Aalto University

Parallel/distributed methods for state-space models

Simo Särkkä
Aalto University / Finnish Center for Artificial Intelligence (FCAI)

Contents

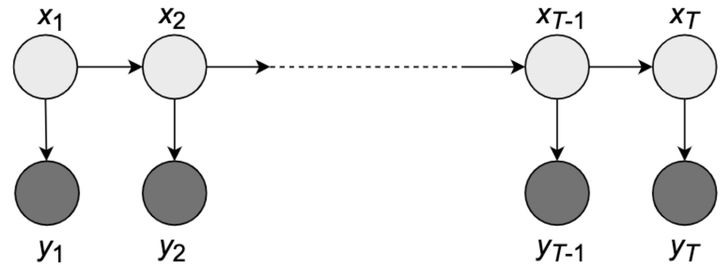
- Goal
- Algorithms
- Implementations
- Experiments
- Conclusion

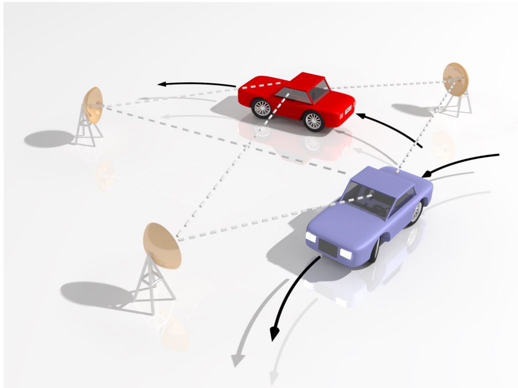
Goal

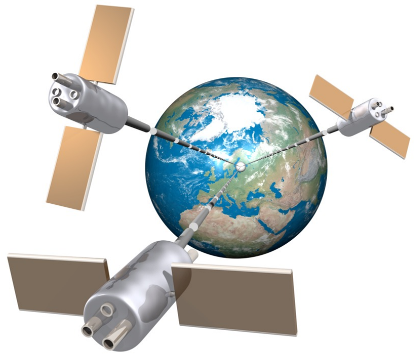
Probabilistic state space models

- **Important in many fields:**
  - target tracking
  - space-craft guidance
  - machine learning
  - speech processing
  - audio signal processing
  - biomedicine

$$\mathbf{x}_k \sim p(\mathbf{x}_k \mid \mathbf{x}_{k-1}),$$
$$\mathbf{y}_k \sim p(\mathbf{y}_k \mid \mathbf{x}_k),$$

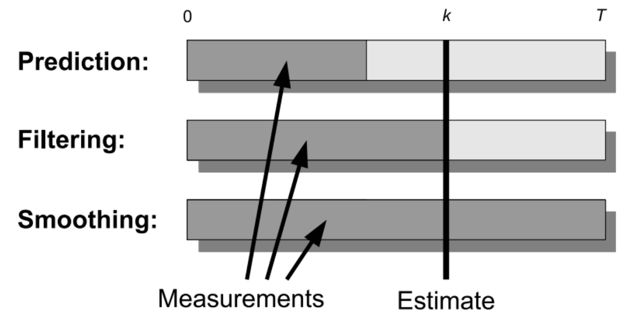


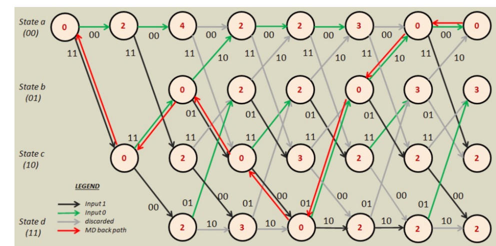


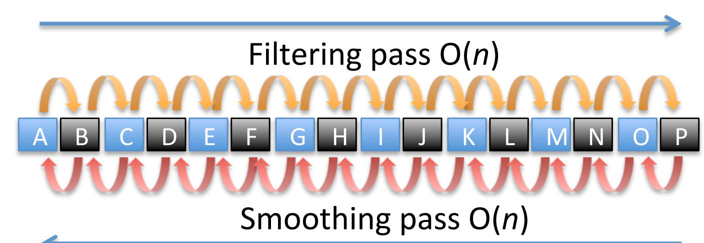


Inference in state space models

- **Many efficient algorithms:**
  - Bayesian filter and smoother
  - Kalman filters and smoothers
  - Forward-backward algorithms
  - Viterbi algorithm
- **Not designed for parallelism:**
  - $O(n)$ steps with n measurements
  - Nice, but we can do better by parallel computing

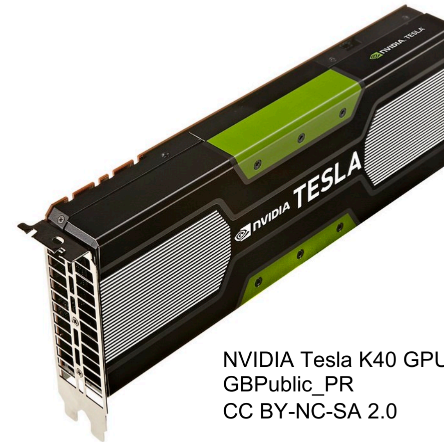




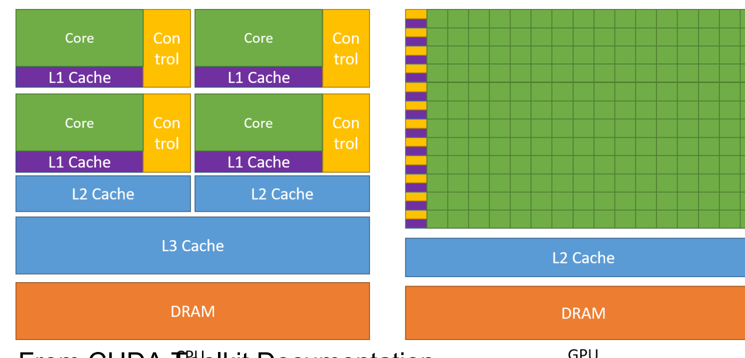


Graphics processing units

- Graphics processing unit (GPU) is a device with a huge number of cores
- Can run million(s) of execution threads simultaneously
- Very well suited for parallel computing
- Extensively used in deep learning nowadays (TensorFlow etc.)
- The programming model is a bit different than of "normal" CPUs



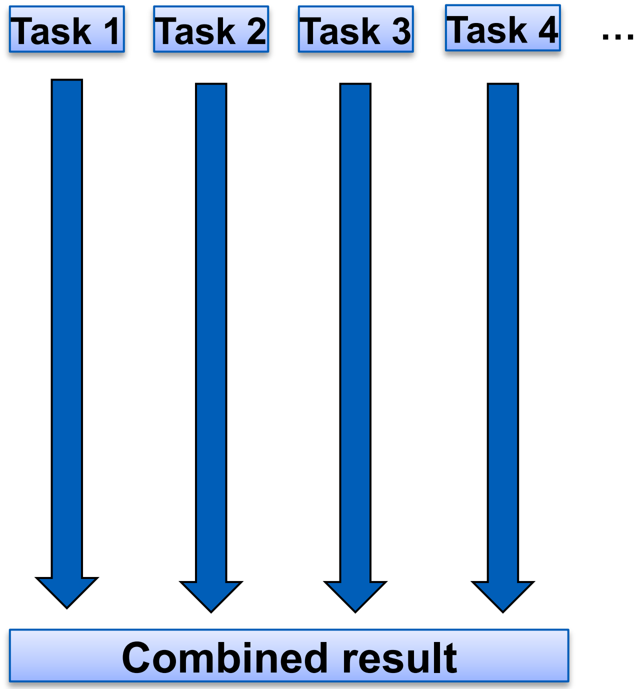
NVIDIA Tesla K40 GPU by GBPublic_PR



From CUDA Toolkit Documentation
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

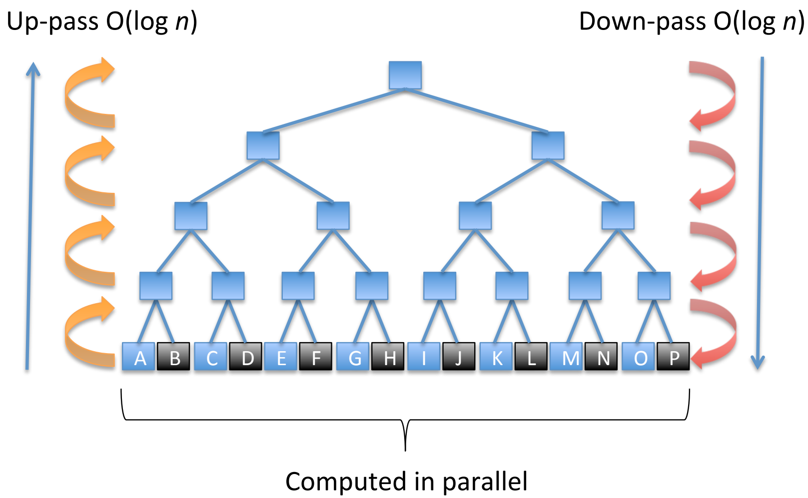
Parallelizing computations

- GPUs cannot just parallelize ”any” algorithm
- Need to be able to decompose problem into independent subproblems/tasks
- Embarrassingly parallel problems are easy
- Surprising many sequential algorithms can be reduced to “all-prefix-sum” computation
- For arbitrary “sum” operation, can be solved via divide-and-conquer in O(log n) parallel time
- Can we do this for inference in state-space models?



Parallel inference in state-space models

- **It turns out that it is possible:**
  - We can use the *scan* algorithm to get to $O(\log n)$
- **Requires associative operator reformulation of filtering/smoothing/Viterbi**
- **Suitable for GPUs and parallel clusters (or TPUs/NPUs).**
- **Perfect for e.g. TensorFlow and other similar frameworks**



Algorithms

All-prefix-sums problem

- **We have a sequence of numbers**

  $[a_1,\ldots,a_n]$,
- **We wish to compute the prefix sums**

  $[a_1,(a_1 + a_2),(a_1 + a_2 + a_3),\ldots]$
- **More generally, for an associative operator ✧ we want**

  $[a_1,(a_1 ✧ a_2),(a_1 ✧ a_2 ✧ a_3),\ldots]$
- **Simple sequential solution**

  $s_0 = 0$ (or the neutral element for ✧)

  for $i = 1,\ldots,n$

  $s_i = s_{i-1} ✧ a_i$

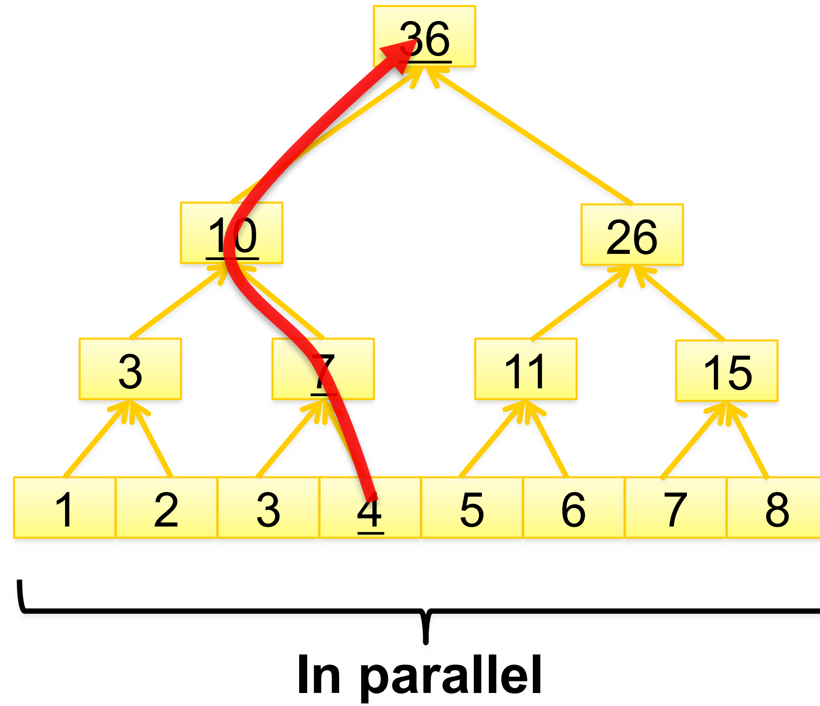
- **Associative operation ✧:**

  $(a ✧ b) ✧ c = a ✧ (b ✧ c)$
- **Neutral element 0:**

  $0 ✧ a = a$

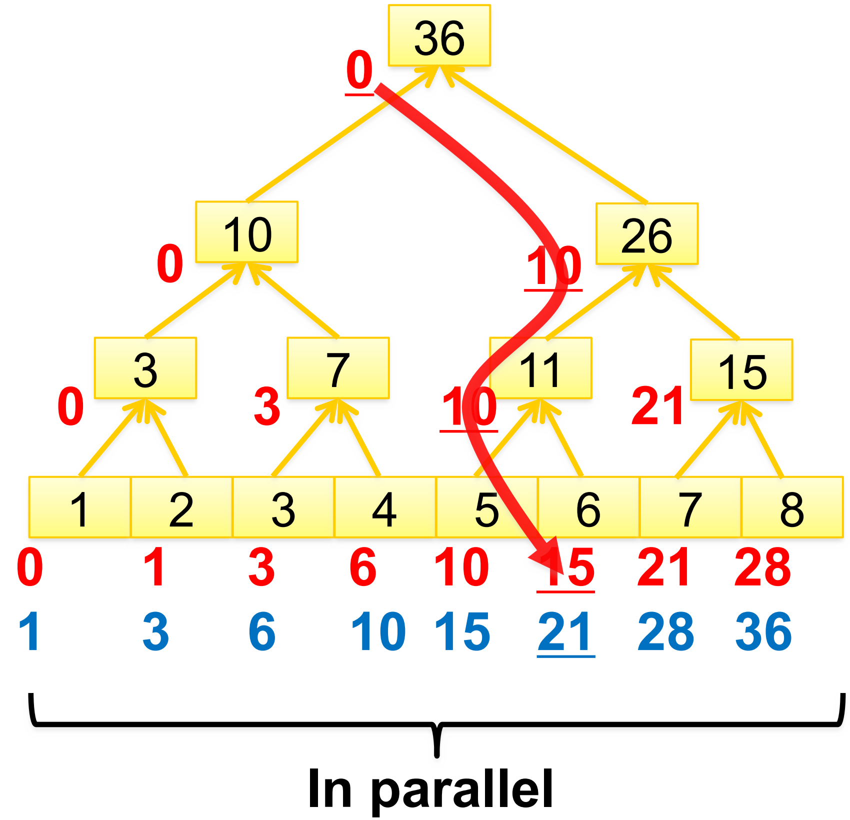
Parallel all-prefix-sums (scan): up-sweep

- **Consider, for example, sums**
  [1 2 3 4 5 6 7 8]
- **We start by forming a binary tree of sums**
- **Can be parallelized in side-direction -> *O*(log *n*) steps**
  - Actually, needs $O(n)$ operations, but *span* complexity is $O(\log n)$
- **Also works for more general associative operators ✧**



Parallel all-prefix-sums (scan): down-sweep

- **Let's assign a (left) value *L* to each node as follows:**
  - Root has $L = 0$
  - Every left child inherits the present value L
  - Every right child gets the left child's sum plus current L
  - At leaf we output $s_i = L + a_i$
- **We get all-prefix-sums in *O*(log *n*) steps.**



Demonstration with concatenation

Up:

```
                abcdefgh
          abcd              efgh
     ab         cd        ef          gh
   a    b    c    d    e    f    g    h
```

Down:

```
                          ()abcdefgh
         ()abcd                             (abcd)efgh
     ()ab         (ab)cd          (abcd)ef               (abcdef)gh
  ()a    (a)b   (ab)c   (abc)d   (abcd)e   (abcde)f   (abcdef)g
(abcdefg)h
```

- Root has L = ‘’
- Every left child inherits the present value L
- Every right child gets the left child's string plus current L
- At leaf we output L plus a_i

In-place algorithm by Blelloch (1990)

```
a = [1 2 3 4 5 6 7 8];
n = length(a);
s = a;

fprintf('-- up-sweep\n');
for d=0:log2(n)-1
    for i=0:2^(d+1):n-1 % This is a parallel loop
        i1 = i + 2^d;
        i2 = i + 2^(d+1);
        s(i2) = s(i1) + s(i2);
        fprintf('%d ',s); fprintf('\n');
    end
end

fprintf('-- down-sweep\n');
s(n) = 0;
for d=log2(n)-1:-1:0
    for i=0:2^(d+1):n-1 % This is a parallel loop
        i1 = i + 2^d;
        i2 = i + 2^(d+1);
        t = s(i1);
        s(i1) = s(i2);
        s(i2) = s(i2) + t;
        fprintf('%d ',s); fprintf('\n');
    end
end

s = s + a; % in parallel
fprintf('-- result\n');
fprintf('%d ',s); fprintf('\n');
```

```
-- up-sweep
1 3 3 4 5 6 7 8
1 3 3 7 5 6 7 8
1 3 3 7 5 11 7 8
1 3 3 7 5 11 7 15
1 3 3 10 5 11 7 15
1 3 3 10 5 11 7 26
1 3 3 10 5 11 7 36
-- down-sweep
1 3 3 0 5 11 7 10
1 0 3 3 5 11 7 10
1 0 3 3 5 10 7 21
0 1 3 3 5 10 7 21
0 1 3 6 5 10 7 21
0 1 3 6 10 15 7 21
0 1 3 6 10 15 21 28
-- result
1 3 6 10 15 21 28 36
```

Parallelization of Bayesian filtering

- **Classical Bayesian filter:**

$$p(x_k \mid y_{1:k-1}) = \int p(x_k \mid x_{k-1})\, p(x_{k-1} \mid y_{1:k-1})\, \mathrm{d}x_{k-1},$$

$$p(x_k \mid y_{1:k}) = \frac{p(y_k \mid x_k)\, p(x_k \mid y_{1:k-1})}{\int p(y_k \mid x_k)\, p(x_k \mid y_{1:k-1})\, \mathrm{d}x_k}.$$

- **Parallelization elements:**

$$a_k = (f_k, g_k) \in \mathcal{F}$$

$$f_k\left(x_k \mid x_{k-1}\right) = p\left(x_k \mid y_k, x_{k-1}\right),$$

$$g_k\left(x_{k-1}\right) = p\left(y_k \mid x_{k-1}\right),$$

- **Associative operator:**

$$\left(f_i, g_i\right) \otimes \left(f_j, g_j\right) = \left(f_{ij}, g_{ij}\right),$$

$$f_{ij}\left(x \mid z\right) = \frac{\int g_j\left(y\right) f_j\left(x \mid y\right) f_i\left(y \mid z\right) \mathrm{d}y}{\int g_j\left(y\right) f_i\left(y \mid z\right) \mathrm{d}y},$$

$$g_{ij}\left(z\right) = g_i\left(z\right) \int g_j\left(y\right) f_i\left(y \mid z\right) \mathrm{d}y.$$

- **Final result:**

$$a_1 \otimes a_2 \otimes \cdots \otimes a_k = \begin{pmatrix} p\left(x_k \mid y_{1:k}\right) \\ p\left(y_{1:k}\right) \end{pmatrix}.$$

Ref: Särkkä, S. and García-Fernández, Á. F. (2021). Temporal Parallelization of Bayesian Smoothers. IEEE Trans. Autom. Control, 66(1):299-306. arXiv:1905.13002

Parallelization of Kalman filtering

- **In linear Gaussian case we can parametrize by**

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

$$\begin{aligned}
f_i(y \mid z) &= \mathrm{N}(y; A_i z + b_i, C_i), \\
g_i(z) &\propto \mathrm{N}_I(z; \eta_i, J_i), \\
f_j(y \mid z) &= \mathrm{N}\left(y; A_j z + b_j, C_j\right), \\
g_j(z) &\propto \mathrm{N}_I\left(z; \eta_j, J_j\right),
\end{aligned}$$

- **The operator becomes**

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

- **The filter means will be in elements *b*, and covariances in *C*.**

Parallelization of Bayesian smoothing

- **General elements & operator:**

$$a_k = p(x_k \mid y_{1:k}, x_{k+1}) \in \mathcal{S},$$

$$a_i \otimes a_j = a_{ij},$$

$$a_{ij}(x \mid z) = \int a_i(x \mid y)\, a_j(y \mid z)\, \mathrm{d}y.$$

$$a_k \otimes a_{k+1} \otimes \cdots \otimes a_n = p(x_k \mid y_{1:n}).$$

- **Linear Gaussian case:**

$$\begin{aligned} a_k(x_k \mid x_{k+1}) &= p(x_k \mid y_{1:k}, x_{k+1}) \\ &= \mathrm{N}(x_k; E_k x_{k+1} + g_k, L_k), \end{aligned}$$

$$\begin{aligned} E_{ij} &= E_i E_j, \\ g_{ij} &= E_i g_j + g_i, \\ L_{ij} &= E_i L_j E_i^\top + L_i. \end{aligned}$$

Parallel Extended and Sigma-Point Smoothers

- **Extended Kalman filters and smoothers are non-linear versions of Kalman filters and smoothers**
- **IEKS iteratively linearized via Taylor series, and the applies a Kalman smoother**
  - The linearization can be made in parallel
  - Kalman pass can be parallelized via the linear/Gaussian method
- **Iterated sigma-point methods (iterated UKS etc.) can be interpreted as using statistical linear regression instead**
  - Can be parallelized in an analogous manner

Ref: Yaghoobi, F., Corenflos, A., Hassan, S. and Särkkä. S. Parallel Iterated Extended and Sigma-Point Kalman Smoothers (2021). Proc. ICASSP. arXiv:2102.00514

Parallel Inference in Hidden Markov Models (HMMs)

- **HMMs are discrete-state state-space models**
- **Naturally formulated in terms of potentials**

$$\psi_1(x_1) = p(y_1 \mid x_1)\, p(x_1),$$
$$\psi_k(x_{k-1}, x_k) = p(y_k, \mid x_k)\, p(x_k \mid x_{k-1}),$$

- **Inference reduces to sum-products/max-products for**

$$p(\mathbf{x}) = \frac{1}{Z} \psi_1(x_1) \prod_{t=2}^{T} \psi_t(x_{t-1}, x_t).$$

- **The parallelization elements *a* are now the potentials**
- **Sum-product operator:**

$$a_{i:j} \otimes a_{j:k} = \sum_{x_j} \psi_{i,j}(x_i, x_j) \psi_{j,k}(x_j, x_k),$$

- **Max-product operator is analogous with max()**
- **Viterbi can be done as well**

Ref: Hassan, S. S., Särkkä, S. and García-Fernández, Á. F. Temporal Parallelization of Inference in Hidden Markov Models. IEEE Trans.Sig.Proc., 69:4875-4887. arXiv:2102.05743

Implementations

Belloch in GPU with CUDA

- Numba has CUDA target, which can be used to generate GPU code
- Alternative would be C/C++ based CUDA which is more tedious to use
- For example, the implementations of array extension to 2^n and down pass look as on the right
- The loop over tree levels is in CPU
- Can be directly used in Google Colab (which has GPUs)

```python
@cuda.jit
def prefix_sum_prescan(a,tmp):
    start  = cuda.grid(1)
    stride = cuda.gridsize(1)

    orig_n = len(a)
    n = len(tmp) # The size of tmp must be 2**ceil(log2(n))

    for i in range(start, n, stride):
        if i < orig_n:
            tmp[i] = a[i]
        else:
            tmp[i] = 0 # Should be neutral element of the operator
```

```python
@cuda.jit
def prefix_sum_up(d,tmp):
    start  = cuda.grid(1)
    stride = cuda.gridsize(1)

    n = len(tmp) # The size of tmp must be 2**ceil(log2(n))

    for i in range(start, n, stride):
        step     = 1 << (d+1)
        halfstep = 1 << d

        if i % step == step - 1:
            i1 = i - halfstep
            i2 = i
            tmp[i2] = tmp[i1] + tmp[i2] # more generally op(tmp[i1],tmp[i2])
```

Parallel filter and smoother with CUDA

- We can implement filters/smoothers by replacing “+” with the matrix operations
- It turns out that Numba only has a limited support for matrix routines in kernels
- Another option is to use C++ CUDA which supports Eigen matrix library
- Yet another option would be to use say OpenCL
- This can be a bit tedious (fun) though

Associative scan in TensorFlow & JAX

- TensorFlow already has the Blelloch algorithm:

```
tfp.math.scan_associative(
    fn, elems, max_num_levels=48, validate_args=False, name=None
)
```

- JAX also has it:

```
jax.lax.associative_scan(fn, elems, reverse=False)    [source]
```

Perform a scan with an associative binary operation, in parallel.

- These also have full matrix support – and automatic differentiation support
- TensorFlow probability even has new “parallel_filter” which is … well … implementation of Särkkä, S. and García-Fernández, Á. F. (2021).

Parallel state-space inference in TensorFlow and JAX

Examples for the paper Temporal Parallelization of Bayesian Smoothers [1]

Author: Adrien Corenflos

These notebooks illustrate the gain in perfomance of using the parallel implementation of Kalman filters and smoothers on GPU. They can be downloaded to be run locally or on Google Colab:

- JAX: Open in Colab
- TensorFlow: Open in Colab

Last run with:

- Tensorflow:
  tensorflow==2.4.0
  tensorflow_probability==0.11.0
- JAX:
  jax==0.1.77
  jaxlib==0.1.55+cuda101

https://github.com/EEA-sensors/sequential-parallelization-examples/tree/main/python/temporal-parallelization-bayes-smoothers

Parallel Iterated Extended and Sigma-Point Kalman Smoothers

Companion code in JAX for the paper Parallel Iterated Extended and Sigma-Point Kalman Smoothers [2].

What is it?

This is an implementation of parallelized Extended and Sigma-Points Bayesian Filters and Smoothers with CPU/GPU/TPU support coded using JAX primitives, in particular associative scan.

Supported features

- Extended Kalman Filtering and Smoothing
- Cubature Kalman Filtering and Smoothing
- Iterated versions of the above

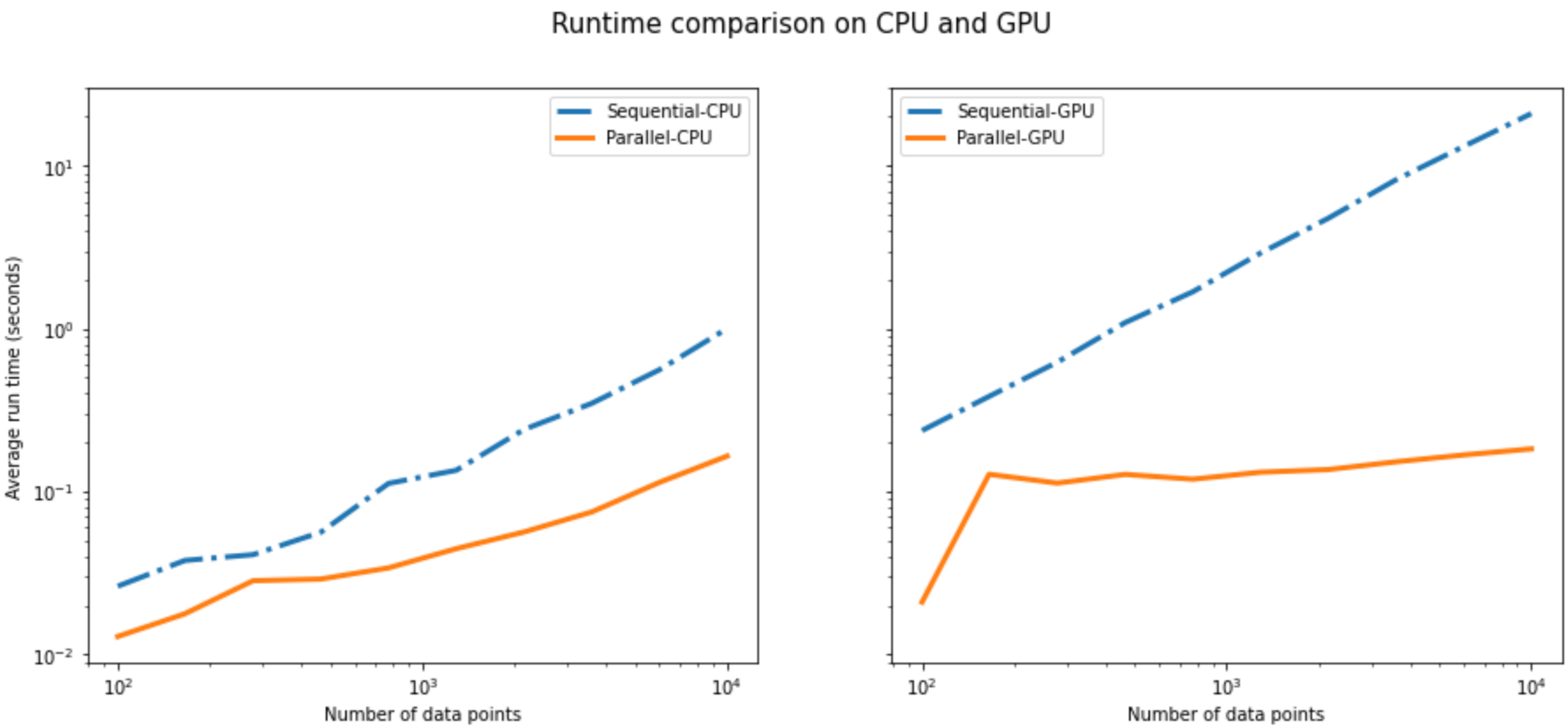
Installation

- With GPU CUDA 11.0 support
  - Using pip
    Run `pip install https://github.com/EEA-sensors/parallel-non-linear-gaussian-smoothers.git -f https://storage.googleapis.com/jax-`

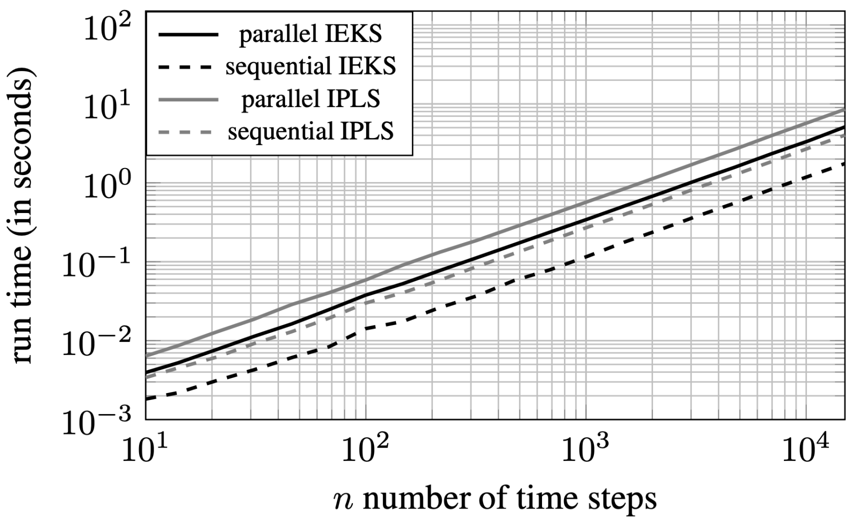
https://github.com/EEA-sensors/parallel-non-linear-gaussian-smoothers

Experiments

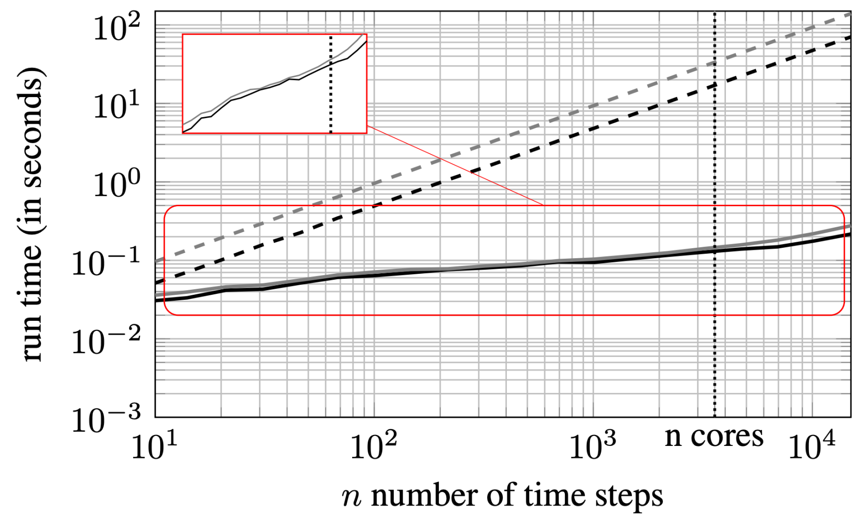
Results for linear Gaussian systems



Results for non-linear systems

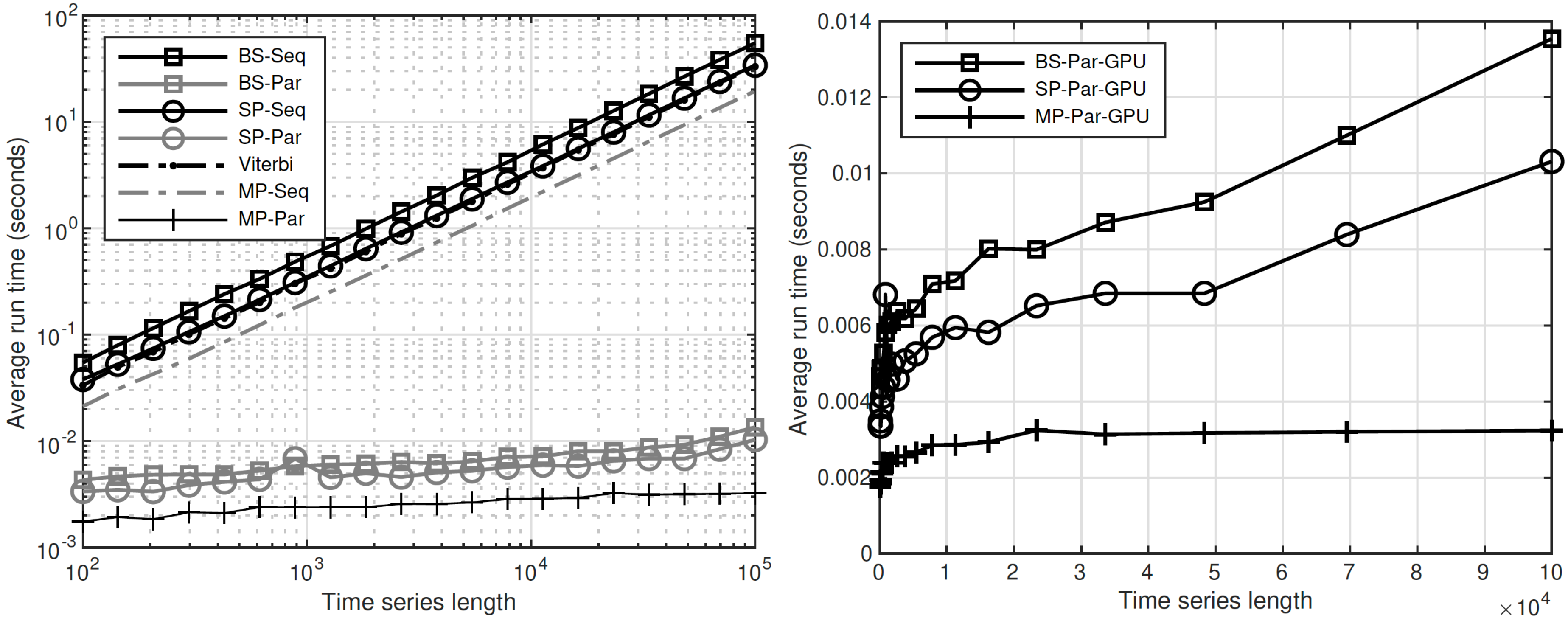


(a) CPU run time



(b) GPU run time

Results for HMMs



Aalto University

Conclusion

Summary

- **All-prefix-sums can be computed in parallel using *scan* algorithm of which one is by Blelloch.**
- **The “sum” can be any associative operation.**
- **Useful in especially for parallel computing in GPUs.**
- **Inference in probabilistic state-space models can be reformulated as a sequence of associative operations.**
- **We get parallel Kalman/Bayesian/Viterbi methods**
- **We can implement everything with CUDA on GPU, but TensorFlow and JAX already have the scan implemented.**

Some references

Blelloch, G. E. (1989). Scans as primitive parallel operations. IEEE TransComp.

Blelloch, G. E. (1990). Prefix sums and their applications. TechRep CMU-CS-90-190.

Cook, S. (2013). CUDA programming: a developer's guide to parallel computing with GPUs.

Murphy, K. P. (2012). Machine Learning: a Probabilistic Perspective. MIT Press.

Särkkä, S. (2013). Bayesian Filtering and Smoothing. Cambridge University Press.

Särkkä, S. and García-Fernández, Á. F. (2021). Temporal Parallelization of Bayesian Smoothers. IEEE Trans. Autom. Control, 66(1):299-306. arXiv:1905.13002

Yaghoobi, F., Corenflos, A., Hassan, S. and Särkkä. S. Parallel Iterated Extended and Sigma-Point Kalman Smoothers (2021). Proc. ICASSP. arXiv:2102.00514

Hassan, S. S., Särkkä, S. and García-Fernández, Á. F. (2021). Temporal Parallelization of Inference in Hidden Markov Models. IEEE Trans.Sig.Proc., 69:4875-4887 arXiv:2102.05743