to an arbitrary T by padding the series of elements with neutral elements.

A. Complexity measures of parallel programs

In the following sections, we use simplified forms of parallel random access machine (PRAM) models [30], where the memory access is assumed to take negligible (i.e., zero) computational time and a single application of the associative operator $\otimes$ in a single processor takes a single computational time unit. We use two different complexity measures. The first one is the *span complexity*, $\text{span}(T)$, which refers to the parallel computing steps taken by the program on a parallel computer with an unlimited number of processors. The second one is *work complexity*, $\text{work}(T)$, which is the total amount of computations taken by the algorithm. We assume the latter to be independent of the number of processors. We also always have $\text{span}(T) \leq \text{work}(T)$.

The actual computational time, $\text{time}(T, P)$, taken by the algorithm also depends on the number of processors P in the computer. We always have $\text{time}(T, P) \geq \text{span}(T)$ with the equality attained when $P \rightarrow \infty$. Therefore, with a large number of processors relative to T , the span complexity determines the computational time. However, we also have the work complexity bound $\text{time}(T, P) \geq \text{work}(T)/P$, and thus, with a small number of processors relative to T , we always end up following the work complexity. With growing T , the faster $\text{work}(T)$ grows with T , the faster the latter complexity regime is reached. A useful way to analyze the algorithms is to take $P = T$ and hence analyze how $\text{work}(T)/T$ grows. If it is in $O(1)$, then the algorithm scales quite well; otherwise, we can expect to run out of processors quite quickly.

B. Parallel computation of all-prefix-sums

All-prefix-sums for general operators can be used as computational primitives in various algorithms, including sorting, recurrence equations, and computer graphics, as discussed in [8], [20]. Additionally, they are applicable in state estimation and optimal control, as explored in [7], [15], [16], [18], [19]. In the all-prefix-sums problem, we are given a series of T elements $a_1, a_2, \dots, a_T$ and a binary associative operator $\otimes$ defined on the elements. The all-prefix-sum or scan operation computes $s_1 = a_1, s_2 = a_1 \otimes a_2, \dots, s_T = a_1 \otimes a_2 \otimes \dots \otimes a_T$. We can compute the all-prefix-sum operation sequentially in $O(T)$ time by using a simple loop. However, more importantly for this paper, it is also possible to compute it in parallel in $O(\log(T))$ span time with algorithms explained in the rest of this section. It is worth noting that the span complexity of $O(\log(T))$ only applies in the limit of an infinite number of processing cores, and the actual complexity on a fixed number of cores also heavily depends on the work complexity of the method at hand.

We also often need to compute the reversed all-prefix-sums $\bar{s}_T = a_T, \bar{s}_{T-1} = a_{T-1} \otimes a_T, \dots, \bar{s}_1 = a_1 \otimes a_2 \otimes \dots \otimes a_T$. Given a forward scan algorithm, we can compute these, for example, by reversing the series before and after applying the scan algorithm (reversion is a span $O(1)$ parallel operation). An alternative, often a more efficient implementation, involves reversing the indices inside the algorithm itself. The reversed prefix sums are needed in the computation of the Kalman smoothing solutions.

In the two-filter smoother that we introduce in Section IV-E, we combine the results of forward and reversed all-prefix sums. The idea is that, as we have $a_1 \otimes a_2 \otimes \dots \otimes a_T = (a_1 \otimes a_2 \otimes \dots \otimes a_{k-1}) \otimes (a_k \otimes a_{k+1} \otimes \dots \otimes a_T)$, then we also have $s_T = s_{k-1} \otimes \bar{s}_k$ for all k , which is useful in the two-filter smoother.

C. Hillis–Steele algorithm

The Hillis–Steele algorithm [20] is a simple parallel prefix sum algorithm that has $O(\log T)$ span complexity. The algorithm is based on recursively summing the first half of the series into the second half, as shown in Algorithm 1. Its disadvantage is the high work complexity of $O(T \log T)$, which thus causes the time complexity with a finite number of processors

to be super-linear. Hence $\text{work}(T)/T \in O(\log T)$, which hints that the algorithm runs out of processors quite quickly. We indeed see this problem later on in the experiments.

Algorithm 1 Parallel-scan algorithm of Hillis and Steele [20].

Input: The elements $\{a_k\}_{k=1}^T$ and associative operator $\otimes$.

Output: The result in $\{a_k\}_{k=1}^T$.

```

1: for  $d \leftarrow 0$  to  $\log_2 T$  do
2:    $b \leftarrow \text{copy}(a)$  // Take a copy of the current series
3:   for  $i \leftarrow 1$  to  $T - 2^d$  do {Compute in parallel}
4:      $j \leftarrow i + 2^d$ 
5:      $a_j \leftarrow b_i \otimes b_j$ 
6:   end for
7: end for

```

The operation of the Hillis–Steele [20] algorithm is illustrated in Fig. 1. The algorithm is in-place in the sense that, after computing the current row from the previous row, data from the earlier rows is no longer needed. However, in practice, we need to use double buffering (or a copy of the previous row, as in Alg. 1), because otherwise we would overwrite the data before using it in the computation. Thus, in that sense, the required storage is twice the length of the input array, $2T$.

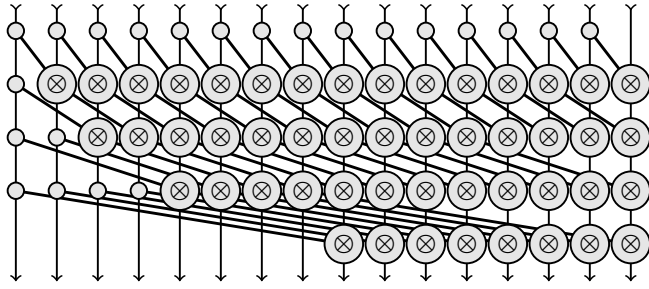


Fig. 1. Illustration of the operation of the algorithm of Hillis and Steele (Alg. 1) [20] with 16 elements. Each column is a data storage element with $\otimes$ being the associative operator between two elements. The arrows indicate increasing time. At the final time step, the data elements contain the prefix sums.

D. Blelloch's algorithm

The algorithm of Blelloch [9] is based on an up-sweep and a down-sweep, which can be seen as in-place implementations of upward and downward traversals in a binary tree. The algorithm is presented as Algorithm 2. The span complexity of the method is also $O(\log T)$, but its work complexity is $O(T)$, which is better than that of the Hillis–Steele algorithm. We indeed have $\text{work}(T)/T \in O(1)$, which suggests that the method runs out of processors more slowly than Hillis–Steele, and this phenomenon is also observed in the experiments. Thus, the algorithm is much faster in practice than Hillis–Steele. It is worth noting that the method requires the storage of length $2T$ – this is because we need additional storage for the copy of the original array (for the final pass). In fact, the temporary variables t need storage space as well, so in that sense the total storage space is actually $2T + T/2$.

Algorithm 2 Parallel-scan algorithm of Blelloch [9].

Input: The elements $\{a_k\}_{k=1}^T$ and associative operator $\otimes$.

Output: The result in $\{a_k\}_{k=1}^T$.

```

1:  $b \leftarrow \text{copy}(a)$  // Store the input
2: // Up-sweep:
3: for  $d \leftarrow 0$  to  $\log_2 T - 1$  do
4:   for  $i \leftarrow 0$  to  $T - 1$  by  $2^{d+1}$  do {Compute in parallel}
5:      $j \leftarrow i + 2^d$ 
6:      $k \leftarrow i + 2^{d+1}$ 
7:      $a_k \leftarrow a_j \otimes a_k$ 
8:   end for
9: end for
10:  $a_T \leftarrow 0$  {Here, 0 is the neutral element for  $\otimes$ }
11: // Down-sweep:
12: for  $d \leftarrow \log_2 T - 1$  to  $0$  do
13:   for  $i \leftarrow 0$  to  $T - 1$  by  $2^{d+1}$  do {Compute in parallel}
14:      $j \leftarrow i + 2^d$ 
15:      $k \leftarrow i + 2^{d+1}$ 
16:      $t \leftarrow a_j$ 
17:      $a_j \leftarrow a_k$ 
18:      $a_k \leftarrow a_k \otimes t$ 
19:   end for
20: end for
21: // Final pass to form the inclusive scan:
22: for  $i \leftarrow 1$  to  $T$  do {Compute in parallel}
23:    $a_i \leftarrow a_i \otimes b_i$ 
24: end for

```

The operation of Blelloch's algorithm [9] is illustrated in Fig. 2. In addition to the need for a temporary variable, a special feature of the algorithm is the step that involves only zeroing the last element in the array. Furthermore, because the algorithm initially computes the exclusive scan, the last step (final pass) consists of a fully parallel operation between the original array and the exclusive scan result.

E. Ladner's and Fischer's circuit

Already in 1980, Ladner and Fischer [21] published an approach to form logarithmic-depth circuits for computing prefix sums. In algorithmic terms, the basic idea [22] is first to compute the pairs $\hat{a}_i = a_{2i-1} \otimes a_{2i}$ and to compute the prefix sums of the resulting array recursively. An in-place implementation of the method is presented as Algorithm 3. Although this type of implementation is known to parallel computing experts, this particular algorithm is difficult to find in existing literature. Its advantage is that it is genuinely in-place, as no additional storage is required beyond the original array of length T . The algorithm has a span complexity of $O(\log T)$ and work complexity $O(T)$. As Blelloch's algorithm, this method also has $\text{work}(T)/T \in O(1)$; however, it always requires two parallel steps fewer than Blelloch's algorithm, which is also a practical advantage.

The operation of Algorithm 3 is illustrated in Fig. 3. Unlike Blelloch's algorithm, there is no zeroing of elements, and the method computes the inclusive scan directly.

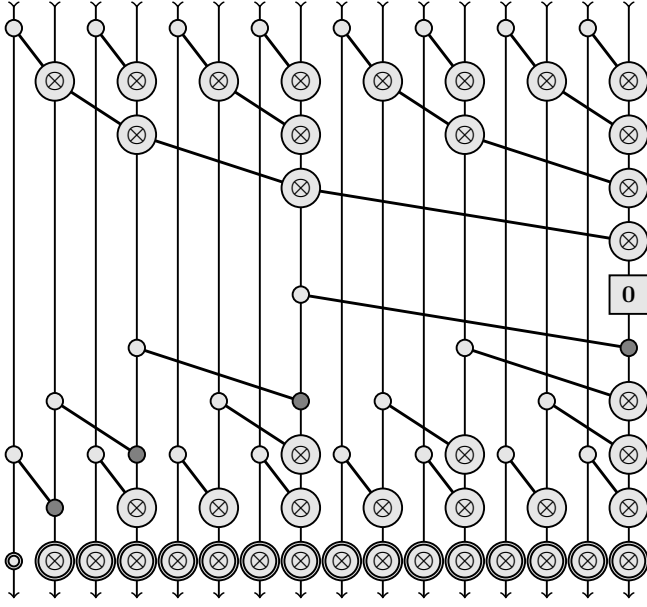


Fig. 2. Illustration of the operation of Blelloch's algorithm (Alg. 2) [9] with 16 elements.

Algorithm 3 In-place implementation of the parallel scan circuit of Ladner and Fischer [21].

Input: The elements $\{a_k\}_{k=1}^T$ and associative operator $\otimes$.

Output: The result in $\{a_k\}_{k=1}^T$.

```

1: // Up-sweep:
2: for  $d \leftarrow 0$  to  $\log_2 T - 1$  do
3:   for  $i \leftarrow 0$  to  $T - 1$  by  $2^{d+1}$  do {Compute in parallel}
4:      $j \leftarrow i + 2^d$ 
5:      $k \leftarrow i + 2^{d+1}$ 
6:      $a_k \leftarrow a_j \otimes a_k$ 
7:   end for
8: end for
9: // Down-sweep:
10: for  $d \leftarrow \log_2 T - 1$  to  $0$  do
11:   for  $i \leftarrow 2^{d+1}$  to  $T - 1$  by  $2^{d+1}$  do {In parallel}
12:      $j \leftarrow i + 2^d$ 
13:      $a_j \leftarrow a_i \otimes a_j$ 
14:   end for
15: end for

```

F. Hybrid algorithms of Sengupta et al.

The algorithm of Sengupta et al. [23] can be seen to employ a tree traversal similar to the Blelloch method, along with an additional fallback to Hillis–Steele for processing short series (say, $\leq N$) within the algorithm execution. The time complexity of the method is $O(\log T)$, and the work complexity is $O(T)$. Because the Hillis–Steele algorithm has a smaller absolute number of span steps, this can, in principle, make the algorithm faster. However, as will be seen in the experiments, this does not always happen in practice, due to higher work complexity. Therefore, a practical special case of the algorithm is also obtained by setting $N = 1$, in which case it becomes essentially equivalent to Algorithm 3 but with a different storage arrangement for the intermediate values.

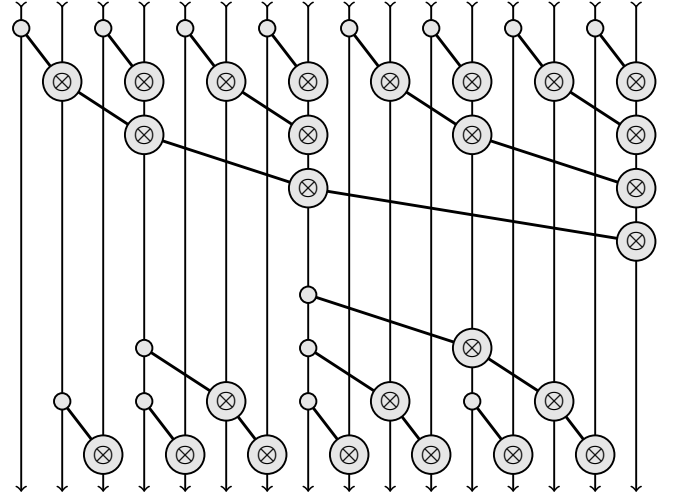


Fig. 3. Illustration of the operation of the in-place Ladner and Fischer's algorithm (Alg. 3) [21] with 16 elements.

Algorithm 4 Parallel-scan algorithm of Sengupta et al. [23].

Input: The elements $\{a_k\}_{k=1}^T$, associative operator $\otimes$, threshold parameter N .

Output: The result in $\{a_k\}_{k=1}^T$.

```

1:  $a^{(0)} \leftarrow \text{copy}(a)$  // The zeroth level is the plain series
2:  $d^* \leftarrow \log_2(T/N)$  // Up to which level we process
3: for  $d \leftarrow 1$  to  $d^*$  do
4:   // Reduce pass:
5:   for  $i \leftarrow 1$  to  $T/2^d$  do {Compute in parallel}
6:      $a_i^{(d)} \leftarrow a_{2i-1}^{(d-1)} \otimes a_{2i}^{(d-1)}$ 
7:   end for
8: end for
9: {Run Hillis–Steele algorithm on array  $a^{(d^*)}$ .}
10: // Down-sweep:
11: for  $d \leftarrow d^* - 1$  to  $0$  do
12:   for  $i \leftarrow 1$  to  $T/2^d$  do {Compute in parallel}
13:     if  $i > 1$  then
14:       if  $i \bmod 2 = 1$  then
15:          $a_i^{(d)} \leftarrow a_{(i-1)/2}^{(d+1)} \otimes a_i^{(d)}$ 
16:       else
17:          $a_i^{(d)} \leftarrow a_{i/2}^{(d+1)}$ 
18:       end if
19:     else
20:        $a_i^{(d)} \leftarrow a_i^{(d)}$ 
21:     end if
22:   end for
23: end for
24:  $a \leftarrow a^{(0)}$  // The final result

```

G. Other algorithms

There also exist various modifications to the methods described above, based on block processing and other ideas [24]–[26]. Most of these modifications, however, can be applied equally to all the algorithms above and therefore should not significantly alter their relative performance.

III. SEQUENTIAL BAYESIAN AND KALMAN FILTERING AND SMOOTHING

In this paper, the focus is on parallel implementations of Kalman filters and smoothers, which are special cases of Bayesian filters and smoothers (see, e.g., [6]). We now review these in a sequential setting. Bayesian filters and smoothers are sequential algorithms for estimating the state of a process $x_k \in \mathbb{R}^{n_x}$ at time step k from noisy measurements $y_k \in \mathbb{R}^{n_y}$. In particular, they aim at computing the conditional (posterior) distributions of the states given the measurements. The state evolves according to a Markov process with transition density $p(x_{k+1} | x_k)$. At each time step, the process is observed via a noisy measurement with density $p(y_k | x_k)$.

A. Bayesian filtering

In Bayesian filtering, we aim compute the posterior density $p(x_k | y_{1:k})$ of the state x_k given the measurements $y_{1:k} = (y_1, \dots, y_k)$ up to time step k . The filtering problem can be solved with linear complexity $O(T)$ using the Bayesian filtering recursion, which consists of the prediction and update steps given by [6], [31]

$$p(x_k | y_{1:k-1}) = \int p(x_k | x_{k-1}) p(x_{k-1} | y_{1:k-1}) dx_{k-1}, \quad (1)$$

$$p(x_k | y_{1:k}) = \frac{p(y_k | x_k) p(x_k | y_{1:k-1})}{\int p(y_k | x_k) p(x_k | y_{1:k-1}) dx_k}. \quad (2)$$

In practice, we start from a prior distribution $p(x_0) \triangleq p(x_0 | y_{1:0})$ and perform the above operations for $k = 1, 2, 3, \dots, T$, where T is the total number of measurements.

Kalman filter [1] is a special case of the above Bayesian filter, for linear Gaussian systems.

1) *Kalman filtering*: We consider the linear-Gaussian models of the form

$$p(x_k | x_{k-1}) = N(x_k; F_{k-1}x_{k-1} + u_{k-1}, Q_{k-1}), \quad (3)$$

$$p(y_k | x_k) = N(y_k; H_k x_k + d_k, R_k), \quad (4)$$

where $F_{k-1} \in \mathbb{R}^{n_x \times n_x}$ and $H_k \in \mathbb{R}^{n_y \times n_x}$ are known matrices, $u_{k-1} \in \mathbb{R}^{n_x}$ and $d_k \in \mathbb{R}^{n_y}$ are known vectors, and $Q_{k-1} \in \mathbb{R}^{n_x \times n_x}$ and $R_k \in \mathbb{R}^{n_y \times n_y}$ are covariance matrices. The prior at time step 0 is $p(x_0) = N(x_0; \bar{x}_{0|0}, P_{0|0})$.

In this case, the filtering density is Gaussian

$$p(x_k | y_{1:k}) = N(x_k; \bar{x}_{k|k}, P_{k|k}). \quad (5)$$

The filtering mean $\bar{x}_{k|k}$ and covariance matrix $P_{k|k}$ are calculated via the Kalman filtering recursion with $O(T)$ complexity. The prediction step computes

$$\bar{x}_{k|k-1} = F_{k-1}\bar{x}_{k-1|k-1} + u_{k-1}, \quad (6)$$

$$P_{k|k-1} = F_{k-1}P_{k-1|k-1}F_{k-1}^\top + Q_{k-1}. \quad (7)$$

The update step computes

$$S_k = H_k P_{k|k-1} H_k^\top + R_k, \quad (8)$$

$$\bar{x}_{k|k} = \bar{x}_{k|k-1} + P_{k|k-1} H_k^\top S_k^{-1} (y_k - H_k \bar{x}_{k|k-1} - d_k), \quad (9)$$

$$P_{k|k} = P_{k|k-1} - P_{k|k-1} H_k^\top S_k^{-1} H_k P_{k|k-1}. \quad (10)$$

The Kalman filter algorithm amounts to starting from the prior mean and covariance $\bar{x}_{0|0}, P_{0|0}$ and performing the above prediction and update steps for $k = 1, 2, 3, \dots, T$.

B. Smoothing

In smoothing, we compute the density $p(x_k | y_{1:T})$ for $k < T$, given the measurements up to a time step T . The smoothing problem can also be solved with linear complexity $O(T)$ by running an additional backward recursion, either in a forward-backward form, or via two-filter smoothing.

1) *Forward-backward smoother*: Given the filtering densities for $k = 1, \dots, T$, the forward-backward smoother uses the following recursion for $k = T-1, \dots, 1$ [6], [32]:

$$p(x_k | y_{1:T}) = p(x_k | y_{1:k}) \int \frac{p(x_{k+1} | x_k) p(x_{k+1} | y_{1:T})}{p(x_{k+1} | y_{1:k})} dx_{k+1}. \quad (11)$$

In the linear/Gaussian model, described by (3) and (4), the forward pass corresponds to the Kalman filter. The backward pass computes the smoothed mean $\bar{x}_{k|T}$ and covariance matrix $P_{k|T}$ via the following recursion from $k = T-1$ to $k = 1$:

$$G_k = P_{k|k} F_k^\top (P_{k+1|k})^{-1}, \quad (12)$$

$$\bar{x}_{k|T} = \bar{x}_{k|k} + G_k (\bar{x}_{k+1|T} - \bar{x}_{k+1|k}), \quad (13)$$

$$P_{k|T} = P_{k|k} + G_k (P_{k+1|T} - P_{k+1|k}) G_k^\top. \quad (14)$$

This smoothing algorithm is referred to as the Rauch–Tung–Striebel RTS smoother [2].

2) *Two-filter smoother*: The two-filter smoother is based on the factorization [33]

$$p(x_k | y_{1:T}) \propto p(x_k | y_{1:k}) p(y_{k+1:T} | x_k). \quad (15)$$

The two-filter smoother consists of two independent recursions, one running forward, which corresponds to the Bayesian filtering recursion in (1) and (2) to calculate the first factor, and one running backwards to calculate the second factor, given by

$$p(y_{k+1:T} | x_k) = \int p(x_{k+1} | x_k) p(y_{k+1:T} | x_{k+1}) dx_{k+1}, \quad (16)$$

$$p(y_{k:T} | x_k) = p(y_{k+1:T} | x_k) p(y_k | x_k). \quad (17)$$

For the linear Gaussian case, whose model is given by (3) and (4), the second factor in (15) is of the form

$$p(y_{k+1:T} | x_k) \propto N_I(x_k; \eta_{k|k+1:T}, J_{k|k+1:T}) \quad (18)$$

which represents a Gaussian density in information form with information vector $\eta_{k|k+1:T}$ and information matrix $J_{k|k+1:T}$, evaluated at x_k .

The backward filter recursion starts with [3]

$$\eta_{T|T:T} = H_T^\top R_T^{-1} (y_T - d_T), \quad (19)$$

$$J_{T|T:T} = H_T^\top R_T^{-1} H_T. \quad (20)$$

The backward prediction step computes [34]

$$\eta_{k-1|k:T} = F_{k-1}^\top (I + J_{k|k:T} Q_{k-1})^{-1} (\eta_{k|k:T} - J_{k|k:T} u_{k-1}) \quad (21)$$

$$J_{k-1|k:T} = F_{k-1}^\top (I + J_{k|k:T} Q_{k-1})^{-1} J_{k|k:T} F_{k-1}. \quad (22)$$

The backward prediction step can also be seen as the result of combining two filtering elements, which will be introduced in Section IV-C, when there is no measurement at time step $k - 1$.

The update step computes

$$\eta_{k-1|k-1:T} = \eta_{k-1|k:T} + H_{k-1}^\top R_{k-1}^{-1} (y_{k-1} - d_{k-1}), \quad (23)$$

$$J_{k-1|k-1:T} = J_{k-1|k:T} + H_{k-1}^\top R_{k-1}^{-1} H_{k-1}. \quad (24)$$

Using (15), the smoothed density is then Gaussian with mean and covariance matrix

$$\bar{x}_{k|T} = (I + P_{k|k} J_{k|k+1:T})^{-1} (\bar{x}_{k|k} + P_{k|k} \eta_{k|k+1:T}), \quad (25)$$

$$P_{k|T} = (I + P_{k|k} J_{k|k+1:T})^{-1} P_{k|k}. \quad (26)$$

In summary, for the linear Gaussian case, the two-filter smoother runs the Kalman filter forward, using (6)-(10), and, possibly in parallel, it runs the backward filter, using (21)-(24). The smoothed means and covariance matrices for all time steps are then obtained with (25) and (26).

IV. PARALLEL BAYESIAN AND KALMAN FILTERING AND SMOOTHING

In the previous section, we briefly reviewed sequential algorithms for Bayesian filtering and smoothing, as well as those for Kalman filtering and smoothing. In this section, the aim is to review parallel Bayesian/Kalman filtering methods and the RTS-type of smoothers first proposed in [7]. Finally, in Sections IV-E and IV-F, we describe a novel two-filter form of Bayesian/Kalman smoother.

A. Parallel Bayesian filtering

The parallel formulation of Bayesian smoothing in [7] is based on first computing the Bayesian filtering solution in parallel and then the Bayesian smoothing solution. In this section, we review the parallelization of the Bayesian filter. The parallelization is based on the use of parallel prefix-sum algorithms (see Sec. II), which require that the corresponding operator is associative. However, the operator corresponding to a single time step in the Bayesian filtering equations (1) and (2) is not associative. It can be made associative as follows.

An element used for filtering $a_k \in \mathcal{F}$ is of the form $a_k = (f, g)$ where f is a two-variable function $f: \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \rightarrow \mathbb{R}$ and g is a one-variable function $g: \mathbb{R}^{n_x} \rightarrow \mathbb{R}$.

Definition 1: Given two elements $(f_i, g_i) \in \mathcal{F}$ and $(f_j, g_j) \in \mathcal{F}$, the associative operator $\otimes$ for Bayesian filtering is [7]

$$(f_i, g_i) \otimes (f_j, g_j) = (f_{i,j}, g_{i,j}), \quad (27)$$

where

$$f_{i,j}(x, z) = \frac{\int g_j(y) f_j(x, y) f_i(y, z) dy}{\int g_j(y) f_i(y, z) dy}, \quad (28)$$

$$g_{i,j}(z) = g_i(z) \int g_j(y) f_i(y, z) dy.$$

Specifically, for $k > 1$ the element a_k is [7]

$$a_k = \left(\frac{p(x_k | y_k, x_{k-1})}{p(y_k | x_{k-1})} \right), \quad (29)$$

where $f_k(x_k, x_{k-1}) = p(x_k | y_k, x_{k-1})$ and $g_k(x_{k-1}) = p(y_k | x_{k-1})$. Furthermore, we have

$$a_1 = \left(\frac{p(x_1 | y_1)}{p(y_1)} \right). \quad (30)$$

These definitions of element and associative operator yield

$$a_1 \otimes a_2 \otimes \dots \otimes a_k = \left(\frac{p(x_k | y_{1:k})}{p(y_{1:k})} \right). \quad (31)$$

Let us understand this combination rule in more detail. Let i, j , and k be three time steps such that $i < j < k$, and $a_{i,j} = a_i \otimes a_{i+1} \otimes \dots \otimes a_j$. Then, we have that

$$a_{i,j} = \left(\frac{p(x_j | y_{i+1:j}, x_i)}{p(y_{i+1:j} | x_i)} \right), \quad (32)$$

$$a_{j,k} = \left(\frac{p(x_k | y_{j+1:k}, x_j)}{p(y_{j+1:k} | x_j)} \right). \quad (33)$$

The combination rule therefore computes $a_{i,k} = a_{i,j} \otimes a_{j,k}$ using the following relations, which can be obtained from the Markov properties of the model,

$$p(x_k | y_{i+1:j}, x_j) \propto \int p(y_{j+1:k} | x_j) p(x_k | y_{j+1:k}, x_j), \quad (34)$$

$$\times p(x_j | y_{i+1:j}, x_i) dx_j$$

$$p(y_{i+1:k} | x_i) = p(y_{i+1:j} | x_i) \times \int p(y_{j+1:k} | x_j) p(x_j | y_{i+1:j}, x_i) dx_j. \quad (35)$$

Computing $a_1 \otimes a_2 \otimes \dots \otimes a_k$ sequentially is equivalent to computing the filtering solution using equations (1) and (2), and additionally, the normalization constant $p(y_{1:k})$ using a separate recursion. However, the operator is associative, and hence we can parallelize it using parallel prefix sums.

B. Parallel Bayesian smoothing

The parallel smoother derived in [7] has the so-called Rauch–Tung–Striebel form corresponding to parallelization of the equation (11). In this formulation, the element used for smoothing $a_k \in \mathcal{S}$ is a two-variable function of the form $a: \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \rightarrow \mathbb{R}$.

Definition 2: Given two elements $a_i \in \mathcal{S}$ and $a_j \in \mathcal{S}$, the associative operator $\otimes$ for Bayesian smoothing is

$$a_i \otimes a_j = a_{i,j},$$

where

$$a_{i,j}(x, z) = \int a_i(x, y) a_j(y, z) dy.$$

In particular, the element that enables parallel computation of Bayesian smoothing is [7]

$$a_k = p(x_k \mid y_{1:k}, x_{k+1}) \quad (36)$$

with $a_T = p(x_T \mid y_{1:T})$.

These definitions of element and associative operator lead to

$$a_k \otimes a_{k+1} \otimes \cdots \otimes a_T = p(x_k \mid y_{1:T}). \quad (37)$$

As in filtering, let us understand this combination rule in more detail. Let i, j , and k be three time steps such that $i < j < k$, the elements are then

$$a_{i,j} = p(x_i \mid y_{1:j-1}, x_j), \quad (38)$$

$$a_{j,k} = p(x_j \mid y_{1:k-1}, x_k). \quad (39)$$

The associative operator combines these results by computing

$$a_{i,k} = \int p(x_i \mid y_{1:j-1}, x_j) p(x_j \mid y_{1:k-1}, x_k) dx_j \quad (40)$$

$$= p(x_i \mid y_{1:k-1}, x_k). \quad (41)$$

This result is direct from the Markov properties of the model.

C. Parallel Kalman filter

Let us consider the linear-Gaussian system, with model in (3) and (4). To solve the associated filtering problem with $O(\log T)$ complexity, the element a_k in (29) is described by

$$\begin{aligned} p(x_k \mid y_k, x_{k-1}) &= N(x_k; A_k x_{k-1} + b_k, C_k), \\ p(y_k \mid x_{k-1}) &\propto N_I(x_{k-1}; \eta_k, J_k). \end{aligned} \quad (42)$$

The parameters $(A_k, b_k, C_k, \eta_k, J_k)$ are given as follows [7]. For $k > 1$, (A_k, b_k, C_k) are

$$\begin{aligned} S_k &= H_k Q_{k-1} H_k^\top + R_k, \\ K_k &= Q_{k-1} H_k^\top S_k^{-1}, \\ A_k &= (I_{n_x} - K_k H_k) F_{k-1}, \\ b_k &= u_{k-1} + K_k (y_k - H_k u_{k-1} - d_k), \\ C_k &= (I_{n_x} - K_k H_k) Q_{k-1}. \end{aligned} \quad (43)$$

For $k = 1$, (A_1, b_1, C_1) are

$$\begin{aligned} \bar{x}_{1|0} &= F_0 \bar{x}_{0|0} + u_0, \\ P_{1|0} &= F_0 P_{0|0} F_0^\top + Q_0, \\ S_1 &= H_1 P_{1|0} H_1^\top + R_1, \\ K_1 &= P_{1|0} H_1^\top S_1^{-1}, \\ A_1 &= 0, \\ b_1 &= \bar{x}_{1|0} + K_1 [y_1 - H_1 \bar{x}_{1|0} - d_1], \\ C_1 &= P_{1|0} - K_1 S_1 K_1^\top. \end{aligned} \quad (44)$$

The parameters (η_k, J_k) of the second term are given as

$$\begin{aligned} \eta_k &= F_{k-1}^\top H_k^\top S_k^{-1} (y_k - H_k u_{k-1} - d_k), \\ J_k &= F_{k-1}^\top H_k^\top S_k^{-1} H_k F_{k-1}, \end{aligned} \quad (45)$$

for $k = 1, \dots, T$.

How the associative operator for filtering works when the elements are defined in this manner is provided in the following lemma.

Lemma 3: Given two elements $(A_i, b_i, C_i, \eta_i, J_i)$ and $(A_j, b_j, C_j, \eta_j, J_j)$ the binary operator $\otimes$ for filtering returns an element $(A_{i,j}, b_{i,j}, C_{i,j}, \eta_{i,j}, J_{i,j})$ where

$$\begin{aligned} A_{i,j} &= A_j (I_{n_x} + C_i J_j)^{-1} A_i, \\ b_{i,j} &= A_j (I_{n_x} + C_i J_j)^{-1} (b_i + C_i \eta_j) + b_j, \\ C_{i,j} &= A_j (I_{n_x} + C_i J_j)^{-1} C_i A_j^\top + C_j, \\ \eta_{i,j} &= A_i^\top (I_{n_x} + J_j C_i)^{-1} (\eta_j - J_j b_i) + \eta_i, \\ J_{i,j} &= A_i^\top (I_{n_x} + J_j C_i)^{-1} J_j A_i + J_i. \end{aligned} \quad (46)$$

The pseudocode of the parallel Kalman filter is provided in Algorithm 5.

Algorithm 5 Parallel Kalman filter from [7].

Input: The measurements $y_{1:T}$, the model matrices $F_{0:T-1}$, $Q_{0:T-1}$, $H_{1:T}$, $R_{1:T}$, inputs $u_{0:T-1}$, and initial statistics $\bar{x}_{0|0}, P_{0|0}$.

Output: The Kalman filter means and covariances $\bar{x}_{k|k}$ and $P_{k|k}$ for $k = 1, \dots, T$.

```

1: // Initialization:
2: for  $k \leftarrow 1$  to  $T$  do {Compute in parallel}
3:   Compute initial element
       $a_k \leftarrow (A_k, b_k, C_k, \eta_k, J_k)$ 
      using (43), (44), and (45).
4: end for
5: // Associative scan:
6: Define filtering operator  $\otimes$  via Lemma 3.
7: Call parallel associative scan to obtain the prefix sums:
       $s_{1:T} = \text{parallel\_scan}(a_{1:T}, \otimes)$ .
8: // Extract the results:
9: for  $i \leftarrow 1$  to  $T$  do {Compute in parallel}
10:   Unpack the prefix sums:  $(A_k^+, b_k^+, C_k^+, \eta_k^+, J_k^+) = s_k$ .
11:   Extract the Kalman filter mean and covariance:
       $\bar{x}_{k|k} \leftarrow b_k^+$ ,
       $P_{k|k} \leftarrow C_k^+$ .
12: end for
```

D. Parallel Rauch–Tung–Striebel smoother

Let us now formulate the Rauch–Tung–Striebel (RTS) smoother [2] in a parallel form. For the linear-Gaussian model in (3) and (4), the element for forward-backward smoothing becomes

$$p(x_k \mid y_{1:k}, x_{k+1}) = \mathcal{N}(x_k; E_k x_{k+1} + g_k, L_k), \quad (48)$$

where the parameters (E_k, g_k, L_k) are given as follows. For $k < T$

$$\begin{aligned} E_k &= P_{k|k} F_k^\top (F_k P_{k|k} F_k^\top + Q_k)^{-1}, \\ g_k &= \bar{x}_{k|k} - E_k (F_k \bar{x}_{k|k} + u_k), \\ L_k &= P_{k|k} - E_k F_k P_{k|k}, \end{aligned} \quad (49)$$

and, for $k = T$,

$$\begin{aligned} E_T &= 0, \\ g_T &= \bar{x}_{T|T}, \\ L_T &= P_{T|T}. \end{aligned} \quad (50)$$

The following lemma describes how the associative operator for smoothing works when the elements are defined as above.

Lemma 4: Given two elements (E_i, g_i, L_i) and (E_j, g_j, L_j) the associative operator $\otimes$ for smoothing returns an element $(E_{i,j}, g_{i,j}, L_{i,j})$ where

$$\begin{aligned} E_{i,j} &= E_i E_j, \\ g_{i,j} &= E_i g_j + g_i, \\ L_{i,j} &= E_i L_j E_i^\top + L_i. \end{aligned} \quad (51)$$

The pseudocode of the parallel RTS smoother is provided in Algorithm 6.

Algorithm 6 Parallel RTS smoother from [7].

Input: The Kalman filter results $\bar{x}_{k|k}$ and $P_{k|k}$ for $k = 1, \dots, T$, the model matrices $F_{0:T-1}$, $Q_{0:T-1}$, and inputs $u_{0:T-1}$.

Output: The RTS smoother means and covariances $\bar{x}_{k|T}$ and $P_{k|T}$ for $k = 1, \dots, T$.

- 1: // Initialization:
- 2: **for** $k \leftarrow 1$ **to** T **do** {Compute in parallel}
- 3: Compute initial element
- $a_k \leftarrow (E_k, g_k, L_k)$
- using (49) and (50).
- 4: **end for**
- 5: // Associative scan:
- 6: Define smoothing operator $\otimes$ via Lemma 4.
- 7: Call backward parallel associative scan to obtain the prefix sums:

$$\bar{s}_{1:T} = \text{reverse_parallel_scan}(a_{1:T}, \otimes).$$

- 8: // Extract the results:
- 9: **for** $i \leftarrow 1$ **to** T **do** {Compute in parallel}
- 10: Unpack the prefix sums: $(E_k^+, g_k^+, L_k^+) = \bar{s}_k$.
- 11: Extract the RTS smoother mean and covariance:

$$\begin{aligned} \bar{x}_{k|T} &\leftarrow g_k^+, \\ P_{k|T} &\leftarrow L_k^+. \end{aligned} \quad (52)$$

- 12: **end for**
-

E. Parallel two-filter smoother

The novel two-filter smoother that we propose in this paper is based on the factorization of the smoothed density in (15). The first factor in (15), $p(x_k | y_{1:k})$, can be computed for all k using parallel computation using (31). In addition, using the filtering element in (29) and its associative operator, it is direct to check that

$$a_{k+1} \otimes \dots \otimes a_T = \begin{pmatrix} p(x_T | y_{k+1:T}, x_k) \\ p(y_{k+1:T} | x_k) \end{pmatrix}. \quad (53)$$

Therefore, the second factor in (15), $p(y_{k+1:T} | x_k)$, can be computed for all k using the same elements and associative operator as for filtering, but using a reversed all-prefix-sums operation. In practice, to align the elements of the forward and backward filters better, it is sometimes beneficial to use the equivalent formulation

$$a_{k+1} \otimes \dots \otimes a_T \otimes e = \begin{pmatrix} p(x_T | y_{k+1:T}, x_k) \\ p(y_{k+1:T} | x_k) \end{pmatrix}, \quad (54)$$

where e is the neural element for $\otimes$.

Once we have run the forward and backward filters in parallel, which can be executed on two different GPUs, we can combine their results to obtain the smoothed density using (15). This combination can be done in parallel for all time steps. When operations are performed on two different GPUs, the initialization must also be performed twice to avoid sending the initialization results from one GPU to another. This is also what we do in our implementation.

F. Parallel two-filter linear-Gaussian smoother

This section explains the implementation of the parallel two-filter smoother in Section IV-E for the linear-Gaussian models in (3) and (4). In this case, we run the parallel Kalman filter using the elements and associative operator in Section IV-C, collecting the filtering mean $\bar{x}_{k|k}$ and covariance matrix $P_{k|k}$ for all k .

In parallel to this operation (for instance, using a second GPU), we run a reverse all-prefix-sums operation, using the same elements and associative operator, collecting the backward information vector $\eta_{k|k+1:T}$ and information matrix $J_{k|k+1:T}$ for all k . Finally, we can obtain the smoothed mean $\bar{x}_{k|T}$ and covariance matrix $P_{k|T}$ for all k in parallel using (25) and (26). The pseudocode of the parallel two-filter smoother is provided in Algorithm 7.

V. IMPLEMENTATION OF THE ALGORITHMS

In this section, the aim is to discuss how the algorithms described in the previous sections have been implemented for the experimental evaluation. All the implementations are available in (link to be revealed upon acceptance).²

A. Julia's Metal.jl and CUDA.jl

Our implementations have been done purely in the Julia programming language [17] and, in particular, using Julia's GPU compiler interface packages [35], [36] for Apple Metal (Metal.jl) [37] and NVIDIA's CUDA (CUDA.jl) [38]. We implemented the core matrix routines required by the Kalman filters and smoothers in platform-independent Julia, which is compiled for the GPUs using these packages or for CPU using the standard (just-in-time) compilation features of Julia.

The program listings in Figs. 4 and 5 illustrate the principle of how the code is implemented. The code in Fig. 4 is an illustrative platform-independent implementation of the d th level of the up-sweep, which appears in Algorithms 2 and

²A zip file containing the implementations is provided as a supplementary file for the peer review.

Algorithm 7 Parallel two-filter smoother, including the Kalman filter pass.

Input: The measurements $y_{1:T}$, the model matrices $F_{0:T-1}$, $Q_{0:T-1}$, $H_{1:T}$, $R_{1:T}$, inputs $u_{0:T-1}$, and initial statistics $\bar{x}_{0|0}$, $P_{0|0}$.

Output: The smoother means and covariances $\bar{x}_{k|T}$ and $P_{k|T}$ for $k = 1, \dots, T$.

- 1: // Initialization:
- 2: **for** $k \leftarrow 1$ **to** T **do** {Compute in parallel}
- 3: Compute initial element

$$a_k \leftarrow (A_k, b_k, C_k, \eta_k, J_k)$$

using (43), (44), and (45).

- 4: **end for**
- 5: // Associative scan:
- 6: Define filtering operator $\otimes$ via Lemma 3.
- 7: Call parallel associative scan to obtain the prefix sums:

$$s_{1:T} = \text{parallel_scan}(a_{1:T}, \otimes).$$

- 8: Call reverse parallel associative scan to obtain the reversed prefix sums (this can be done in parallel with the above):

$$\bar{s}_{1:T} = \text{reverse_parallel_scan}((a_{2:T}, e), \otimes),$$

where $e = (0, 0, 0, 0, 0)$.

- 9: // Extract the results:
- 10: **for** $i \leftarrow 1$ **to** T **do** {Compute in parallel}
- 11: Unpack the prefix sums:

$$\begin{aligned} (A_k^+, b_k^+, C_k^+, \eta_k^+, J_k^+) &\leftarrow s_k. \\ (A_k^-, b_k^-, C_k^-, \eta_k^-, J_k^-) &\leftarrow \bar{s}_k. \end{aligned} \quad (55)$$

- 12: Extract the forward and backward Kalman filter results and compute the smoothing solution:

$$\begin{aligned} \bar{x}_{k|k} &\leftarrow b_k^+, \\ P_{k|k} &\leftarrow C_k^+, \\ \eta_{k|k+1:T} &\leftarrow \eta_k^-, \\ J_{k|k+1:T} &\leftarrow J_k^-, \\ \bar{x}_{k|T} &\leftarrow (I + P_{k|k} J_{k|k+1:T})^{-1} (\bar{x}_{k|k} + P_{k|k} \eta_{k|k+1:T}) \\ P_{k|T} &\leftarrow (I + P_{k|k} J_{k|k+1:T})^{-1} P_{k|k}. \end{aligned} \quad (56)$$

- 13: **end for**

3 – the actual Julia code is slightly more complicated to avoid copying of values around, but the principle is the same. The platform-dependent part is given as a closure `index_stride_f` which returns the index and stride for the stride loop. The associative operator $\otimes$ is provided as a closure `op`. The elements themselves are given in array `elems`, the level in d , and the total number of elements in T .

Fig. 5 shows example implementations of the index and stride computation for Metal and CUDA. A similar implementation can also be done for the CPU. It is worth noting that the Julia GPU Compiler [35], [36] itself supports similar platform-abstractions over GPUs, but we have chosen to use

```
function upsweep_kernel(index_stride_f, op, elems, d, T)
    index, stride = index_stride_f()

    delta1 = 1 << d
    delta2 = 1 << (d+1)
    range_j = ((index - 1) * delta2 + delta1):
              (stride * delta2):(T - 1 + delta1)
    range_k = ((index - 1) * delta2 + delta2):
              (stride * delta2):(T - 1 + delta2)

    for (j,k) in zip(range_j, range_k)
        elems[k] = op(elems[j], elems[k])
    end
    return
end
```

Fig. 4. Sketch of Julia implementation of generic up-sweep GPU kernel of a parallel associative scan.

this one to allow for more flexible benchmarking with CPU.

```
function index_stride_f_metal()
    index = thread_position_in_grid_ld()
    stride = threads_per_grid_ld()
end
```

```
function index_stride_f_cuda()
    index = (blockIdx().x-1) * blockDim().x + threadIdx().x
    stride = gridDim().x * blockDim().x
end
```

Fig. 5. Functions to compute the index and stride on Metal and CUDA.

B. Implementation of the parallel prefix-sum algorithms

All the parallel prefix-sum (i.e., scan) algorithms have been implemented as platform-independent codes resembling the one shown in Fig. 4. One practical difference, though, is that to avoid dynamic memory allocation, the operator directly stores the result in the array instead of using the assignment operator `=` for it. Hence, the associative operation is implemented in the form `fun!(i, j, k, elems)`, which performs the operation `elems[i] = op(elems[j], elems[k])`. More precisely, `elems` is actually a tuple of arrays in order to support the associative operations required in the Kalman filters and smoothers.

C. Implementation of parallel Kalman filters and smoothers

For the Kalman filters and smoothers, we implemented a custom set of primitive matrix operations (addition, multiplication, Cholesky, LU, triangular solving, etc.) that can be run in the GPU kernels. The parallel Kalman filter and smoothers were implemented using these primitives with the help of the associative scan algorithms described in the previous section.

D. Calculating the theoretical number of operations

To count the floating point operations required by the algorithms, we implemented flop-counting on top of the matrix routines using Julia's operator overloading, and made a simulated index and stride interface for it (as in Fig. 5), which counts the flops for a given number of simulated threads. This allowed us to count the number of operations using exactly the same platform-independent code as was run on the GPUs.

VI. EXPERIMENTAL RESULTS

In this section, we report the experimental results consisting of counting the number of floating-point operations for simulated runs of the methods, and the algorithm speed measurements using Apple Metal GPU and NVIDIA CUDA GPU, along with the CPUs of the computers.

A. Experimental setup

The performance was measured by using a model of the form given in Eqs. (3) and (4) with 4-dimensional state and 2-dimensional measurements such that F_k , u_k , Q_k , H_k , d_k , and R_k were randomly generated for each k separately. They were generated from multivariate unit Gaussian distributions, but the F_k matrix was replaced with 0.99 times its Q factor from QR-factorization to ensure numerical stability, and Q_k and R_k were formed as products of the random matrices $\Xi \Xi^T$, where Ξ is a Gaussian matrix, to ensure that they are positive definite. The initial mean m_1 and covariance P_1 were similarly randomly generated. The measurements were generated by simulating data from the model.

In the GPU and CPU speed experiments, the time was measured with Julia's `@elapsed` macro, and the time was computed from 12 runs, where the first two were discarded (to eliminate the effect of JIT compilation), and the time was estimated to be the median of the rest of the runs. All the computations were performed using 32-bit floating-point numbers.

The experiments were run on two computers:

- **Metal computer:** Apple MacBook Pro laptop with Apple M3 Max GPU (30 Cores) and with Apple M3 Max GPU at 2400 MHz.
- **CUDA computer:** Desktop computer with 4 NVIDIA A100-SXM4-80GB GPUs and AMD EPYC 7643 48-Core Processor at 1500 MHz.

The associative scan methods were the following:

- **Hillis–Steele:** This is the Hillis–Steele parallel associative scan in Algorithm 1.
- **Blelloch:** This is the Blelloch's parallel associative scan in Algorithm 2.
- **Inplace LaFi:** This is the inplace implementation of the Ladner–Fischer circuit given in Algorithm 3.
- **Sengupta A:** This is the Sengupta's parallel associative scan in Algorithm 4 with $N = 1$.
- **Sengupta B:** This is the Sengupta's parallel associative scan in Algorithm 4 with N set of 14750 in the Metal experiments, 20000 in CUDA experiments, and 15000 in simulated hardware experiments. The Metal and CUDA N 's were empirically selected to give good performance.

The parallel Kalman filtering and smoothing methods are the following:

- **PKF:** This is the parallel Kalman filter described in Algorithm 5.
- **PRTS:** This is the parallel Rauch–Tung–Striebel smoother (including the parallel Kalman filter pass) described in Algorithm 6.
- **PTFS:** This is the parallel two-filter smoother (including the parallel Kalman filter pass) described in Algorithm 7.

B. Performance on a simulated parallel hardware

As described in Section V-D, we implemented floating point operation counting on top of the methods, which can be used to run the codes with a simulated number of threads and estimate the time taken by the methods (“time” in terms of number of floating point operations). In the experiments, we simulated a system with a GPU with 15000 threads, which is slightly more than the number of threads supported by the Metal computer's GPU and somewhat less than the CUDA computer's GPU.

Figure 6 shows the estimated time (in terms of number of operations) as a function of the number of time steps for the tested method, associative scan, and Kalman filter/smoothing combinations. Figure 7 shows more detailed results of runs with $T = 10^3, 10^4, 10^5, 10^6$. It can be seen that with a small number of time steps T , the performance of the Hillis–Steele and Sengupta B (which incorporates Hillis–Steele) is the best. In particular, in Figure 7, we can see that they are the fastest associative scan methods for $T = 10^3, 10^4$. However, at larger numbers of time steps ($T = 10^5, 10^6$ in Fig. 7), the Hillis–Steele method is significantly slower than the other methods. At the larger numbers of time steps, the Inplace LaFi and Sengupta A appear to be slightly faster than the other associative scan methods.

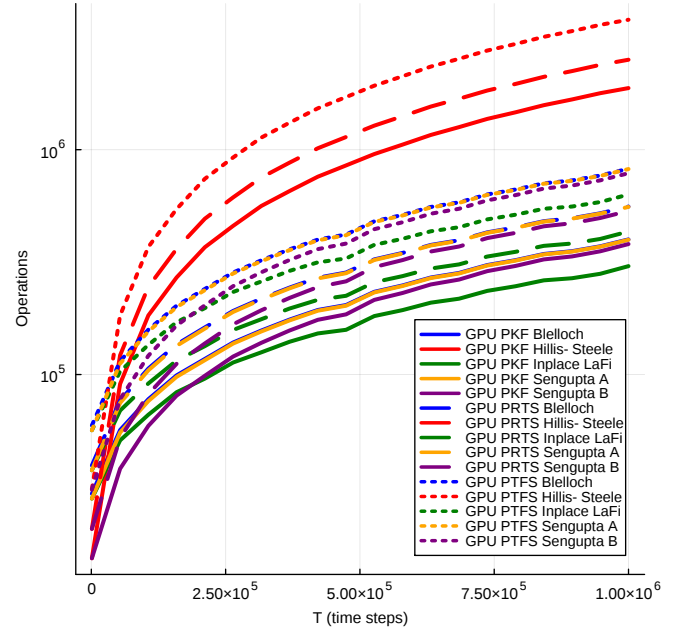


Fig. 6. Number of operations of the different algorithm combinations as a function of time series length T .

Among the Kalman filtering and smoothing methods, the PKF appears to be the fastest systematically, followed by PRTS, and then PTFS. The good performance of PKF is, though, expected because PRTS and PTFS perform the same operations as PKF, along with additional ones.

C. Speeds of different algorithms on Metal GPU

We ran the same code that was used to implement the simulated hardware experiment in the previous section (Sec. VI-B),

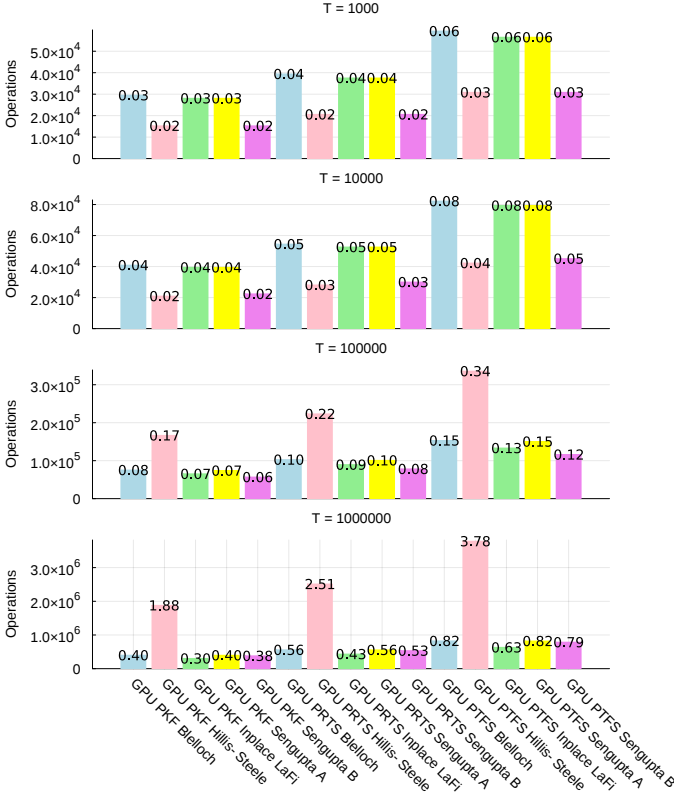


Fig. 7. Number of operations of the different algorithm combinations with time series lengths $T = 10^3, 10^4, 10^5, 10^6$.

on the GPU of the Metal computer. Figure 8 shows the run times as a function of the time series length T and Figure 9 shows the times for time series lengths $T = 10^3, 10^4, 10^5, 10^6$. With short time series lengths ($10^3, 10^4$), in the PKF case, all the associative scan methods perform similarly. However, in the PRTS and PTFS cases, Billeloch's method and Sengupta B seem to be slightly slower than the other methods. With the longer time series lengths ($10^5, 10^6$), this effect disappears and all the methods except Hillis–Steele perform similarly, Hillis–Steele being significantly slower than the other methods. We can see that the Inplace LaFi method performs slightly better than the other methods with the longer time series lengths. We can see that the PKF method is always the fastest, PRTS the second fastest, and PTFS the third, as could be expected.

By comparing the simulated hardware results in Figure 7 and the Metal results in Figure 9, we can observe that with the larger time series lengths, the simulation is a very good predictor for the actual time taken by the methods on the Metal GPU. However, with the shorter time series lengths, the simulation predicted Hillis–Steele and Sengupta B to be much better than they were on the actual hardware.

D. Speeds of different algorithms on CUDA GPU

We also ran the codes used in the simulated hardware experiment in Section VI-B and the Metal experiment in Section VI-C on the GPU of the CUDA computer. The results are shown in Figures 10 and 11. With the shorter time series lengths ($10^3, 10^4$), we observe that Billeloch's method and

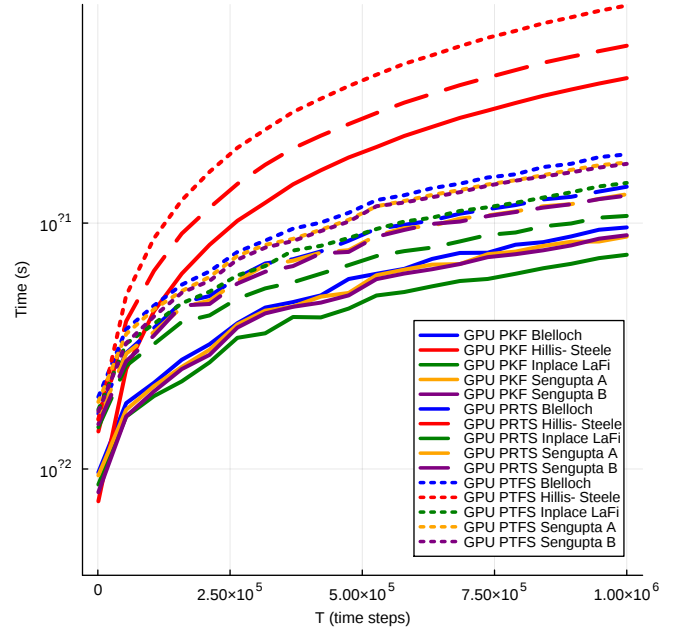


Fig. 8. Run times of the different algorithm combinations on the Metal computer as a function of time series length T .

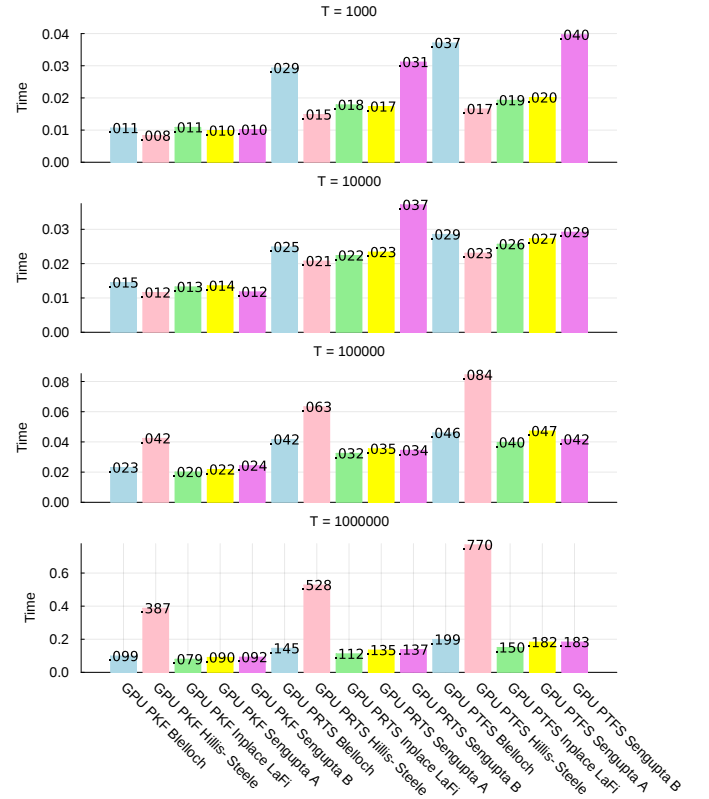


Fig. 9. Run times of the different algorithm combinations on the Metal computer with time series lengths $T = 10^3, 10^4, 10^5, 10^6$.

Inplace LaFi are slightly slower than the other methods, whereas Hillis–Steele, Sengupta A, and Sengupta B perform well. With the longer time series lengths, the Hillis–Steele method starts to perform worse, and the differences between the other methods even out somewhat. However, Sengupta A and Sengupta B still perform the best. Again, the PKF methods are the fastest, then PRTS, and PTFS is the third fastest.

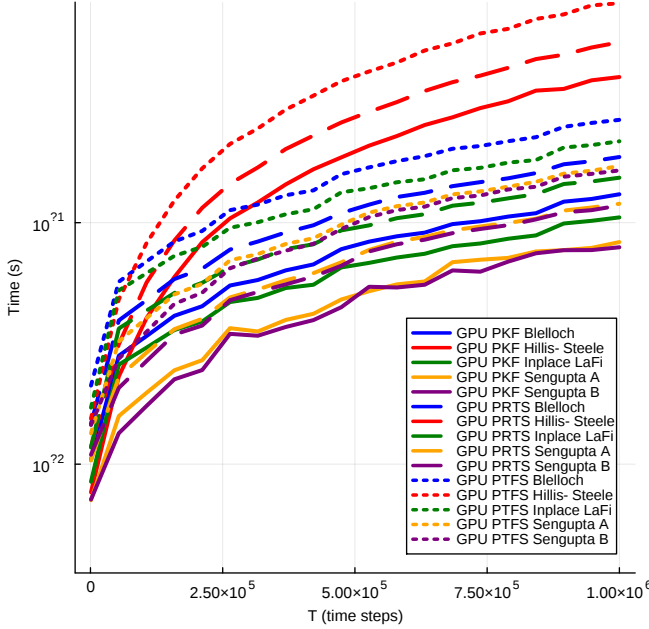


Fig. 10. Run times of the different algorithm combinations on the CUDA computer as a function of time series length T .

When comparing to the simulated results in Figure 7, we can see that the simulated results are again quite good predictors of the actual CUDA GPU behavior with the longer time series lengths, though not as good as in the case of the Metal GPU in Figure 9.

When comparing the Metal GPU results in Figure 9 to the CUDA results in Figure 11, we can see that the overall behavior of the GPUs is very similar. There are differences, though, which can be caused by the hardware differences or the differences in the code generated by the Julia GPU Compiler. However, it is surprising to see how similar the overall performance of the GPUs is. The codes that we run on both of the GPUs are exactly the same, and the computational times are both measured in the same way, which implies that the computational capabilities of the GPUs are very similar.

E. Evaluation of PTFS on two GPUs

In the previous sections, we have been running the algorithms on a single GPU. The results have also shown that in this setting, the PTFS algorithm which we proposed in Section IV-E is actually slower than the previously proposed PRTS [7]. However, as discussed in the same section, the forward and backward passes are independent and can be run on two GPUs in parallel.

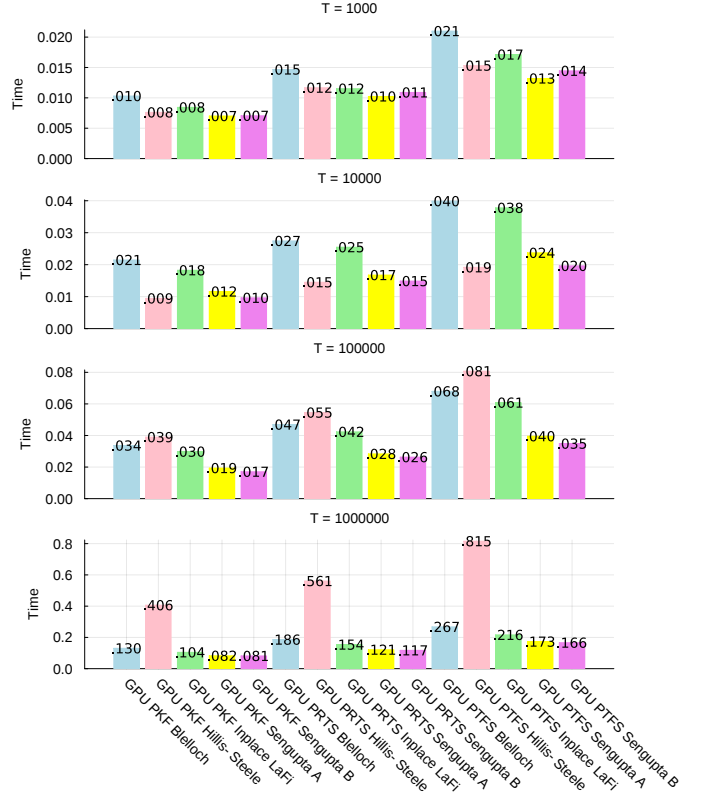


Fig. 11. Run times of the different algorithm combinations on the CUDA computer with time series lengths $T = 10^3, 10^4, 10^5, 10^6$.

We implemented the PTFS algorithm described in Section IV-E on two CUDA GPUs as follows:

- 1) Run the backward pass of the PTFS on GPU0 and simultaneously the forward pass of the PTFS on GPU1.
- 2) Copy the results of the forward pass first to CPU, and from there to GPU0.
- 3) Then on GPU0 compute the combination step.

The results of the experiment where we ran PRTS and PTFS methods on a single GPU and PTFS on two GPUs using the above procedure are shown in Figures 12 and 13. The results now show that the 2-GPU version of the PTFS is significantly faster than either of the 1-GPU algorithms. Note that we have left out PKF because it is precisely the forward pass of PTFS, and we have also left out Hillis–Steele in order to see the differences of the methods more clearly. Among these tested methods, the 2-GPU PTFS with the Sengupta B associative scan appears to perform the best.

F. GPU speedup analysis

So far, we have only investigated the relative performance of the associative scan and Kalman filter/smoothing combinations without computing the actual speedup that the methods provide. To compute this, we performed an experiment in which we ran the classical sequential Kalman filter [1] (the optimal sequential algorithm) on a single core of a GPU and compared its speed to that of the parallel Kalman filter (PKF) [7]. We used the Inplace LaFi (Algorithm 3) as the associative scan algorithm.

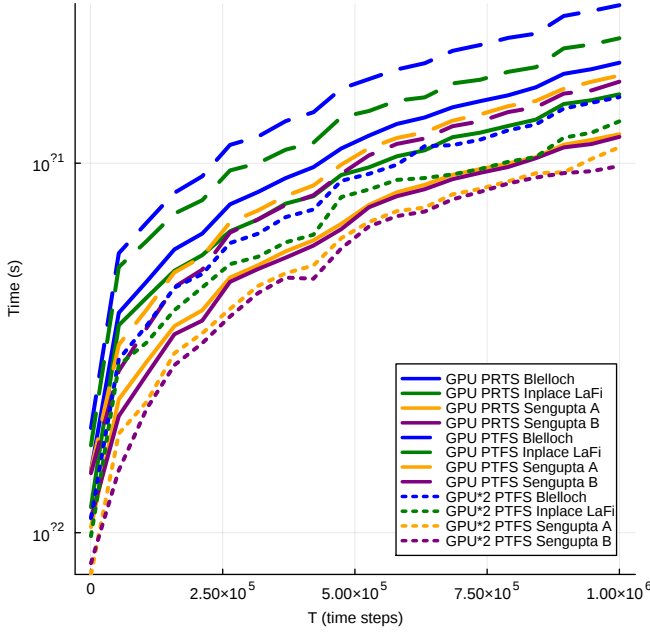


Fig. 12. Run times of the 1-GPU versions of PRTS and PTFS, and the 2-GPU version of PTFS on the CUDA computer as a function of time series length T .

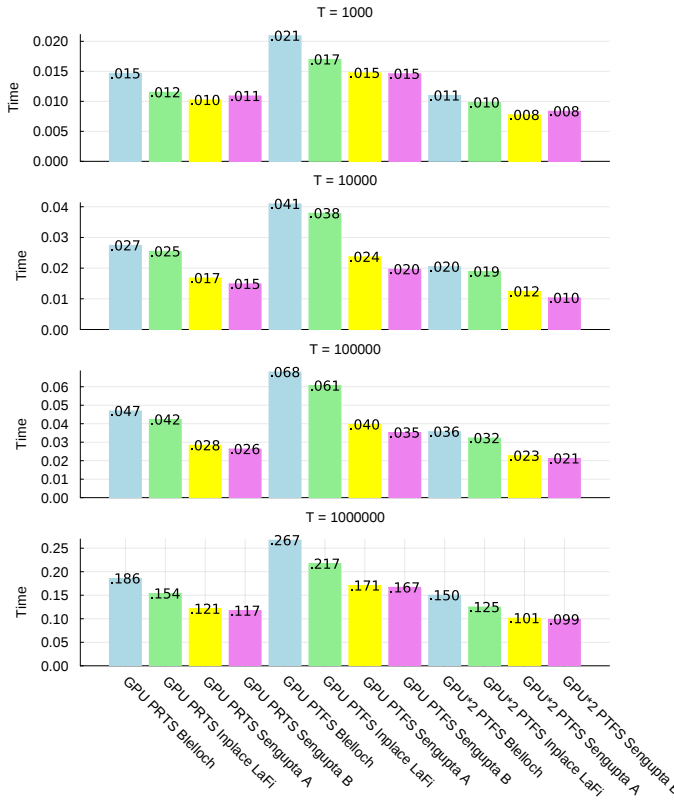


Fig. 13. Run times of the 1-GPU versions of PRTS and PTFS, and the 2-GPU version of PTFS (denoted as GPU*2) on the CUDA computer with time series lengths $T = 10^3, 10^4, 10^5, 10^6$.

The speedups (= ratios of sequential and parallel times) for Metal GPU and CUDA computers are shown in Figures 14 and 15, respectively. It can be seen that the speedup on the Metal GPU increases up to around 750, and the speedup in the CUDA GPU to around 500.

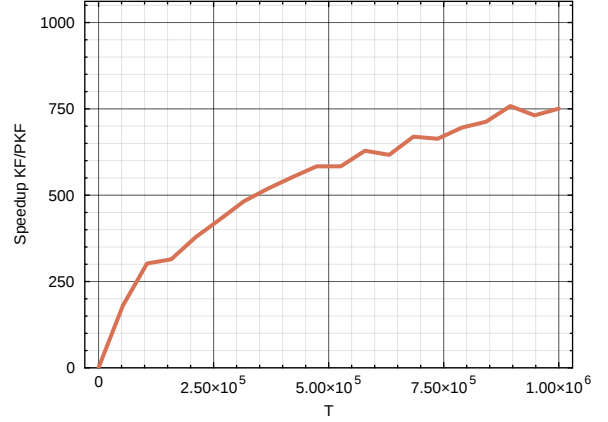


Fig. 14. Speedup of the parallel Kalman filter on the Metal computer as a function of time series length T . The speedup increases from around 0 to a value close to 750.

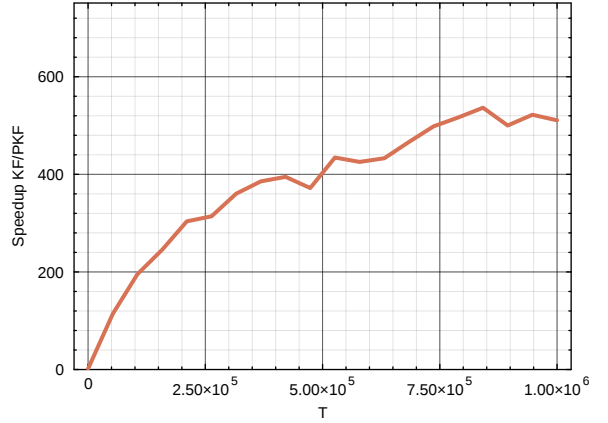


Fig. 15. Speedup of the parallel Kalman filter on the CUDA computer as a function of time series length T . The speedup increases from around 0 to a value close to 500.

VII. CONCLUSION

In this paper, we have performed an experimental evaluation of temporally parallel Kalman filters and smoothers using both simulated and real GPU hardware. A novel parallel two-filter smoother was also proposed, and the Julia codes for the Metal and CUDA GPU experiments are made publicly available.

The experimental results show that the work complexity of the parallel scan method has a significant effect on the practical performance of the methods. In particular, the Hillis–Steele-based methods exhibit the worst performance among the tested parallel scan methods when the time series length is significant, whereas the other methods perform more evenly. The simulated GPU hardware appears to be a very good predictor of the performance of the methods on real GPU

hardware. Furthermore, the performances of the methods on Metal and CUDA GPUs are very similar.

The experimental results also show that while the proposed parallel two-filter smoother (PTFS) is slower than the parallel Rauch-Tung-Striebel (PRTS) smoother on a single GPU, it outperforms the PRTS on a two-GPU setup.

ACKNOWLEDGMENTS

The authors would like to thank the Research Council of Finland for funding.

REFERENCES

- [1] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME, Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.
- [2] H. E. Rauch, F. Tung, and C. T. Striebel, "Maximum likelihood estimates of linear dynamic systems," *AIAA Journal*, vol. 3, no. 8, pp. 1445–1450, Aug. 1965.
- [3] D. Fraser and J. Potter, "The optimum linear smoother as a combination of two optimum linear filters," *IEEE Trans. Autom. Control*, vol. 14, no. 4, pp. 387–390, 1969.
- [4] A. H. Jazwinski, *Stochastic Processes and Filtering Theory*. Academic Press, 1970.
- [5] Y. Bar-Shalom, X.-R. Li, and T. Kirubarajan, *Estimation with Applications to Tracking and Navigation*. Wiley, 2001.
- [6] S. Särkkä and L. Svensson, *Bayesian Filtering and Smoothing*, 2nd ed., ser. Institute of Mathematical Statistics Textbooks. Cambridge University Press, 2023.
- [7] S. Särkkä and A. F. García-Fernández, "Temporal parallelization of Bayesian smoothers," *IEEE Trans. Autom. Control*, vol. 66, pp. 299–306, 2021.
- [8] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.
- [9] —, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, 1990.
- [10] E. K. B. Lee and S. Haykin, "Parallel implementation of the extended square-root covariance filter for tracking applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 446–457, 1993.
- [11] P. M. Lyster, S. E. Cohn, R. Ménard, L. P. Chang, S. J. Lin, and R. G. Olsen, "Parallel implementation of a Kalman filter for constituent data assimilation," *Monthly Weather Review*, vol. 125, no. 7, pp. 1674–1686, 1997.
- [12] O. Rosen and A. Medvedev, "Efficient parallel implementation of state estimation algorithms on multicore platforms," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 1, pp. 107–120, 2013.
- [13] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: <http://github.com/google/jax>
- [14] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: <https://www.tensorflow.org/>
- [15] F. Yaghoobi, A. Corenflos, S. Hassan, and S. Särkkä, "Parallel iterated extended and sigma-point Kalman smoothers," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021.
- [16] —, "Parallel square-root statistical linear regression for inference in nonlinear state space models," *SIAM Journal on Scientific Computing*, vol. 47, no. 2, pp. B454–B476, 2025.
- [17] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.
- [18] S. S. Hassan, S. Särkkä, and A. F. García-Fernández, "Temporal parallelization of inference in hidden Markov models," *IEEE Trans. Signal Process.*, vol. 69, pp. 4875–4887, 2021.
- [19] S. Särkkä and A. F. García-Fernández, "Temporal parallelisation of dynamic programming and linear quadratic control," *IEEE Trans. Autom. Control*, vol. 68, pp. 851–866, 2023.
- [20] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.
- [21] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the Association for Computing Machinery*, vol. 27, no. 4, pp. 831–838, 1980.
- [22] R. M. Karp and V. Ramachandran, "Parallel algorithms for shared-memory machines," in *Algorithms and Complexity*, ser. Handbook of Theoretical Computer Science, J. Van Leeuwen, Ed. Elsevier, 1990, pp. 869–941.
- [23] S. Sengupta, A. Lefohn, and J. Owens, "A work-efficient step-efficient prefix-sum algorithm," in *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*. Chapel Hill, NC, 2006, pp. 26–27.
- [24] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2007, p. 97–106.
- [25] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, 2008, pp. 205–213.
- [26] D. Merrill and A. Grimshaw, "Parallel scan for stream architectures," Department of Computer Science, University of Virginia, Tech. Rep. CS2009-14, 2009.
- [27] C. Breshers, *The art of concurrency: A thread monkey's guide to writing parallel applications*. O'Reilly Media, Inc., 2009.
- [28] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [29] K. J. Kohlhoff, V. S. Pande, and R. B. Altman, "K-means for parallel architectures using all-prefix-sum sorting and updating steps," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1602–1612, 2013.
- [30] J. JaJa, *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [31] Y. C. Ho and R. C. K. Lee, "A Bayesian approach to problems in stochastic estimation and control," *IEEE Transactions on Automatic Control*, vol. 9, no. 4, pp. 333–339, 1964.
- [32] G. Kitagawa, "Non-Gaussian state-space modeling of nonstationary time series," *Journal of the American Statistical Association*, vol. 82, no. 400, pp. 1032–1041, 1987.
- [33] —, "The two-filter formula for smoothing and an implementation of the Gaussian-sum smoother," *Annals of the Institute of Statistical Mathematics*, vol. 46, pp. 605–623, 1994.
- [34] —, "A note on the relation between Balenzuela's algorithm for two-filter formula for smoothing and information filter," 2023. [Online]. Available: <https://arxiv.org/abs/2306.12184>
- [35] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.
- [36] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, vol. 132, pp. 29–46, 2019.
- [37] J. developers, "Metal.jl: Metal programming in Julia," 2025, [Accessed October 15, 2025]. [Online]. Available: <https://github.com/JuliaGPU/Metal.jl>
- [38] —, "CUDA.jl: CUDA programming in Julia," 2025, [Accessed October 15, 2025]. [Online]. Available: <https://github.com/JuliaGPU/CUDA.jl>



Simo Särkkä received his MSc. degree (with distinction) in engineering physics and mathematics, and DSc. degree (with distinction) in electrical and communications engineering from Helsinki University of Technology, Espoo, Finland, in 2000 and 2006, respectively. Currently, he is a Professor with Aalto University. His research interests are in multi-sensor data processing and control systems. He has authored or coauthored over 200 peer-reviewed scientific articles and three books. He is a Senior Member of IEEE.



Ángel F. García-Fernández received the telecommunication engineering degree and the Ph.D. degree from Universidad Politécnica de Madrid (UPM), Spain, in 2007 and 2011, respectively.

He is currently an Associate Professor at the Information Processing and Telecommunications Center of UPM. Previously, he held other academic positions at Chalmers University of Technology, Sweden, Curtin University, Australia, Aalto University, Finland, and the University of Liverpool, UK. His main research interests are in the area of Bayesian

inference, with emphasis on multiple target tracking.