GPU-accelerated homology search with MMseqs2

Felix Kallenborn[1,*], Alejandro Chacon[2,*], Christian Hundt[2], Hassan Sirelkhatim[2], Kieran Didi[2], Sooyoung Cha[3,4], Christian Dallago[2,✉], Milot Mirdita[3,✉], Bertil Schmidt[1,✉], and Martin Steinegger[3,4,5,6,✉]

Sensitive search of rapidly growing protein sequence databases for evolutionary information requires continual acceleration. This is achieved by innovating algorithms to filter sequences, or perform alignments. Here, we present GPU-optimized algorithms for gapless filtering, achieving up to 100 TCUPS across eight GPUs, and gapped alignment using protein profiles. Implemented in MMseqs2-GPU, they result in 20× faster and 71× cheaper search on a NVIDIA L40S GPU compared to MMseqs2 k-mer on a 128-core CPU. In ColabFold, they accelerate structure prediction 23× at matching accuracy to AlphaFold2, and in Foldseek they accelerate protein structure alignment by up to 27×. MMseqs2-GPU is open-source software available for CUDA devices at mmseqs.com.

Contact: ***cdallago@nvidia.com, mmirdit@snu.ac.kr, bertil.schmidt@uni-mainz.de, martin.steinegger@snu.ac.kr***

Many advances in biology have been enabled by computational tools to retrieve evolutionary related sequences (homologs) from reference databases (1–4). Building on the sequence-based protein homology paradigm (5, 6), these tools detect homologs to an input query among millions to billions of reference entries by searching for similar amino-acid sequences. Over the decades, homology search was critical for inference of protein properties (7–9) such as secondary structure prediction (10), detecting direct coupling between protein residue pairs (11), and tertiary structure prediction, long-considered a grand challenge in biology (12). Specifically, remote homologs have been shown to be pivotal as input to contemporary deep learning methods like AlphaFold2 and others (13–15) to predict accurate structures (16–18).

To retrieve remote homologs, sensitive tools to detect pairwise similarity between a query and reference sequences in a database are required. Theoretically, high sensitivity can be achieved by applying the dynamic programming-based, *gapped* Smith-Waterman-Gotoh algorithm (19, 20) to find the optimal path (alignment) (21) in each query-reference alignment-matrix. However, the ever-growing size of reference sequence databases (17) renders this exhaustive approach impractically slow. As a result, heuristic-based methods like BLAST (1), PSI-BLAST (22), MMseqs2 (4), and DIAMOND (3) incorporate prefiltering techniques to prune the majority of dissimilar sequences before executing the computationally expensive gapped computation. This is typically done by employing a seed-and-extend strategy, in which short k-mer words ("seeds") are indexed and matched, followed by their extension to *gapped* alignments. Sensitive aligners like HMMER (2) and HHblits (23) instead apply a simplified dynamic programming approach, which scores all gap-free paths (strict diagonals) of the alignment-matrix between sequence pairs to find the highest scoring ungapped match. Unlike k-mer-based methods, being a lower-bound approximation of a gapped alignment, gapless alignments result in a score for all pairs at the cost of computational efficiency.

Several approaches were explored to achieve higher execution speed regardless of heuristic, like central processing unit (CPU) specific instructions and parallelization, or employing specialized hardware like Field-Programmable Gate Arrays (FPGAs; 24), and Graphics Processing Units (GPUs; 25, 26).

Here, we present two novel GPU-accelerated algorithms (Fig. 1) for fast gapless prefiltering and gapped alignment using position-specific scoring matrices (PSSMs; 27). Integrated in MMseqs2, they achieve speed comparable to fast k-mer-based approaches while maintaining high sensitivity and delivering scores for every sequence pair. We confirm these findings through two benchmarks, one focusing on homology retrieval, in the best case achieving six-fold speed-up compared to next best CPU-based method BLAST, and on query-centered MSA-generation for structure prediction, achieving a 23-fold speed-up compared to AlphaFold2.

Previous accelerator-based approaches for gapless prefiltering achieved performance of up to 1.7 TCUPS on an Alveo U50 FPGA (24) and 0.4 TCUPS on a K40 GPU (25) (Methods, "Cell updates per second"). Our GPU-accelerated algorithms for gapless prefiltering and gapped alignment leverage the high parallelism of GPUs. The gapless prefilter, mapping query PSSMs to columns and reference sequences to rows in a matrix, processes each matrix row in parallel, utilizing shared GPU memory to optimize access to PSSMs (Figure 1c) and 16-bit floating-point numbers packed in a 32-bit word (half2) to maximize throughput. This achieved 13.5 TCUPS on a single L40S GPU and 102 TCUPS on a server with eight L40S GPUs, outperforming previous acceleration methods by one-to-two orders of magnitude (24, 25). Additionally, we incorporated a modified CUDASW++4.0 (28) that operates on PSSMs for gapped alignment, employing a wavefront pattern to efficiently handle dynamic programming dependencies.

Prior to evaluating homology search speed, we sought parameters allowing to reach similar sensitivity (measured on the MMseqs2 benchmark (4)) across homology search methods. This resulted in all methods reaching ROC1 between 0.40-0.41, specifically setting DIAMOND with the `--ultra-sensitive` option and MMseqs2 k-mer with the `-s 8.5` setting. Furthermore, to examine the ability of MMseqs2 GPU to increase sensitivity, we investigated iterative profile searches, where initial hits are converted into PSSMs that en-

[1]Department of Computer Science, Johannes Gutenberg University Mainz, Mainz, Germany. [2]NVIDIA Corp., Santa Clara, USA. [3]School of Biological Sciences, Seoul National University, Seoul, Republic of Korea. [4]Interdisciplinary Program in Bioinformatics, Seoul National University, Seoul, Republic of Korea. [5]Institute of Molecular Biology and Genetics, Seoul National University, Seoul, Republic of Korea. [6]Artificial Intelligence Institute, Seoul National University, Seoul, Republic of Korea. * These two authors contributed equally.

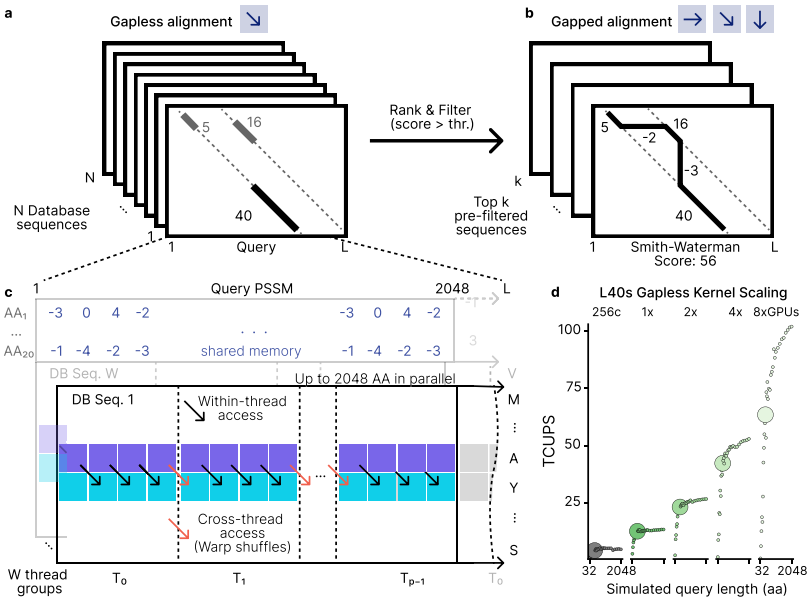

Fig. 1. MMseqs2-GPU workflow and gapless alignment performance. (**a**) The gapless alignment involves scanning database sequences against the query, followed by ranking and filtering based on alignment scores. Only the top database sequences with scores above a threshold proceed to (**b**) gapped alignment using the Smith-Waterman-Gotoh algorithm. In the GPU-optimized gapless alignment (**c**), the query profile is split into segments of up to 2048 amino acids, which are loaded into shared memory for fast access across several thread groups, avoiding slower global memory access. Each thread computes alignment scores independently within its segment, while cross-thread warp shuffles enable efficient data sharing for diagonal dependencies. (**d**) The tera cell updates per second (TCUPS) performance for different query lengths of gapless alignments reach up-to 100 TCUPS on 8 GPUs outperforming 128/256 core/thread CPUs.

code probabilities for each residue in a sequence. MMseqs2-GPU reached ROC1 of 0.612 and 0.669 at two and three iterations, respectively. This was slightly higher than PSI-BLAST, reaching 0.574 and 0.591, and slightly lower than JackHMMER, reaching 0.614 and 0.685 at two and three iterations, respectively (Methods, "Sensitivity"; Supplementary Data 1, "Sensitivity benchmark").

We then benchmarked speed for homology search focusing on two common scenarios: a single query protein against a target database of roughly 30M sequences (single batch), common for scientists working on a protein system, and a set of query proteins against the same 30M target database (batch 6370), common for proteome analysis. MMseqs2-GPU on a single NVIDIA L40S PCIe outperformed the commonly-used JackHMMER, being 180-200 times faster (Fig. 2; Supplementary Data 1, "MSA tool comparison"). In single-batch mode, MMseqs2-GPU was six times faster than the second-fastest method BLAST (Fig. 2a, left), while at batch size 100 it was twice as fast as the second-fastest in that setting, MMseqs2 k-mer (Fig. 2a, middle). At batch size 6370, MMseqs2 k-mer on a sizable 128 Cores CPU is about 2.5x faster than MMseqs2-GPU on a single L40S, however on a multi-GPU system, MMseqs2-GPU takes the lead at 2x the speed of MMseqs2 k-mer (Fig. 2a, right). Testing MMseqs2-GPU on other NVIDIA GPUs, A100 PCIe and H100 PCIe, it exceeded CPU-based methods at batch sizes one and 100, but resulted slower than MMseqs2 k-mer at batch size 6370. On an energy efficient NVIDIA L4 PCIe, MMseqs2-GPU exceeded CPU-based methods at batch size one (Materials, "Speed"; Supplementary Data 1, "MSA tool comparison").

Diamond benefited from batching more than MMseqs2-GPU (Fig. 2a, slopes from left to right). We further investigated Diamond's batching behavior by using larger batch sizes (10,000 and 100,000 sequences). We found that Diamond's average per-query speed does not improve and remains around 0.42s, which was measured on batch 6370. At batch size 6370, MMseqs2-GPU and MMseqs2 k-mer reached 0.37 seconds per query, making them overall faster than Diamond (Materials, "Diamond scaling"; Supplementary Data 1, "MSA tool comparison").

MMseqs2-GPU scales in multi-GPU systems either by distributing the target database across GPUs (**Fig. 2a**), or linearly, by sharding the query set across GPUs, albeit with database replication on each GPU. In practice, this means that if the database can be fit in a single GPU, users gain speedup proportional to the number of GPUs. If instead a database cannot fit a single GPU, but may fit multiple, users can gain up to five-fold speedups.

Considering the single-batch scenario and MMseqs2 k-mer as baseline, we investigated cloud compute cost leveraging AWS EC2 cost estimates. MMseqs2-GPU using one L40S was 20 times faster and 71 times cheaper than MMseqs2 k-mer, while JackHMMER was about nine times slower and more expensive, making MMseqs2-GPU on one L40S the fastest and cheapest cloud solution for single-batch execution (Fig. 2b).

We also investigated energy consumption in the single-batch scenario. MMseqs2-GPU was 80 and 2 times more energy efficient than JackHMMER and MMseqs2 k-mer, respectively (Supplementary Data 1, "Power consumption").

Over time, MMseqs2's k-mer-based prefiltering was optimized to handle smaller input batch sizes, particularly for web servers (29) and to generate MSA input for ColabFold (30). While scaling target databases improved performance, it required substantial RAM (up to 2TB) for fast MSA generation within ColabFold. MMseqs2-GPU addresses this issue, requiring only one byte per residue in the target database, which can further be reduced by clustered searches against representative sequences followed by member realignment.

Accurate computational protein structure prediction has become ubiquitous since the release of AlphaFold2 (13) and RoseTTAFold (14) in 2021. Variations of these methods using alternative representations were introduced for higher throughput (31), however methods that leverage query-centered MSAs

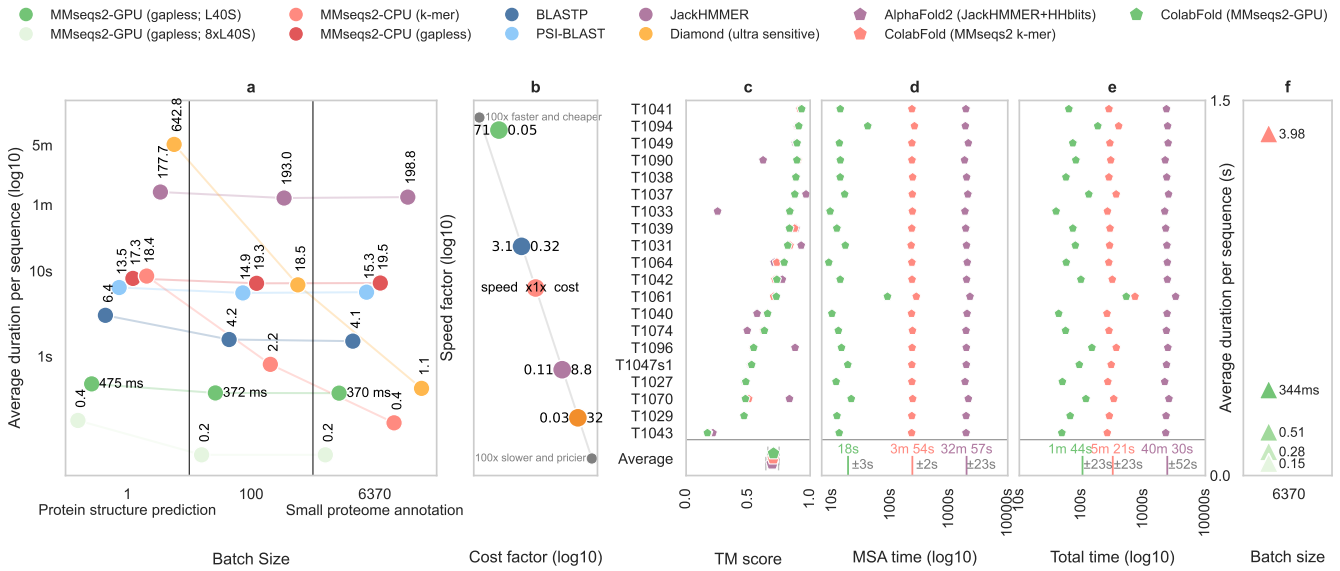

Fig. 2. MMseqs2-GPU runtimes for homology search. (a) In single-batch processing, MMseqs2-GPU (dark green) on a single L40S GPU is ∼six times faster than the second best BLAST (dark blue), and ∼178 times faster than JackHMMER (purple) on average for 6370 queries against a 30M target database. MMseqs2-GPU can split the target database across GPUs to enable larger database searches, also achieving a speedup compared to single GPU execution (up to five times faster on 8x L40S compared to one, bright green vs. dark green). When present, JackHMMER values refer to measuring 10% of the queries, as execution time for the whole query set was prohibitively inefficient. Similarly, execution time for Diamond ultra sensitive in the single-batch setting refers to 10% of the query set. **(b)** For single-batch processing, MMseqs2-GPU delivers the fastest speed at lowest cloud compute cost, being 71 times faster than MMseqs2 k-mer at 0.05 times the cost. **(c-e)** Faster folding speeds at no accuracy cost. On 20 CASP14 targets, ColabFold leveraging MMseqs2-GPU (green) results 3 and 23 times faster than ColabFold using MMseqs k-mer (orange) or AlphaFold2 using JackHMMER (violet), respectively. All methods reach similar TM score (∼0.70±0.05).**(f)** Foldseek built on MMseqs2-GPU reaches top performance at four times the speed of standard Foldseek even at proteome batch size (6370) due to its required higher sensitivity. Scaling on eight L40S GPUs leads to ∼seven times speedup compared one L40S, and ∼27 times speedup compared to 128 CPU cores.

remain the most accurate (16). We compared the homology search and structure prediction steps for the single batch scenario common for protein prediction servers for three variants: the canonical setting of AlphaFold2 leveraging HHblits (23) and JackHMMER (2), and ColabFold leveraging either MMseqs2 k-mer or MMseqs2-GPU on one L40S. We compare these pipelines for end-to-end structure prediction of the 20 free modelling (FM) sequences from CASP14 (32).

ColabFold utilizing MMseqs2-GPU was fastest, being three times faster than ColabFold MMseqs2 k-mer and 23 times faster than AlphaFold2 leveraging HHBlits and JackHMMER (Fig. 2b). These speed differences had no affect on prediction accuracy, with all three configurations achieving TM-score around 0.70±0.5 and LDDT around 0.76±0.4 (Fig. 2c; Supplementary Data 1, "Folding tool comparison").

MMseqs2 serves as the backbone to various methods through its modular architecture, such as the protein structure aligner Foldseek (33). Foldseek requires high sensitivity to obtain good structure alignment, reducing the utility of fast k-mer-based double-diagonal search. We measured Foldseek's search speed for CPU (Foldseek k-mer) and GPU (Foldseek-GPU) on 6370 protein structure queries sampled from the AlphaFold Database clustered at 50% structure identity, searching them against the same reference database. Foldseek-GPU on a single L40S outperformed Foldseek k-mer using 128 CPU-cores approximately by a factor of 4, and was ∼27 times faster with eight L40S GPUs (Fig. 2f). Repeating the Foldseek SCOPe-based sensitivity benchmark with Foldseek-GPU yielded modest increases in sensitivity across Family (0.874 vs. 0.861), Superfamily (0.492 vs. 0.487), and Fold recognition (0.109 vs. 0.107) compared to Foldseek k-mer.

MMseqs2-GPU ranks as the fastest evolutionary search tool across various experiments. Using one NVIDIA L40S, it results faster in most settings, being surpassed by MMseqs2 k-mer at larger batch sizes on a large CPU server. However, through seamless multi-GPU parallelization, it overtakes even ultra-fast, word-based MMseqs2 k-mer running on a 256 threaded CPU. Unlike word-based search methods (1, 3, 4), MMseqs2-GPU's sensitive gapless pre-filter, similar to JackHMMER, does not allow trading sensitivity for speed. However, reaching high-sensitivity through iterative profile searches is essential for detecting remote homologs, which is critical for tasks like structure prediction (16), making it an excellent choice for these workflows. At parity of sensitivity, it is the cheapest option according to cloud cost estimates for single batch execution.

One of the challenges of running CPU-based homology searches efficiently is the reliance on high-performance computing resources, often requiring large-scale systems with extensive RAM and many threads, which are inaccessible to most researchers. For example, our experiments ran on a server with 1TB RAM and 256 threads — resources typically out of reach for most academics. MMseqs2-GPU overcomes this limitation by offloading compute-intensive tasks to the GPU. Even on cost-effective GPUs like the NVIDIA L4, which are available through platforms such as Google Colab Pro, MMseqs2-GPU accelerates single protein searches over CPU-based methods (Supplementary Data 1, "MSA tool

comparison"), offering a ten-fold speed increase over JackHMMER in searching UniRef90 (Supplementary Data 1, "Colab MSA benchmark"). While GPU memory can be a limitation, MMseqs2-GPU mediates this with three strategies: efficient database streaming, partitioning, and clustered searches.

Protein structure prediction is usually ran on GPU-equipped compute resources, with MSA computation on the CPU consuming the majority of runtime (83% of total execution time; see Supplementary Data 1, "Folding tool comparison"), resulting in idle GPU time. In contrast, ColabFold paired with MMseqs2-GPU runs nearly all computation on GPU, inducing better resource utilization and, crucially, reducing MSA generation to only 18% of total execution time.

We demonstrated that MMseqs2-GPU reduces the end-to-end execution time of protein folding without compromising accuracy, enabling higher throughput at lower cost (Fig. 2). Many bioinformatics applications rely on homology searches and could benefit immediately. For example, orthology inference tools that already use MMseqs2 can seamlessly switch to the GPU backend (34, 35), as well as structural protein search methods like Foldseek (33). Especially workflows that already leverage GPUs but are impractically slow at inference, like retrieval augmented protein language models (36), stand to benefit. Therefore, MMseqs2-GPU broadens access to rapid and affordable homology searches.

References

1. Altschul, S. F., Gish, W., Miller, W., Myers, E. W. & Lipman, D. J. Basic local alignment search tool. *Journal of Molecular Biology* **215**, 403–410 (1990).
2. Eddy, S. R. Accelerated profile HMM searches. *PLoS Computational Biology* **7**, e1002195 (2011).
3. Buchfink, B., Reuter, K. & Drost, H.-G. Sensitive protein alignments at tree-of-life scale using DIAMOND. *Nature Methods* **18**, 366–368 (2021).
4. Steinegger, M. & Söding, J. MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. *Nature Biotechnology* **35**, 1026–1028 (2017).
5. Zuckerkandl, E. & Pauling, L. in *Evolutionary divergence and convergence in proteins* (eds Bryson, V. & Vogel, H. J.) *Evolving genes and proteins* 97–166 (Elsevier, 1965).
6. Sander, C. & Schneider, R. Database of homology-derived protein structures and the structural meaning of sequence alignment. *Proteins: Structure, Function, and Bioinformatics* **9**, 56–68 (1991).
7. Watson, J. D., Laskowski, R. A. & Thornton, J. M. Predicting protein function from sequence and structural data. *Current Opinion in Structural Biology* **15**, 275–284 (2005).
8. Lee, D., Redfern, O. & Orengo, C. Predicting protein function from sequence and structure. *Nature Reviews Molecular Cell Biology* **8**, 995–1005 (2007).
9. Radivojac, P. *et al.* A large-scale evaluation of computational protein function prediction. *Nature Methods* **10**, 221–227 (2013).
10. Rost, B. & Sander, C. Prediction of protein secondary structure at better than 70% accuracy. *Journal of Molecular Biology* **232**, 584–599 (1993).
11. Morcos, F. *et al.* Direct-coupling analysis of residue coevolution captures native contacts across many protein families. *Proceedings of the National Academy of Sciences* **108**, E1293–E1301 (2011).
12. Searls, D. B. in *Chapter 1 - Grand challenges in computational biology* (eds Salzberg, S. L., Searls, D. B. & Kasif, S.) *Computational Methods in Molecular Biology*, Vol. 32 of *New Comprehensive Biochemistry* 3–10 (Elsevier, 1998).
13. Jumper, J. *et al.* Highly accurate protein structure prediction with AlphaFold. *Nature* **596**, 583–589 (2021).
14. Baek, M. *et al.* Accurate prediction of protein structures and interactions using a three-track neural network. *Science* **373**, 871–876 (2021).
15. Abramson, J. *et al.* Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature* **630**, 493–500 (2024).
16. Kryshtafovych, A., Schwede, T., Topf, M., Fidelis, K. & Moult, J. Critical assessment of methods of protein structure prediction (CASP)—Round XV. *Proteins: Structure, Function, and Bioinformatics* **91**, 1539–1549 (2023).
17. Lee, S. *et al.* Petabase-Scale Homology Search for Structure Prediction. *Cold Spring Harbor Perspectives in Biology* **16**, a041465 (2024).
18. Zhang, J. *et al.* Computing the Human Interactome. *bioRxiv* 2024.10.01.615885 (2024).
19. Smith, T. F., Waterman, M. S. *et al.* Identification of common molecular subsequences. *Journal of Molecular Biology* **147**, 195–197 (1981).
20. Gotoh, O. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* **162**, 705–708 (1982).
21. Rognes, T. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics* **12**, 221 (2011).
22. Altschul, S. F. *et al.* Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research* **25**, 3389–3402 (1997).
23. Steinegger, M. *et al.* HH-suite3 for fast remote homology detection and deep protein annotation. *BMC Bioinformatics* **20**, 473 (2019).
24. Anderson, T. & Wheeler, T. J. An FPGA-based hardware accelerator supporting sensitive sequence homology filtering with profile hidden Markov models. *BMC Bioinformatics* **25**, 247 (2024).
25. Jiang, H. & Ganesan, N. CUDAMPF: a multi-tiered parallel framework for accelerating protein sequence search in HMMER on CUDA-enabled GPU. *BMC Bioinformatics* **17**, 106 (2016).
26. Selvitopi, O. *et al. Distributed Many-to-Many Protein Sequence Alignment using Sparse Matrices. SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–14 (IEEE, 2020). URL https://ieeexplore.ieee.org/document/9355305/.
27. Henikoff, S. & Henikoff, J. G. Position-based Sequence Weights. *Journal of Molecular Biology* **243**, 574–578 (1994).
28. Schmidt, B., Kallenborn, F., Chacon, A. & Hundt, C. CUDASW++ 4.0: ultra-fast GPU-based Smith–Waterman protein sequence database search. *BMC Bioinformatics* **25**, 342 (2024).
29. Mirdita, M., Steinegger, M. & Söding, J. MMseqs2 desktop and local web server app for fast, interactive sequence searches. *Bioinformatics* **35**, 2856–2858 (2019).
30. Mirdita, M. *et al.* ColabFold: making protein folding accessible to all. *Nature Methods* **19**, 679–682 (2022).
31. Rives, A. *et al.* Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proceedings of the National Academy of Sciences* **118**, e2016239118 (2021).
32. Kryshtafovych, A., Schwede, T., Topf, M., Fidelis, K. & Moult, J. Critical assessment of methods of protein structure prediction (CASP)—Round XIV. *Proteins: Structure, Function, and Bioinformatics* **89**, 1607–1617 (2021).
33. Van Kempen, M. *et al.* Fast and accurate protein structure search with Foldseek. *Nature Biotechnology* **42**, 243–246 (2024).
34. Emms, D. M. & Kelly, S. OrthoFinder: phylogenetic orthology inference for comparative genomics. *Genome Biology* **20**, 238 (2019).
35. Cosentino, S., Sriswasdi, S. & Iwasaki, W. SonicParanoid2: fast, accurate, and comprehensive orthology inference with machine learning and language models. *Genome Biology* **25**, 195 (2024).
36. Truong Jr, T. & Bepler, T. Poet: A generative model of protein families as sequences-of-sequences. *Advances in Neural Information Processing Systems* **36**, 77379–77415 (2023).

Online Methods

A. Background: fast sequence comparison principles

Prefiltering is an essential step in homology search to reduce the overall execution time by ranking all reference database sequences quickly, and filtering a subset against which the computationally more demanding Smith-Waterman-Gotoh can be run. As one of the earliest tools, BLAST (1) proposed index structures for initial seed finding with bi-directional gapless extension to match a given minimum score.

MMseqs2 (4) introduced a prefiltering approach based on double-consecutive k-mer matches on the same diagonal. Through this approach, all k-mers of a reference database are stored in a random-access-memory (RAM) based index structure for quick retrieval (hereafter *the index*). For single query searches at high sensitivity, MMseqs2 generates long lists of similar k-mers from the query sequence and matches them against the index to check the prefiltering criterion of two consecutive matches on one diagonal. While highly optimized, random accesses to the index results in poor cache-locality.

Diamond (3) generates lists of k-mers for both queries and references in order to sort and compare them co-linearly. Instead of similar k-mers, Diamond utilizes multiple spaced-k-mer patterns. Co-linear comparison results in improved cache-locality at the expense of indexing and sorting overhead.

In contrast to *word*-based prefiltering approaches, HMMER (2) and HHblits (23) implemented a more sensitive, albeit slower ranking technique by simplifying the Smith-Waterman-Gotoh algorithm to exclude *gaps* from the alignment. In essence, this allows finding the longest common subsequence (LCS) consisting of residue substitutions only between any pair of sequences, allowing for better resolution than word-based methods.

Modern hardware accelerators like GPUs lend themselves to highly parallel workflows through their high core count, albeit at lower operational complexity per core. Thus, the gapless prefiltering is well suited to exploit their capabilities, due reduced instruction count with little data-dependencies, while additionally avoiding branching and random memory accesses typically employed in k-mer index lookups.

B. MMseqs2-GPU algorithm and parallelization

Pairwise gapless alignments are computed between a query represented by the PSSM Q and each reference sequence. Q is constructed either through a single query sequence and a substitution matrix, or from a sequence profile, based on previous search results. Once computed, Q provides a score for placing any amino acid at any position along the length of the query. Next, our *gapless prefilter* computes a pairwise local alignment between the reference sequence $S = (s_1, ..., s_n)$ and the PSSM Q of length m by dynamic programming (DP), and populates a matrix M using the recurrence relation $M[i,j] = \max\{M([i-1,j-1] + Q[i,s_j], 0\}$ for all $1 \leq i \leq m, 1 \leq j \leq n$. Initialization is given by $M[i,0] = M[0,j] = 0$. Algorithm 1 outputs the maximum value in M, which represents the score of the optimal local alignment without gaps (a.k.a., *gapless*) between S and Q. The top k sequences passing an inclusion threshold are then passed to a Smith-Waterman-Gotoh algorithm operating on profiles. Both of these algorithmic steps are executed on GPU.

Input : Protein sequence $S[]$ of size n, PSSM $Q[]$ of size $m \times |\Sigma|$
Output : Score

```
Score ← 0;
for i ← 0 to m do
  | M[i,0] ← 0
end
for j ← 0 to n do
  | M[0,j] ← 0
end
for i ← 1 to m do
  | for j ← 1 to n do
  |   | M(i,j) ← max{M(i−1,j−1) + Q[i,s_j], 0};
  |   | Score ← max{Score, M(i,j)};
  | end
end
```

Algorithm 1: Gapless Prefilter Score Computation

B.1. CPU-SIMD gapless prefilter. MMseqs2 offers a gapless prefilter accelerated on CPUs through Single Instruction Multiple Data (SIMD) instructions, through the `ungappedprefilter` module first introduced in MMseqs2 6.f5a1c. In this work, we describe the module, its integration into the `search` workflow with `--prefilter-mode 1` and its extension to incorporate soft-masking of low-complexity regions with tantan (37)–in addition to the GPU integration described below.

The prefiltering process begins by preparing a striped query profile (38) on a single-thread, and finally utilizes all available CPU-cores to linearly compare all reference sequences on disk. During prefiltering, all soft-masked target database amino-acids are represented as the unknown `X` residue.

At its core, the gapless alignment follows Farrar's (38) striped Smith-Waterman-Gotoh first used in HMMER (2), which is adjusted to compute only the gapless diagonal. It avoids affine gap computation and requires only five SIMD operations to update a striped query segment to retrieve the maximum gapless diagonal score.

The algorithm uses 8-bit integers to represent alignment scores to maximize the parallel SIMD register usage. Scores that exceed the 8-bit limit of 255 are clamped to 255, which is significant and indicates a strong potential match. Such matches are then aligned using full Smith-Waterman-Gotoh. Depending on the supported instruction of the CPU this gapless implementation uses 128-bit (SSE2) or 256-bit (AVX2) vector size. We also evaluated AVX512 but found no performance benefit and did not implement it. In addition, our implementation exploits multi-core CPUs by computing different target sequences independently using multi-threading.

B.2. GPU-accelerated gapless prefilter.

Core algorithm. We present a novel implementation of a gapless prefilter on GPUs, designed to leverage the simplified

dependency scheme inherent in gapless alignments for high parallelism and performance. Unlike the Smith-Waterman-Gotoh algorithm, where each cell in the DP matrix depends on its top, left, and diagonal neighbors, gapless alignment reduces this dependency to only the diagonal neighbor (Fig. 1). This crucial simplification enables parallel computation across all cells within the same row, potentially enhancing processing speed. However, to realize this potential the design of an appropriate parallelization scheme is required that optimizes necessary memory accesses and inter-thread communication while efficiently handling highly variable sequence lengths.

Memory accesses and data types. To optimize memory accesses and performance, the query PSSM is stored in shared memory, facilitating fast, multi-threaded access (Fig. 1c). Meanwhile, reference sequence residues, which map to the y-axis of the DP matrix, are stored in global memory, requiring one byte per residue. Our calculations employ 16-bit floating-point numbers or 16-bit integers, supported by the DPX instructions available with NVIDIA's Hopper GPU architecture, packed into a 32-bit word using half2 or s16x2 data types.

Thread grouping and tile optimization. Drawing inspiration from CUDASW++4.0 (28), our implementation assigns each gapless alignment task to a thread group, with typical sizes of 4, 8, or 16 threads. This configuration allows multiple thread groups to handle different alignments concurrently. The DP matrix is processed row-by-row, with each row partitioned among threads that are responsible for up to 128 cells each. To optimize performance across various query lengths, we employ different tile sizes, which are determined by the product of the thread group size and the number of columns processed per thread. They are realized as template parameters, with the group size constrained to be a divisor of 32, and half the number of columns being a multiple of 4. Optimal tile size configurations for different query lengths were identified via a separate grid search program, minimizing out-of-bounds computations when the query length is less than the tile size. As the processing begins, reference sequence residues are loaded from global memory, and each thread accesses its corresponding values from the scoring profile. To boost throughput, we designed the memory access pattern so that each thread loads eight consecutive 16-bit scores from shared memory in a single instruction.

Shared memory bank conflict mitigation. We further optimized shared memory access for thread groups of size four to mitigate bank conflicts. Shared memory is organized into 32 four-byte banks, and we utilize two-byte values in the PSSM, meaning two columns are packed per four-byte word. The PSSM is arranged in shared memory such that the i-th four-byte column maps to memory bank i mod 32.
Bank conflicts occur when multiple accesses within the same 128-byte wavefront target the same bank but different addresses, leading to serialization. This can arise when multiple thread groups within a warp access the same PSSM columns but different rows (corresponding to different residues in the reference sequence). For group sizes of eight or larger, this is not an issue, as each group loads $8 * 16 = 128$ bytes, thus fitting a single wavefront. However, with a group size of four, each group loads only $4 * 16 = 64$ bytes. This allows a second group to load from a different PSSM row (but the same columns) within the same wavefront, resulting in a two-way bank conflict. To resolve this, we employ two copies of the PSSM in shared memory. These copies are stored such that the i-th thread group consistently uses memory banks 0-15 if i is even, and banks 16-31 if i is odd. This ensures that accesses from different groups within the same wavefront are directed to distinct memory banks, eliminating the bank conflict.

Per-cell computation and warp shuffles. Once the scores are loaded, threads perform computations for their assigned columns. Each thread performs the following operations for each cell: (1) adds the score from the scoring profile to the value from the previous diagonal cell, (2) sets the result to zero if it is negative, and (3) updates the local maximum score if the current cell's value exceeds it. To facilitate data communication between threads within a group, particularly for diagonal dependencies, we use warp-shuffle operations. Since the group size is at most 32, all threads within a group belong to the same warp. Warp shuffles allow for efficient register-based data exchange between threads, avoiding slower shared memory access with explicit synchronization, or the even slower use of global memory.

Data permutation and vectorization. To double computational throughput, we utilize hardware instructions capable of processing two 16-bit values packed into a 32-bit word. However, direct packing of neighboring matrix columns into a single 32-bit word introduces dependencies that hinder independent processing. To address this challenge, we apply a data permutation technique that rearranges columns within a thread, aligning diagonal dependencies directly within the packed data. For example, in cases where a thread handles 32 columns, we pack columns (0, 16), (1, 17), (2, 18), and so forth. This arrangement ensures that each pair of packed columns corresponds to cells that are diagonally adjacent in the DP matrix. As a result, the dependency between packed values is restricted to the preceding value from the previous row, enabling full exploitation of vectorized operations for computational efficiency.

Long sequence handling. This strategy allows matrix tiles with up to 2,048 columns to be processed efficiently with a thread group of size 16. For protein sequences exceeding 2,048 residues, the profile matrix is divided into multiple tiles processed sequentially. In such cases, the last column of each tile is stored in global memory to serve as the starting point for the first column of the next tile.

Multi-GPU Parallelization. To maximize performance on systems with multiple GPUs, we distribute the reference database across the available GPUs. Specifically, the target database sequences are split into chunks, and each GPU processes a separate chunk against the query sequence. The results from each GPU are then combined to produce the final result list. This parallelization strategy reduces execution time and allows

for processing larger databases by leveraging the combined available GPU memory.

B.3. GPU-accelerated Smith-Waterman-Gotoh. In addition to the gapless prefilter on GPU, we implemented a version of Smith-Waterman-Gotoh with affine gap penalties operating on protein profiles (PSSMs) as a modification of CUDASW++4.0 (28) to align reference sequences to the same query profiles used in the prefilter. As described in the previous section, this required to transpose the computed DP matrix to place the profile along the x-axis, and leveraged the same parallelization strategies. Matrix tiles are processed by thread groups of size 4, 8, 16, or 32 using in-register computations with 32-bit capacity to avoid overflows.

In contrast to the prefilter algorithm, the Smith-Waterman-Gotoh algorithm does not allow for threads in the same group to operate on the same row in parallel since cells depend on their left neighbor. Consequently, a wavefront pattern is used to have threads work on different rows along the minor diagonals.

B.4. Cell updates per second. Speed of DP algorithms is typically reported by converting runtime into the number of DP matrix cell updates that are performed per second; e.g., TCUPS (Trillions of Cell Updates Per Second) as $\mathrm{TCUPS} = (\sum_i m_i \times n_i)/(t \times 10^{12})$, where t is the runtime in seconds and m_i and n_i are the lengths of the aligned sequences.

To measure TCUPS performance for MMseqs2-GPU (Fig. 1d), we performed a synthetic benchmark generating sequences of equal lengths for several lengths. For each possible DP matrix tile size l, a randomly generated query of length l was aligned to a database of five million randomly generated sequences of length l. Runtime was then converted to TCUPS. This synthetic benchmarks explores the best case scenario by employing uniform length, as this avoids reduced hardware utilization caused by adjacent CUDA thread groups processing different reference sequence lengths. For real-world database searches, the variable length effect is minimized by sorting the database sequences by length in ascending order.

We now analyze if our approach is able to effectively remove overheads occurred by memory accesses using half2 arithmetic as an example by modeling the *theoretical peak performance* (TPP) of the utilized GPU hardware as:

$$\mathrm{TPP} = \frac{\#\mathrm{SMs} \times \mathrm{Throughput_per_Instruction} \times \mathrm{Clock}}{\mathrm{Cycles_per_cell_update}} \quad \textbf{(1)}$$

Throughput_per_instruction refers to the number of results per clock cycle per SM of native arithmetic instructions on the considered hardware. A corresponding table for devices of various compute capabilities is provided in the CUDA documentation [1]. Cycles_per_cell_update in Eq. 1 models the maximum attainable performance constrained by the algorithm structure and the specifications of the architecture. In our case it refers to the theoretical minimal number of clock cycles needed by an individual SM of the utilized GPU to calculate one DP matrix cell in M. This value can be initially determined by the inner loop in Algorithm 1 as 3 (i.e. 2 max instructions and 1 add instruction). The L40S GPU used in our experiments further enables dual ports for max and add floating point operations which reduces Cycles_per_cell_update to 2 (when using single-precision arithmetic) and to 1 (when using half2 arithmetic). Based on the specified Throughput_per_instruction of 64 for both max and add instructions on the L40S (compute capability 8.9), we can use Eq. 1 to calculate the TPP for half2 arithmetic on the L40S as follows:

$$\mathrm{TPP(L40S)} = \frac{142 \times 64 \times 2.56\ \mathrm{GHz}}{1} = 23.2\ \mathrm{TCUPS}$$

Note that we disregard the lookup operation to the PSSM, any value to register movements such as warp shuffles, and any data transfers in our TPP calcuation. Our synthetic benchmark on a single L40S acheives a performance of up to 13.5 TCUPS. Our approach is able to achieve an efficiency of 58% on an L40S GPU architectures, which shows that our various optimizations such as warp-shuffles and PSSM lookups from shared memory are able to effectively transform the problem to compute-bound and minimize overheads from memory accesses.

C. MMseqs2-GPU Workflow

The MMseqs2-GPU workflow starts with query and reference sets residing in CPU RAM. For the first step, i.e. prefiltering, query profiles are transferred to the GPU and permuted to enable efficient CUDA shared memory accesses during lookup. Additionally, the reference set is partitioned into smaller batches, which are transferred to the GPU. Throughout the prefiltering step, gapless alignment scores are stored for subsequent sorting. In the next stage, after all reference database batches have been processed, *gapped*, i.e. Smith-Waterman-Gotoh alignment scores are computed for top reference sequences satisfying inclusion thresholds using the same partitioned approach used for gapless alignment. Finally, the prefiltered reference sequences and their corresponding alignment scores are transferred back to the CPU.

We realized several optimizations in the back-end of the workflow to increase processing speed. To reduce communication latency between CPU and GPU, database batch processing is pipelined using CUDA streams, overlapping the data transfer of batch $i+1$ with the processing of batch i. Notably, this approach allows us to efficiently handle databases larger than the available GPU memory. To maximize GPU memory utilization and minimize data transfer, the workflow caches as many reference database batches as fit in GPU memory, and re-uses them for subsequent queries. This GPU caching behavior is further enhanced by using a dedicated GPU server process responsible for database caching and alignment computation. The communication between GPU server and client processes is done through Linux shared-memory.

D. Hardware setup

With the exception of benchmarks executed on Google Colab compute, the same base system hardware setup was used.

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions

Specifically, the system featured two AMD EPYC 7742 64-core (2.25 GHz; Thermal Design Power (TDP) of 225 Watt) CPUs (effectively 128 physical cores running 256 logical threads and TDP of 450 Watt), with 1TB of DDR4 RAM and 2TB NVMe Intel SSDPE storage.
For the GPU benchmarks, the base system configuration additionally included either an L4, A100, L40S, or H100 PCIe NVIDIA GPU, which are set at TDP 72, 300, 350 and 350 Watts, respectively.

E. Benchmarks

E.1. Sensitivity. The same approach and datasets described in MMseqs2 (4) were used to conduct the sensitivity benchmark. This benchmark involved annotating full-length UniProt sequences with structural domain annotations from SCOP (39), designating 6,370 sequences as queries and 3.4 million as reference sequences. The full-length query sequences included disordered, low-complexity, and repeat regions, which are known to cause false-positive matches, especially in iterative profile searches. Additionally, 27 million reversed UniProt sequences were included as reference sequences (resulting in a total of 30,430,281 reference sequences).
Like in previous work, true-positive matches are defined as those with annotated SCOP domains from the same family, while false positives match reversed sequences or sequences with SCOP domains from different folds. The sensitivity of a single search is measured by the area under the curve (AUC) before the first false-positive match, indicating the fraction of true-positive matches found with a better E-value than the first false-positive match.

E.2. Speed. For speed benchmarks, the same query and reference sequences as for the sensitivity benchmarks were used, and methods' parameters were set to reach comparable sensitivity where possible. Specifically, we ran the following methods:

- **JackHMMER v3.4** (2): at 1 iteration (equivalent to a phmmer search), with an E-value cutoff of $10,000$, using 128 threads (corresponding to the real number of cores, which differs from the actual number of threads), omitting the alignment section for the generated output.
- **BLAST v2.16.0** (1): with an E-value cutoff of $10,000$, using 128 threads, and limiting the number of filtered reference sequences to $4,000$.
- **PSI-BLAST v2.16.0** (1): at 2 iterations, with an E-value cutoff of $10,000$, using 128 threads, and limiting the number of filtered reference sequences to $4,000$.
- **Diamond ultra sensitive v2.1.9** (3): using `blastp`, the ultra-sensitive setting and `--algo 1` (i.e. `query-indexed`) option, with an E-value cutoff of $10,000$, using 128 threads, and limiting the number of filtered reference sequences to $4,000$.
- **MMseqs2 k-mer commit 16-747c6** (4): using `easy-search` and the sensitivity at 8.5, with an E-value cutoff of $10,000$, using 128 threads, and limiting the number of filtered reference sequences to $4,000$.
- **MMseqs2 gapless commit 16-747c6** (4): using `easy-search` and the prefilter mode at 1, with an E-value cutoff of $10,000$, using 128 threads, and limiting the number of filtered reference sequences to $4,000$.
- **MMseqs2-GPU commit 16-747c6**: using `easy-search`, the prefilter mode at 1 and enabling the GPU usage via the `--gpu 1 --gpu-server 1` options, with an E-value cutoff of $10,000$, using 1 thread, and limiting the number of filtered reference sequences to $4,000$.

We measured the total execution time of these methods based on the invocations above, and derived an average per-sequence execution. Additionally, we set up searches in either single batch mode, emulating the behaviour in e.g., a protein 3D structure prediction server, or by batching sequences into groups of 10 and 100 sequences, or one batch of 6370 sequences, emulating the annotation of small protein sets or proteomes.
Due to the prohibitive execution time of JackHMMER (2) in any batch mode, or Diamond (3) in single-batch mode, we reduced the query set to only 10%, or 637 query sequences to extrapolate results for the remaining cases. As batch 6370 would require to run 100% of the query set, and batching results from 1 to 100 for 10% of the query set showed negligible effect for JackHMMER, we excluded running this final benchmark with an eye on sustainability.

E.3. Diamond scaling. Diamond (3) was developed to annotate vast metagenomic databases, and its performance is tuned to large query sets and batch sizes. In order to allow a fair comparison of Diamond's performance, we thus performed a query database and batch scaling benchmark exclusively for Diamond. For this benchmark, all previous execution parameters were conserved with the exception of the query database, since up-sampling the 6370 query set to obtain larger query database sizes is not a realistic setting. Instead, we randomly sampled (without replacement) from UniRef90 (40) to obtain query databases of size 100, 1,000, 10,000 and 100,000 sequences. We set the batch size equal to the database size for each run.

E.4. Structure prediction. To compare total runtime and the effect that MSA choice has on structure prediction accuracy, we replicated the ColabFold (30) benchmarks to predict the CASP14 (32) free-modeling queries (FM).
We measured MSA input generation and model inference times for the three configurations listed below. For all three configurations, we omit template search and relaxation and execute the default three recycling iterations.

- **AlphaFold2**: a baseline-configuration using the AlphaFold2 (13) pipeline, utilizing JackHMMER (2) and HHblits (23) to compute input MSAs from three databases, and structure prediction inference using

AlphaFold2 weights. Specifically, this configuration utilizes the following homology search databases: UniRef90 2020_01 (JackHMMer), MGnify 2018_12 (JackHMMer), Uniclust30 2018_08 (HHBlits), BFD first release (HHblits). Five AlphaFold models were run on one L40S GPU via the docker container from github.com/google-deepmind/alphafold (commit f251de6).

- **ColabFold MMseqs2**: a configuration using ColabFold v1.5.5, utilising MMseqs2 with default k-mer on CPU to compute input MSAs, and structure prediction inference loading AlphaFold2 weights. By default ColabFold utilizes recent versions of the databases, however, we retrieved older versions to approximate a fair comparison to the AlphaFold2 configuration, specifically UniRef30 2021_03 and ColabFoldDB 2021_08. Five AlphaFold models were run on one L40S GPU using ColabFold code obtained from github.com/sokrypton/ColabFold (commit fdf3b23).
- **ColabFold MMseqs2-GPU**: a configuration using ColabFold v1.5.5, utilising MMseqs2-GPU with default gapless running on one L40S GPU to compute input MSAs, and structure prediction inference loading AlphaFold2 weights. We used the same databases and codebase as in the *ColaFold MMseqs2* configuration. All five models were run on one L40S GPU.

Structure prediction inference in all three configurations leveraged an L40S. To compare structure prediction accuracy, we computed several metrics between the predicted and ground truth structures, and focused results on local superposition-free score (lDDT) (41) and template modeling score (TM).

E.5. Cloud cost. We compared cloud cost to run the homology search methods presented in the speed benchmark. To do so, we retrieved on-demand hourly cost for AWS EC2 instances (retrieved on 2024/09/24) that best matched our hardware setup for instances in the US East (Ohio) region. We obtained:

- **CPU-based methods**: for JackHMMER, Diamond (ultra sensitive), MMseqs2-k-mer, and BLAST, we selected the instance "c7a.32xlarge" with 128 physical CPU cores and 256 GBs of system RAM at an hourly cost of $6.57
- **GPU-based method**: for MMseqs2-GPU, we selected the instance "g6e.xlarge" with 2 physical CPU cores and 32 GBs of system RAM and one L40S GPU at an hourly cost of $1.86

To obtain per-query-cost, we multiplied the hourly cost by the actual total runtime for each method and divided by number of queries. We reported results for batch size of one, but additional results can be found in Supplementary Material 1, "Cloud cost estimates". Note that prices of cloud instances can differ regionally, and over time. To attempt a robust comparison, we reported the cost factor (i.e., the difference in cost compared to a baseline, which we selected to be MMseqs2 k-mer) rather than the total theoretical cost.

E.6. Energy consumption. To obtain average watt/h utilization, we leveraged `powerstat` and `nvidia-smi` which allow to sample hardware counters (RAPL) included in CPUs and GPUs, and their memory. We performed this measurement for the full query set in single-batch mode for various methods. We then multiplied watt/h average by the execution times to obtain Joules.

E.7. Foldseek. We benchmarked Foldseek (33) using MMseqs2-GPU by sampling 6370 protein structures represented as 3Di strings (avg. size: 261 3Di letters). These were sampled from AFDB50 v4 (53.6M entries, avg. length: 264 3Di letters) against the same database. The database was indexed using `createindex` and stored in memory.

- **Foldseek k-mer commit c438b9** (33): using `search` with option `--db-load-mode 2` for fast index reading and 128 threads `--threads 128`.
- **Foldseek-GPU commit c438b9**: using `search` with option `--db-load-mode 2` for fast index reading, 128 threads `--threads 128` and `--gpu 1` for gapless GPU alignment.

We retrieved the Foldseek SCOPe-based sensitivity benchmark from github.com/steineggerlab/foldseek-analysis (commit 1737c71). Foldseek-GPU was executed with parameters `--max-seqs 2000 -e 10 --gpu 1`. Foldseek k-mer was executed with parameters `-s 9.5 --max-seqs 2000 -e 10`.

References

37. Frith, M. C. A new repeat-masking method enables specific detection of homologous sequences. *Nucleic Acids Research* **39**, e23 (2011).
38. Farrar, M. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* **23**, 156–161 (2007).
39. Hubbard, T. J., Ailey, B., Brenner, S. E., Murzin, A. G. & Chothia, C. SCOP: a Structural Classification of Proteins database. *Nucleic Acids Research* **27**, 254–256 (1999).
40. Alley, E. C., Khimulya, G., Biswas, S., AlQuraishi, M. & Church, G. M. Unified rational protein engineering with sequence-based deep representation learning. *Nature Methods* **16**, 1315–1322 (2019).
41. Mariani, V., Biasini, M., Barbato, A. & Schwede, T. lDDT: a local superposition-free score for comparing protein structures and models using distance difference tests. *Bioinformatics* **29**, 2722–2728 (2013).

Acknowledgments

We thank Yuxing Peng, Maximilian Stadler, and Maximilian Baust for invaluable suggestions and support, Arne Elofsson and Shiraz A. Shah for reporting issues and providing feedback, and Eli Levy Karin for critical feedback on the manuscript. M.S. acknowledges support by the National Research Foundation of Korea grants (2020M3-A9G7-103933, 2021-R1C1-C102065 and 2021-M3A9-I4021220, RS-2024-00396026), Samsung DS research fund, Creative-Pioneering Researchers Program and AI-Bio Research Grant through Seoul National University. M.M. acknowledges support from the National Research Foundation of Korea (grant RS-2023-00250470). B.S. acknowledges support from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 439669440 TRR319 RMaP TP C01. The funding body did not participate in the design of the study and collection, analysis, and interpretation of data and in writing the manuscript.

Author contributions

Felix Kallenborn: Software; Writing - Original Draft; Writing - Review & Editing. **Alejandro Chacon**: Software; Investigation, Writing - Original Draft; Writing - Review & Editing. **Christian Hundt**: Resources; Project administration. **Hassan Sirelkhatim**: Visualization; Investigation; Validation. **Kieran Didi**: Software; Investigation. **Sooyoung Cha**: Software; Investigation. **Christian Dallago**: Conceptualization; Resources; Writing - Original Draft; Writing - Review & Editing; Supervision; Visualization; Project administration. **Milot Mirdita**: Conceptualization; Methodology; Software, Investigation; Writing - Original Draft; Writing - Review & Editing; Supervision. **Bertil Schmidt**: Software; Methodology; Conceptualization; Resources; Writing - Original Draft; Writing - Review & Editing; Supervision;. **Martin Steinegger**: Software; Methodology; Conceptualization; Resources; Writing - Original Draft; Writing - Review & Editing; Supervision.

Competing interests

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: C.D., A.C., C.H., H.S., K.D. are employed by NVIDIA. M.S. declares an outside interest in Stylus Medicine.

Data availability

Data utilized to perform benchmarks for this study are freely available. For speed, sensitivity, and energy consumption benchmarks, we leveraged target sequences stored at mmseqs.steineggerlab.workers.dev/targetdb.fasta.gz and reference sequences stored at mmseqs.steineggerlab.workers.dev/query.fasta . For the folding benchmark, CASP14 targets are available at predictioncenter.org/casp14/index.cgi , while the reference ColabFold databases are available at colabfold.mmseqs.com. For Foldseek benchmarks, we retrieved data and scripts from github.com/steineggerlab/foldseek-analysis

Code availability

All code developed in this study is available under MIT license and documented at mmseqs.com. Analysis scripts are available at github.com/steineggerlab/mmseqs2-gpu-analysis