High-Performance Tensor Formulation of the Viterbi Algorithm for Hidden Semi-Markov Models

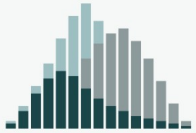
Lorenzo Piarulli Elia Belli Daniele De Sensi

HLC-Lab, Department of Computer Science, Sapienza University of Rome





SAPIENZA
UNIVERSITÀ DI ROMA

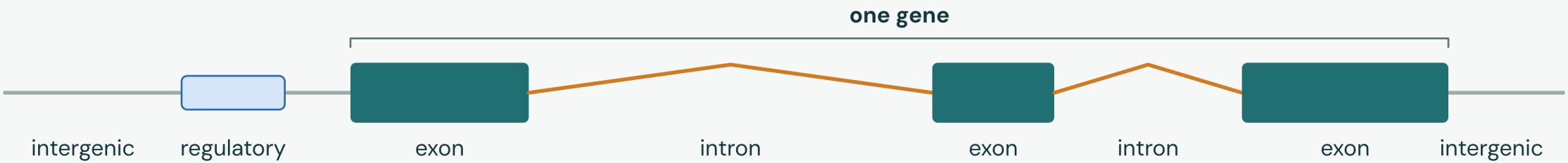


hlc
High-Performance
Large-Scale
Computing

SC26 · CHICAGO, IL · NOVEMBER 15–20, 2026

MOTIVATION

The genome annotation problem



…GATTACACCGTAGGCTAATCGGATCCATGGCGTACGTTAGCCTAAGTCGATCGGTACCATTGCAAGTCTAGGTTCAGGCTAACGTATVCGT…

WHY IT IS IMPORTANT

Every new genome needs it: it is the map for studying genes, proteins and disease variants.

WHY IT IS HARD

Boundary signals are short and ambiguous, and regions span from dozens to thousands of bases.

HOW IT IS SOLVED

Gene finders such as AUGUSTUS and SNAP decode the sequence with Generalized Hidden Markov models, based on **Hidden Semi Markov Models**.

MOTIVATION

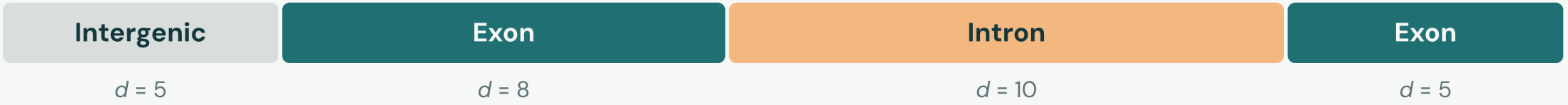
What's Viterbi? an example

OBSERVATIONS · THE DNA SEQUENCE

T	T	A	C	G	A	T	G	G	C	A	G	C	G	T	A	A	G	T	C	T	A	G	G	C	T	T	A

Viterbi Decoding: infer the hidden state of every base

HIDDEN STATES · THE GENETIC REGION



Each region is a hidden state that **lasts for a duration**: in real genomes, from dozens to thousands of bases.

MOTIVATION

Regions have lengths: from HMMs to HSMMs

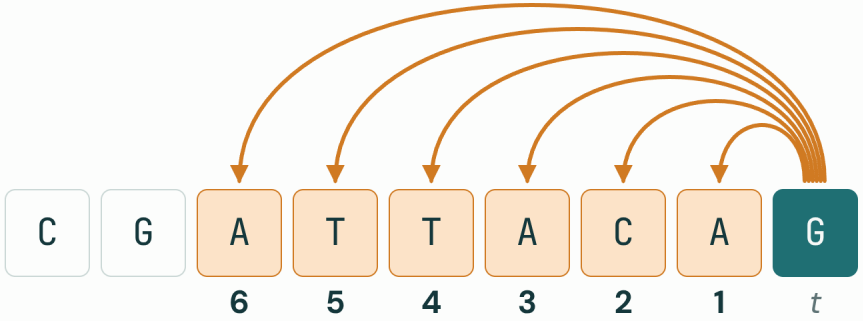
Hidden Markov Model



look-back distance

Distance is always 1

Hidden Semi-Markov Model



look-back distance (here $D = 6$)

Any distance up to D: check all of them

HMM

$$O(T \cdot N^2)$$

HSMM

$$O(T \cdot N^2 \cdot D)$$

with D from hundreds to thousands

MOTIVATION

Regions have lengths: from HMMs to HSMMs

2.3 h

one decode on one core

N = 25, *D* = 1,000, *T* = 1M
hsmmlearn on Xeon 8480+

148 days

extreme scale, one core

N = 100, *D* = 10,000, *T* = 10M
It fails, estimated

0

GPU implementations

existing HSMM tools are sequential and single-threaded

So practitioners truncate their data or simplify their models.

OVERVIEW

Our contributions

01

Tensor formulation

The inner loops of HSMM Viterbi become dense tensor operations, with identical output.

02

Four implementations

Single- and multi-core CPU, and the first GPU version. Open source: tensor-hsmm.

03

Broad evaluation

3 CPU and 5 GPU architectures, with problem sizes from genomics.

14×
single core

>200×
multi-core

>570×
GPU

Speedups over the sequential C++ baseline (hsmmlearn).

Methodology

1 **Tensor formulation**

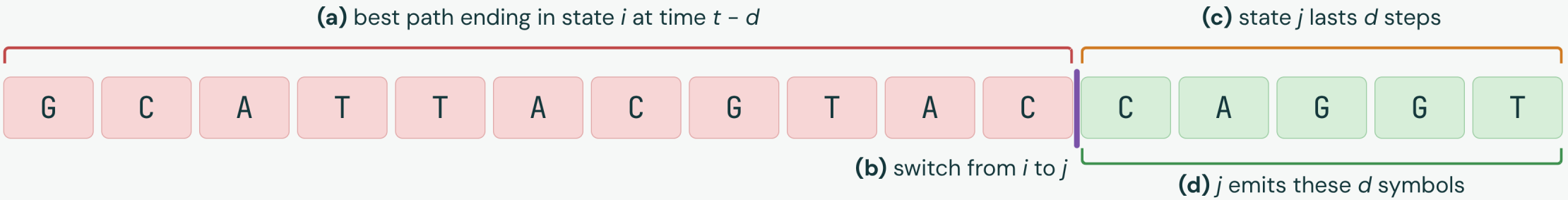
the theory: from nested loops to broadcasted tensor operations

2 Implementation

four versions, CPU and GPU optimizations

Viterbi: build the best path segment by segment

$\delta_t(j)$ = score of the best path ending in state *j* at time *t*: the best over every previous state *i* and every duration *d*.



LABELS USED FROM HERE ON

(a) δ	(b) A	(c) P	(d) B	π
best scores · $N \times T$	transitions · $N \times N$	durations · $N \times D$	emissions · $N \times M$	initial · N

Viterbi decodes **and** trains: ab initio models alternate decoding and re-estimation dozens of times.

The classic algorithm: four nested loops

$$\delta_t(j) \;=\; \max_{\forall i,\,\forall d} \Big[\; \underbrace{\delta_{t-d}(i)}_{\text{(a)}} \cdot \underbrace{a_{ij}}_{\text{(b)}} \cdot \underbrace{p_j(d)}_{\text{(c)}} \cdot \underbrace{\prod_{k=0}^{d-1} b_j(o_{t-k})}_{\text{(d)}} \;\Big]$$

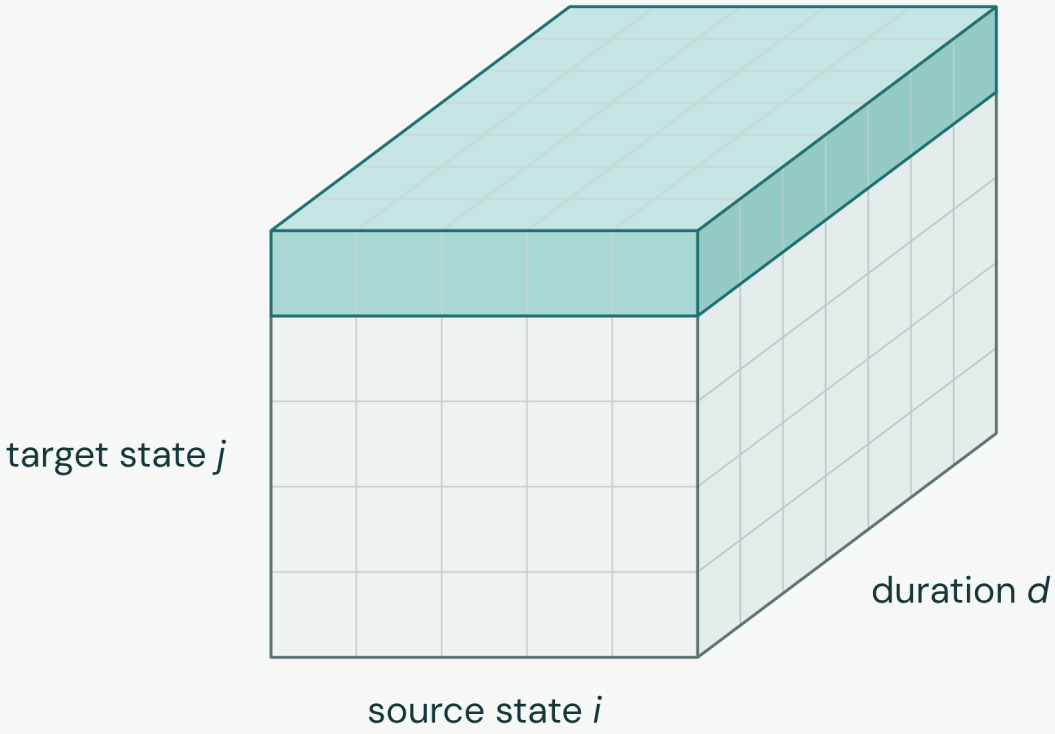
- **(a)** best path to *i*, *d* steps ago
- **(b)** transition *i* → *j*
- **(c)** *j* lasts exactly *d* steps
- **(d)** *j* emits the last *d* symbols

```
for t = 2 … T
   for j = 1 … N
      for d = 1 … min(t, D)
         for i = 1 … N
            δ[t-d][i] · a[i][j] · p[j][d] · ∏ b
            keep the max → δ[t][j], ψ[t][j]
```

$O(T \cdot N^2 \cdot D)$

The *d* and *i* loops feed one running max: only the *j* loop is parallel, so at most *N* threads.

Key idea: the inner loops form a 3D tensor



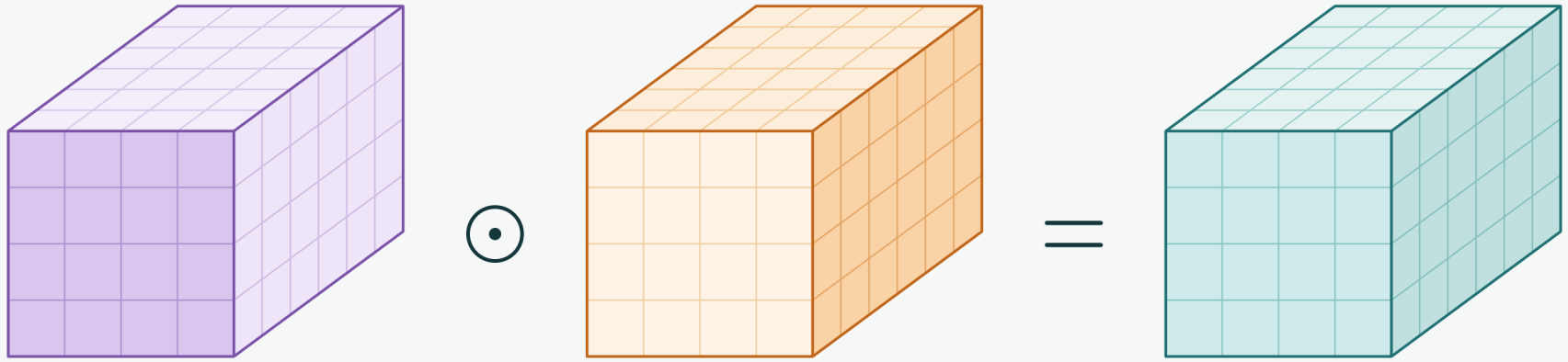
For a fixed time *t*, every (*j*, *i*, *d*) candidate is one cell of an ***N* × *N* × *D*** tensor: the **Brick**.

one slice = all (*i*, *d*) candidates for one target *j*

1. **Construction** *only once*
   the time-invariant factors
2. **Update** *every step*
   add the time-dependent factors
3. **Max extraction** *every step*
   the best (*i*, *d*) in each slice

Phase 1: build the static Brick once

$$\mathcal{B}_{first}^{N\times N\times D} = \widetilde{\mathbf{A}}^{N\times N\times\uparrow} \odot \widetilde{\mathbf{P}}^{N\times\uparrow\times D}$$



A · $N \times N$
repeated along d

P · $N \times D$
repeated along i

Static Brick
reused at every step

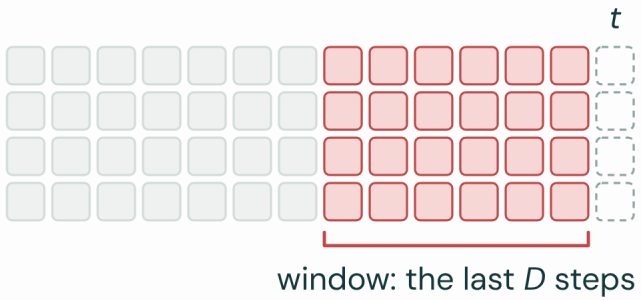
Transitions and durations do not depend on t: their product is computed once, before the time loop.

Phase 2: update the Brick at every time step

Past deltas

$$\Delta_{past}[\,i,\,d\,] = \delta_{t-d}(i)$$

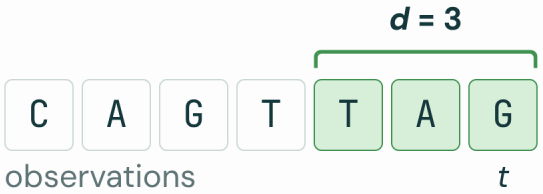
The last *D* deltas of every state, newest first; repeated along *j*.



Emission product

$$\mathbf{E}[\,j,\,d\,] = \prod_{k=0}^{d-1} b_j(o_{t-k})$$

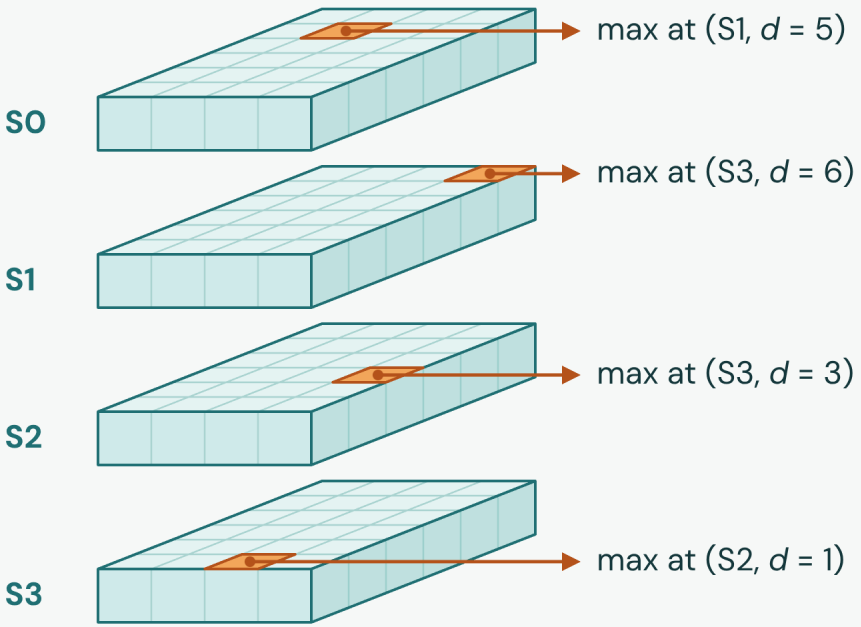
Product of the last *d* emissions of state *j*; repeated along *i*.



$$\mathcal{B}_{(t)}^{N\times N\times D} = \mathcal{B}_{first}^{N\times N\times D} \odot \widetilde{\Delta}_{past\,(t)}^{\uparrow\times N\times D} \odot \widetilde{\mathbf{E}}_{(t)}^{N\times\uparrow\times D}$$

Phase 3: one max per slice, then backtrack

$$\delta_t(j) = \max_{(i,\,d)} \mathcal{B}_{(t)}[j,\, i,\, d] \qquad \psi_t(j) = \arg\max_{(i,\,d)} \mathcal{B}_{(t)}[j,\, i,\, d]$$



EVERY TIME STEP

1. Update the emission product
2. Slide the window of past deltas
3. Form the Brick: two broadcasted products
4. Max and argmax in every slice

INITIALIZATION · FIRST *D* STEPS

$$\boldsymbol{\Delta}[\forall j,\, 1{:}D]^{N\times D} = \boldsymbol{\pi}^{N\times\uparrow} \odot \mathbf{P}^{N\times D} \odot \mathbf{E}^{N\times D}$$

BACKTRACKING

Follow the stored (*i*, *d*) back from the end: sequential, and cheap.

Methodology

1 Tensor formulation

the theory: from nested loops to broadcasted tensor operations

2 **Implementation**

four versions, CPU and GPU optimizations

METHODOLOGY · IMPLEMENTATION

One formulation, four implementations

Version	Target	Built with	Role
Tens-Py	CPU, single core	Python, NumPy	readable reference, validation
Tens-1C	CPU, single core	C++	cache-aware, fused, branchless
Tens-MC	CPU, multi-core	C++, OpenMP	N·D-way parallel
Tens-GPU	NVIDIA and AMD GPUs	CUDA, HIP	one codebase, ported with hipify

ONE TIME STEP IN NUMPY · SKETCH

```
B = B_first + delta_past[None, :, :] + E[:, None, :]
delta[t] = B.reshape(N, -1).max(axis=1)
psi[t] = B.reshape(N, -1).argmax(axis=1)
```

LOG-SPACE

Products become sums: no underflow, and the update is plain broadcasted additions.

CPU: what the tensor view unlocks

01

Cache-friendly layout

The Brick is stored as (j, d, i): each state's $D \times N$ block fits in L2 when small enough, and the inner loop over i is contiguous.

02

Cached emissions

A rolling cache adds one new log-emission per entry: from $O(D)$ to amortized $O(1)$ per state and step.

03

Fused update + argmax

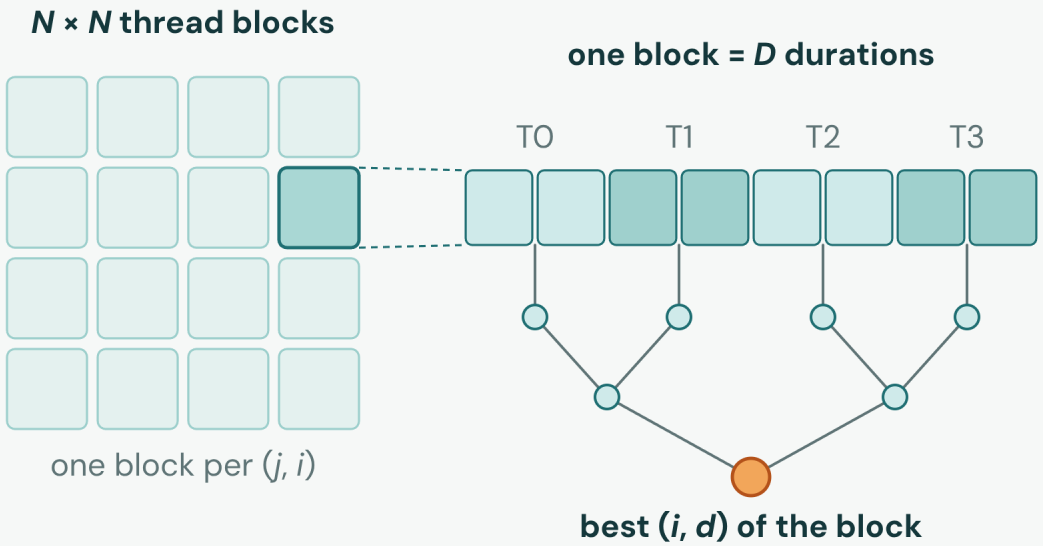
The full Brick is never stored. The inner loop is branchless: predicated instructions, no mispredictions.

04

$N{\cdot}D$-way multi-core

Phase A: threads take (j, d) pairs and scan all i. Phase B: reduce over d. One persistent OpenMP team.

GPU: the Brick mapped onto thread blocks



Kernel 1 · update + local max

Threads compute their Brick elements in shared memory; the block reduces them to its best (*i*, *d*), finishing with warp shuffles.

Kernel 2 · global argmax

Combines the block maxima of each target state *j*. Two kernels avoid a grid-wide synchronization.

d-contiguous layouts for coalesced loads · double-buffered emission cache

EVALUATION

Experimental setup

Baselines

Base-1C: hsmmlearn, the C++ core of the R hsmm package

Base-MC: Base-1C with OpenMP over states (ours)

Problem sizes

taken from computational genomics

N = 10 – 75 states

D = 100 – 10,000

T = 1k – 10M steps

Hardware

CPUS

AMD EPYC 7A53 · NVIDIA Grace · Intel Xeon 8480+

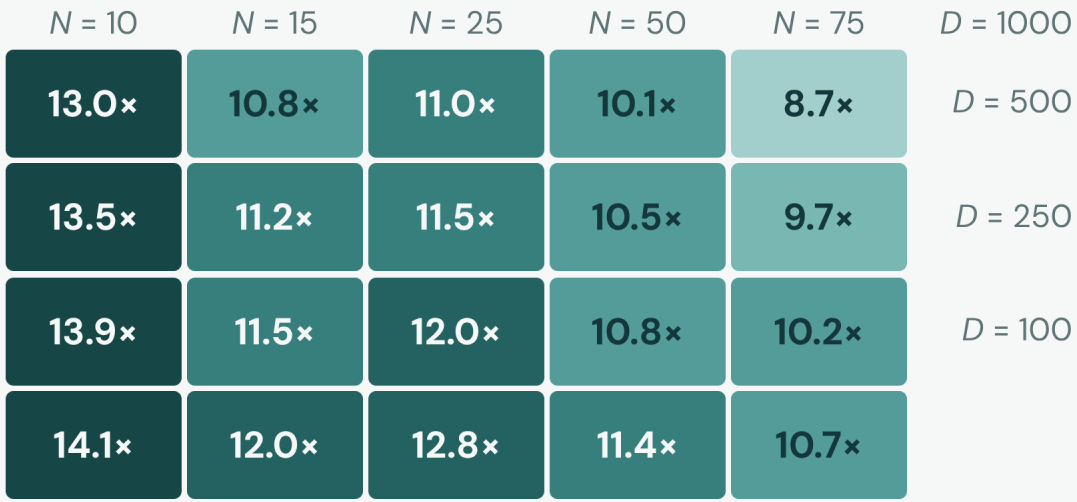
GPUS

NVIDIA A100 · H100 · H200 · AMD MI250X · MI300X

Same results: output identical to hsmmlearn in every tested configuration · FP64 everywhere

EVALUATION · SINGLE CORE

Single core: up to 14× faster



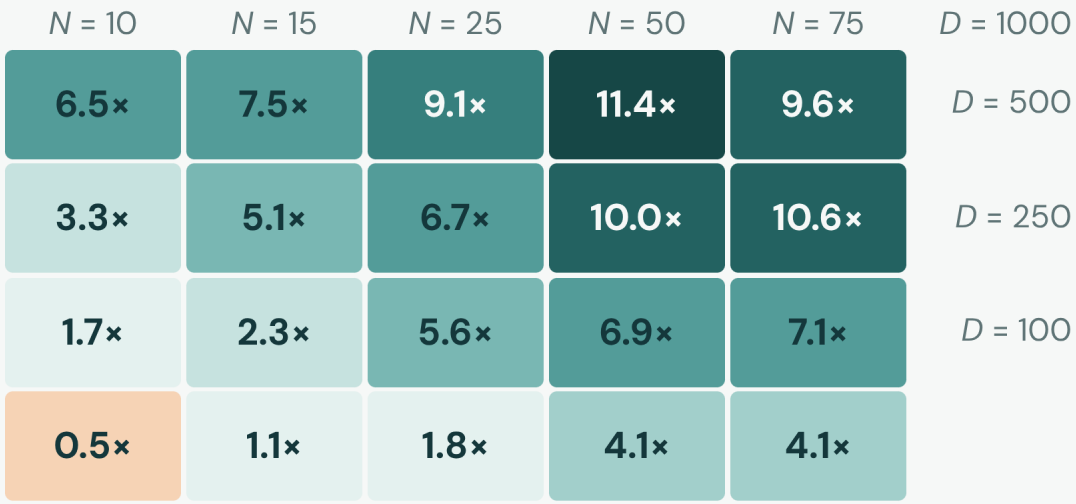
Why: 36× fewer instructions

LIKWID metric	Base-1C	Tens-1C
Instructions	610B	16.9B
Vectorization	< 0.001%	30.9%
Bad speculation	6.9%	0.46%
DRAM volume	2.63 GB	1.42 GB

Even plain NumPy (Tens-Py) is **4.5×** faster than the C++ baseline.

EVALUATION · MULTI-CORE

Multi-core: parallel over *N*·*D*, not just *N*



Tens-MC speedup over Base-MC · Xeon 8480+, 112 cores · *T* = 100k

1× 11× below 1×: slower

Why it scales

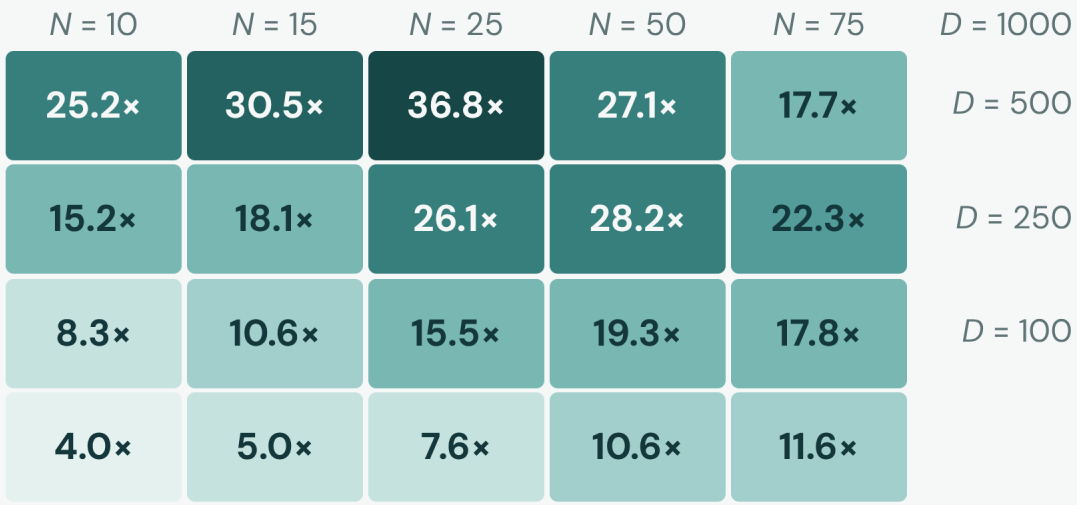
Base-MC can split work only over the *N* target states; Tens-MC exposes *N*·*D* independent tasks.

LIKWID metric	Base-MC	Tens-MC
Instructions	21.9B	9.9B
Vectorization	0.003%	80.4%
Bad speculation	6.25%	1.16%
DRAM volume	4.59 GB	1.43 GB

Short runs (*T* = 10k) can't amortize setup and barriers: down to 0.1×. The overhead fades on longer sequences.

EVALUATION · GPU

GPU: up to 54× over the multi-core baseline



Tens-GPU on H100 over Base-MC on Xeon 8480+ (112 cores) · T = 100k

4× 37×

54×

at T = 1M (N = 15, D = 500): 5.7 min for Base-MC on the 112-core Xeon, 6.4 s on one H100

5–12×

from MI250X to H200, over the fastest CPU baseline (Grace) · T = 1M, N = 50, D = 1,000

570×

over the sequential baseline (Base-1C on Grace): 3.8 h become 24 s on one H200 · T = 1M, N = 50

EVALUATION · ENERGY AND SCALE

Less energy, and new problem sizes

ENERGY · N = 50, T = 10K

2%

of the sequential baseline's energy, for Tens-GPU on MI250X (D = 100)

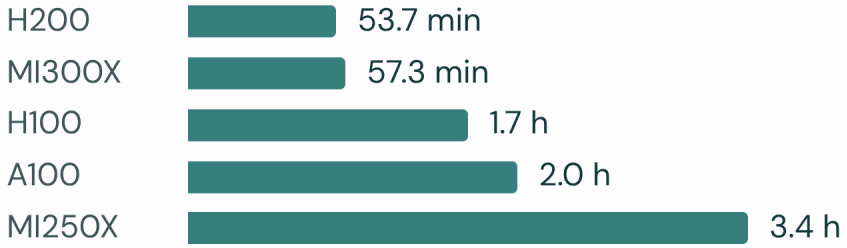
Tens-1C: about 10× less energy than Base-1C

Tens-MC: 525.5 J vs 1.5 kJ for Base-MC (D = 1,000)

STRESS TEST · N = 100, D = 10,000, T = 10M

53.7 min

on one H200, vs. an estimated 148 days for the sequential baseline



CONCLUSION

Takeaways

1. **From sequential to tensor formulation**
   HSMM Viterbi as broadcasted products plus one argmax per slice, with the same output.

2. **Faster than before, first GPU version**
   Up to 14× on one core, over 200× multi-core and over 570× on GPU, with as little as 2% of the energy.

3. **A new scale, opening new possibilities**
   A decode estimated at 148 days on one core runs in under an hour on an H200.



tensor-hsmm

github.com/HLC-Lab/tensor-hsmm

Thank you! Questions?

piarulli@di.uniroma1.it