PEPE: Scalable extraction of multi-modal protein language model representations

Jahn Zhong[1,*,(0000-0002-7247-1392)], Niccolò Cardente[1,(0009-0009-4021-1631)], Geir Kjetil Sandve[2,3,(0000-0002-4959-1409)], Habib Bashour[1,(0000-0001-6660-1843)], Maria Francesca Abbate[1,†,(0009-0006-1718-8769)], and Victor Greiff[1,4,*,†,(0000–0003–2622–5032)]

[1] Department of Immunology, University of Oslo, Oslo, Norway
[2] Scientific Computing and Machine Learning section, Department of Informatics, University of Oslo, Oslo, Norway
[3] UiORealArt Convergence Environment, University of Oslo, Norway
[4] Imprint Labs, LLC. New York, NY, USA

[*] Corresponding authors
[†] Joint supervision

Email: jahn.zhong@medisin.uio.no, victor.greiff@medisin.uio.no

Abstract

Motivation

Protein language models (PLMs) have demonstrated significant potential in capturing the complex interaction patterns between amino acids in protein sequences. These models, trained on large datasets of protein sequences, can generate embeddings, high-dimensional numerical representations that encode valuable information about the structure, function, and evolution of proteins. However, conventional usage has largely been based on arbitrarily determined variable sets (embedding modes), including the choice of embedding layer, pooling method, and padding, which can potentially lead to a suboptimal representation with low information content for a given downstream task. The scalability of protein embedding mode extraction is limited by inefficiencies in both space (memory) and time (computation). (i) Accumulating all outputs in memory and writing them to disk in a single operation leads to a memory bottleneck. (ii) Additionally, repeated embedding of the same sequence to extract different embedding modes introduces unnecessary computational overhead and reduces throughput significantly.

Results

Here, we present PEPE (Parallel Extraction for Protein Embeddings), a command-line tool designed for high-throughput multi-modal protein sequence embedding extraction. We demonstrate that PEPE's parallel process achieves a total run time several orders of magnitude faster than sequential approaches. We also demonstrate how, for a state-of-the-art (SOTA) method, peak memory usage scales with output size and fails once the memory capacity is exceeded, whereas PEPE's peak memory usage remains consistently below the critical limit, allowing the extraction of multimodal embeddings that exceed the available memory. PEPE supports a wide range of publicly available and custom protein language models, providing a simple command-line interface for researchers. PEPE enables the generation of protein embedding datasets at previously unfeasible scales, facilitating the identification of optimal protein embedding settings for downstream analyses without requiring additional resources for fine-tuning.

Availability and Implementation

PEPE is a command-line tool written in Python and published under MIT license. The source code and documentation are available at https://github.com/csi-greifflab/pepe-cli. PEPE is also available for installation from PyPI under https://pypi.org/project/pepe-cli and deposited on Zenodo at https://zenodo.org/records/15912054.

1. Introduction

Recent advances in protein language models (PLMs) have enhanced our ability to learn and interpret the underlying patterns of protein sequences, enabling breakthroughs in computational tasks such as structure prediction [1–3], function annotation [4], and de novo protein design [5,6]. Central to these developments is the adoption of transformer-based [7,7] PLMs, such as ESM2 [2], AntiBERTa2-CSSP [3], ProtTrans [8] or xTrimoPGLM [9]. Typically, PLMs are pre-trained on large and diverse training datasets using self-supervised masked language modeling tasks [10,11], with the aim of learning generalizable feature representations (embeddings) of protein sequences. Pre-trained models are used for transfer learning mainly by: (1) fine-tuning for new tasks with relatively little labeled data [8,10,12–15], or (2) using their embeddings as input features for downstream models [8,10,12–15 14,16–19].

In this work, we use the term embedding to refer to high-dimensional numerical representations extracted from outputs of PLMs. We call different representations of the same sequence “embedding modes” (i.e., different layers or pooling methods). As each embedding layer modifies the outputs of the previous layer and passes them to the next, embeddings from the final layer are considered to be the most enriched in information and are usually the chosen embedding mode for transfer learning tasks [20–22]. However, the relevance of deeper layers varies by task, and in some cases intermediate layers could be more informative, especially for rare sequences, such as those of adaptive immune receptors (AIRs) [20–25].

In the absence of general “best practices”, the optimal embedding mode for a given transfer learning task may be determined by benchmarking multi-modal embedding datasets that contain multiple embedding modes of each protein sequence. While the design and execution of such benchmarks have become more accessible through modern platforms like ImmuneML[26], generating the underlying datasets remains challenging. Although state-of-the-art (SOTA) tools, such as PLMFit [27], are theoretically able to extract many embedding modes of a sequence, in practice, they use a sequential process, which limits scalability due to constraints in memory capacity and throughput (see Figure 1A). The sequential process begins by tokenizing the input protein sequences and dividing them into batches. Then, each batch of tokens is embedded and temporarily stored in memory. Finally, after the outputs of all batches have been stored, they are written to a storage file (extraction). All steps are then repeated for each embedding mode to be extracted, resulting in significant computational overhead. To address these limitations, we designed PEPE (**P**arallel **E**xtraction for **P**rotein **E**mbeddings), where, concurrently with the computation of embeddings, multiple embedding modes can be extracted from a single forward pass through the model.

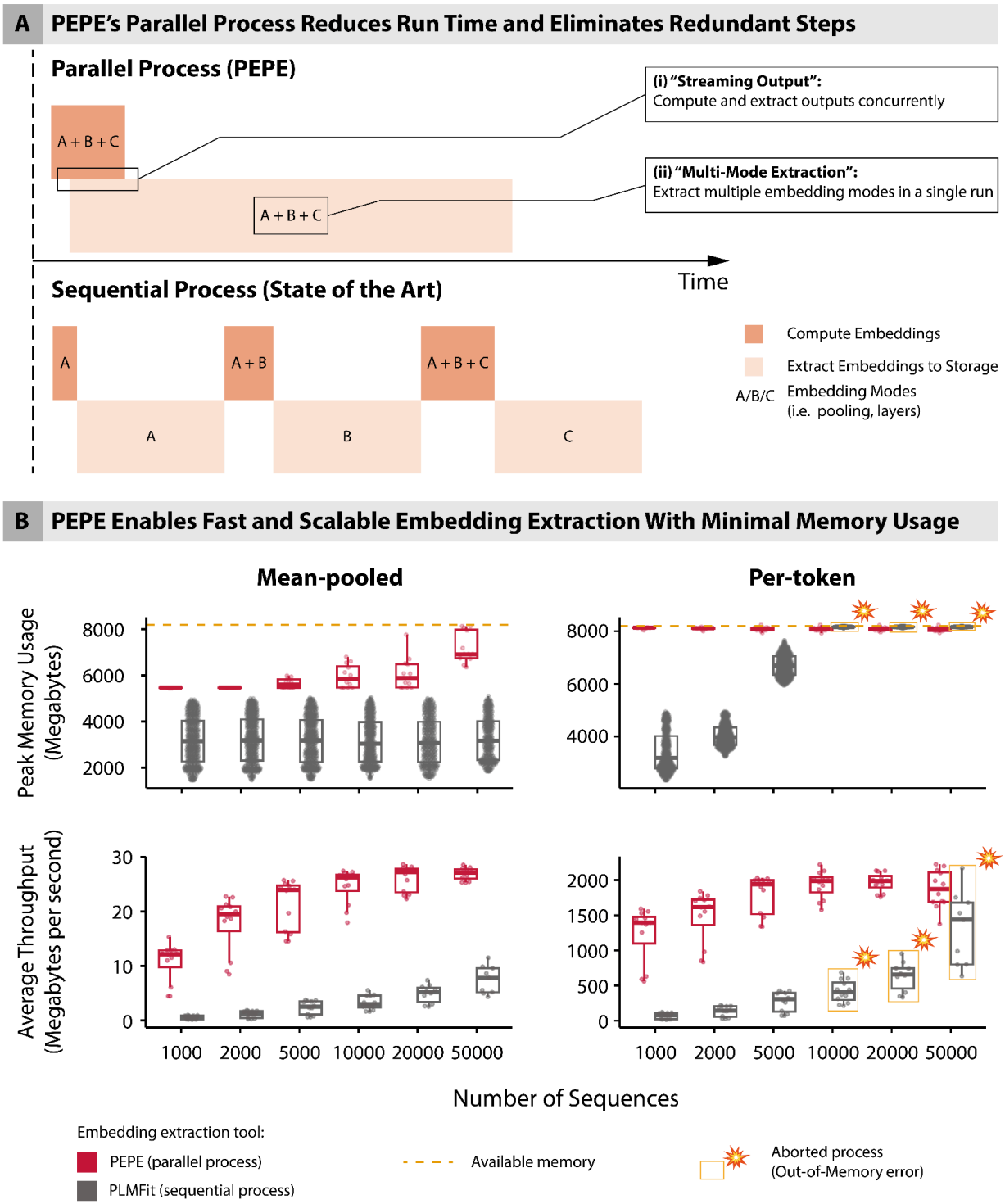


Figure 1: PEPE's parallel processing enhances capacity and throughput compared to state-of-the-art methods. (A) Schematic comparison between parallel process used by PEPE and sequential process used by state-of-the-art (SOTA) tools such as PLMFit [27], AMULETY [27,28] and Bio-embeddings [29], for the extraction of embedding modes from PLMs. Orange blocks represent the computation of embeddings and beige blocks represent the extraction of one or multiple embedding modes (A/B/C) to storage. PEPE offers two key advantages: **(i)** Its ***"Streaming Output"*** function

allows the total size of outputs to exceed the maximum available memory and reduces total run time. **(ii)** ***"Multi-Mode Extraction"*** further reduces run time and resource overhead by extracting all selected modes from a single embedding run. **(B)** Comparison of peak memory usage (upper panels) and average throughput (lower panels) of embedding extraction between PEPE and PLMFit (SOTA). Amino acid (AA) sequences of antibody variants (121 AAs) were collected into sets of size 1000, 2000, 5000, 10000, 20000, and 50000. Both programs were instructed to extract multiple embedding modes (per-token and mean-pooled) from all 33 embedding layers of the ESM2 model. Per-token embeddings are the unreduced output of the embedding layers, while mean-pooled embeddings are averaged across token-dimensions. All commands were run as SLURM jobs and limited to 1 CPU core, 8192MB of CPU memory (yellow dashed line), and 1 NVIDIA A100 GPU. PEPE's "Multi-Mode Extraction" was used to extract all 33 layers in a single run, while for the equivalent output, PLMFit's sequential process requires a repeated run for each layer. Data points in yellow rectangles indicate aborted processes due to out-of-memory errors. All measurements were repeated 11 times (Table 3).

2. Results

Here, we present PEPE (**P**arallel **E**xtraction for **P**rotein **E**mbeddings), a Python command-line tool for the scalable generation of multi-modal embeddings from both public and custom PLMs, enabling embedding extraction at previously unfeasible scales (Figure 1A). We compared PEPE to PLMFit in terms of scalability and performance by using them to extract per-token and mean-pooled embeddings across all 33 layers of ESM2 (total of 66 embedding modes per sequence) while recording memory usage over run time (see Materials and Methods). The extractions were performed with six input datasets of increasing scale (1000, 2000, 5000, 10000, 20000, and 50000 sequences of antibody variants with 121 amino acids in length). As shown in Figure 1B, although PEPE's peak memory usage is generally higher than that of PLMFit when extracting mean-pooled embeddings, equivalent extractions had 3.6- to 18.7-fold higher throughput on average with PEPE compared to PLMFit (see Table 3). When extracting per-token embeddings that are 121 times larger than their mean-pooled equivalents, PEPE's peak memory usage was also higher than PLMFit's for datasets with small numbers of sequences where the average throughput increase was between 6.5- and 17.3-fold. However, when processing datasets with 10000 sequences and above, PLMFit exceeds the memory limit of 8192MB, causing the process to fail due to out-of-memory errors.

In comparison, PEPE's peak memory usage did not exceed the limit with the above-described test settings, regardless of the total output size, demonstrating its ability to scale without memory constraints with "Streaming Output" (see Methods).

Table 3. Comparison of embedding extraction performance between PEPE and PLMFit

Number of Sequences	Total Output Size (Megabytes)	Average Cumulative Run Time (Minutes)		Average Throughput (Megabits/second)		Fold-change
		PEPE	PLMFit	PEPE	PLMFit	
Mean-Pooled Embeddings						
1000	64.5	1.2	34.1	10.7	0.6	18.7
2000	128.9	1.4	34.3	17.7	1.1	15.7
5000	322.3	2.7	37.1	21.2	2.3	9.1
10000	644.5	4.4	39.1	24.6	3.2	7.6
20000	1289.1	8.3	50.6	26.0	4.9	5.4
50000	3222.7	20.0	80.3	26.9	7.5	3.6
Per-Token Embeddings						
1000	7927.7	1.2	34.6	1231.1	71.1	17.3
2000	15855.5	1.9	34.7	1477.1	133.7	11.0
5000	39638.7	3.8	37.8	1788.9	273.7	6.5
10000	79277.3	6.9	Failed	1932.6	-	-
20000	158554.7	13.5	Failed	1965.9	-	-
50000	396386.7	36.0	Failed	1872.4	-	-

3. Materials and Methods

3.1. Program Usage

PEPE is a Python command-line tool for the scalable generation of multi-modal PLM embeddings from protein sequences. PEPE employs a parallel process that addresses the inefficiencies and redundancies of SOTA methods, utilizing a sequential approach. For more detailed instructions and examples, see https://github.com/csi-greifflab/pepe-cli.

A minimum of three options is required to run PEPE: a protein sequence file in FASTA format, the name of the embedding model, and a directory where outputs will be saved as Numpy arrays. The following command template extracts mean-pooled embeddings from the final layer (default options) of the specified model and stores them in the designated output directory:

```
pepe   --model_name <model_name> \
       --fasta_path <file_path> \
       --output_path <directory_pathy> \
       --extract_embeddings <"per-token"/"mean-pooled"> \
       --layers <layer_indices>
```

Out of the box, PEPE supports ESM-1 [15] and ESM-2 family models [2], as well as AntiBERTa2 [3] and ProstT5 family models [30]. The corresponding model strings can be found in the Github repository. There are three ways to set the embedding model using `--model_name`: (i) select a supported model by passing its model string (e.g. "esm2_t33_650M_UR50D"); (ii) select a compatible PLM by passing its Hugging Face Hub address [31] (e.g. "alchemab/antiberta2); or (iii) select a local model by passing its directory path. Additionally, to decrease run times and conserve storage space, outputs can be saved as either 32-bit (full precision) or 16-bit (half precision) floating point numbers.

3.2. Embedding Modes

Multiple representations (embedding modes) of the same sequence can be extracted from a PLM (see Introduction). Embedding modes are selected by type,

pooling, and layer (see Table 2). The two types of modes are conventional embeddings (outputs of embedding layers) and attention matrices (outputs from self-attention heads within embedding layers).

Currently, PEPE supports three pooling options for embeddings, including uncompressed "per-token" and "mean-pooled" embeddings. For proteins with sub-regions known to be functionally relevant (such as the CDRs of antibodies), we also support "substring-pooling". Here, the user supplies a CSV file with substrings that match the input sequences. Each sequence is then embedded and pooled by averaging over the substring positions.

In addition to conventional embeddings, PEPE can also extract token attention matrices. They can be extracted from individual self-attention heads of a hidden layer (per-head), averaged across all heads of a layer (per-layer) and across all heads of all layers (per-model).

Each type and pooling of embeddings can then be extracted from all or any of a PLM's layers. The layers can be defined by supplying a list of integers with negative integers indexing layers in reverse order (-1 = final layer of PLM). PEPE can be instructed to extract embeddings from all PLM layers by passing "all" to the `--layers` option.

3.3. Streaming Output and Multi-Mode Extraction

PEPE's "Streaming Output" is enabled by extracting embeddings to memory-mapped Numpy array files[32]. Memory-mapping is used to read and write to smaller parts of an array without pre-loading the entire file into memory. Here, all available memory is utilized as a buffer that is continuously filled with output data and emptied by writing it to a file. In order to optimize writing speed, writes are grouped and executed in batches. This allows PEPE to (i) generate output files that are larger than the system's memory capacity and (ii) concurrently extract batches of outputs to disk as they are embedded (Figure 1A; Table 1).

Embedding extraction tools using a sequential process work by embedding a given set of protein sequences, extracting a single embedding mode, typically

the mean-pooled outputs of the final embedding layer. In a scenario where multiple embedding modes are extracted, the identical embedding procedure must be repeated for each additional embedding mode. PEPE avoids redundant calculations through "Multi-Mode Extraction", where each embedding mode is assigned a separate memory-mapped file and all are extracted from a single forward pass of the PLM.

3.4. Comparing Scalability of Multi-Modal Embedding Extraction between PEPE and SOTA

Both PEPE and PLMFit (representative of SOTA) were installed in isolated virtual Python environments, as per the authors' instructions. A total of 339482 antibody heavy chain variant sequences with uniform length (L = 121 amino acids) were sourced from a deep mutational scanning study [33]. Six datasets with increasing scale (1000, 2000, 5000, 10000, 20000 and 50000 sequences) were subsampled and converted to FASTA and CSV format as input files for PEPE and PLMFit, respectively. "esm2_t33_650M_UR50D" (ESM2), a 650 million parameter PLM from the ESM-2 family with 33 layers and 1280 dimensions (D), was used as an embedding model.

For each of the six datasets, PEPE and PLMFit were then used to extract mean-pooled (1280 dimensions) and per-token (121 * 1280 dimensions) embeddings from each of the model's 33 layers at "half-precision" (16-bit floating point numbers). As sequential processes are typically only able to extract either mean-pooled or per-token from a single layer at a time, PLMFit required 66 separate processes per dataset (396 total) for the experiment. The equivalent extractions were performed with either two separate PEPE processes per dataset (mean-pooled and per-token) or a total of twelve. From a Python script, each run was executed as a subprocess. Using the 'psutil' Python library, the resident set size (RSS), or physical memory usage, of each process was recorded at 0.1-second intervals. While peak memory usage (Figure 1B, upper panels) is

shown for each process, average throughput (Figure 1B, lower panels) was computed by dividing the total output size by the cumulative run time of extraction processes across all layers of a given dataset size.

All computations were performed on an HPC cluster with SLURM scheduling [34] and repeated 11 times. SLURM jobs were assigned one AMD EPYC 7552 CPU core, 8192 MB of CPU memory, one NVIDIA A100 GPU with CUDA version 12.8 [35] and fast solid-state drive storage. In cases when outputs are written to spinning disk hard drives, we recommend increasing batch size for better performance.

Table 1. Comparison of workflow and efficiency features between PEPE and existing embedding extraction tools.

	PEPE	PLMFit
Workflow & Scalability Features		
Streaming Output	✅	❌
Multi-Mode Extraction	✅	❌
Hugging Face Hub Compatibility	✅	❌
FASTA Input	✅	❌
FP16/FP32 Output Selection	✅	❌

Table 2. Embedding modes supported by PEPE and existing embedding extraction tools.

	PEPE	PLMFit
Output Embedding Modes		
Per-Token	✅	✅

Mean-Pooling	✅	✅
Substring-Pooling	✅	❌
Attention weights (per head)	✅	❌
Attention weights (per layer)	✅	❌
Attention weights (per model)	✅	❌

4. Discussion

Although embeddings extracted from the final transformer layer of a PLM have been shown to be useful features for transfer learning tasks (see Introduction), determining the optimal embedding mode (embedding layer, pooling, etc.) requires the extraction and comparison of multiple embedding modes. The scale of an extraction is determined by the number of input sequences and the number and dimensionality of selected embedding modes (see Materials and Methods). Although PEPE only requires a single run per input dataset, mean-pooled and per-token embeddings were extracted separately to ensure comparability with PLMFit runs. Alternative embedding programs such as AMULETY [28] and Bio-embeddings [29] were not evaluated, as they only supported the extraction of embeddings from a PLM's final layer without custom source code modifications. Nonetheless, since they also work sequentially, we expect them to share similar memory limitations when scaling the size of outputs.

We compared the performance of PEPE against an SOTA method in multi-mode embedding extraction scenarios (Figure 1B) and found that in SOTA methods, peak memory usage increases with output size until available memory is exceeded and the process fails. In comparison, PEPE successfully finished all runs

in a fraction of PLMFit's equivalent run time, while peak memory usage remained constant even with increasing output size.

Further, while we compared multiple successive single-mode processes with PLMFit to a single multi-mode extraction with PEPE by cumulative run time, it is also conceivable to use cumulative memory usage over peak run time and compare multiple simultaneous single-mode processes with PLMFit to a single multi-mode extraction with PEPE instead. However, since even the smallest-scale PLMFit run had a peak memory usage above 2048MB, the cumulative memory usage over 33 embedding layers would far exceed the available memory capacity (total output size > 64GB) and would require a dedicated GPU for each process.

Finally, while SOTA methods use memory as intermediate output storage and require significant overhead to avoid out-of-memory errors, PEPE uses memory-mapped Numpy arrays that efficiently utilize all available memory as a write cache to speedup embedding extraction without exceeding limits (see Methods and Fig. 1B). Although we chose a relatively small memory capacity (8192 MB) for our experiment, reducing PEPE's memory allocation even further would have resulted in little to no compromises in terms of run time. This is because the rate at which embeddings are generated and filled into cache far exceeds the rate at which the cache can be emptied and embeddings written to disk.

In conclusion, PEPE eliminates virtually all memory limitations inherent to SOTA methods like PLMFit, increases throughput by several fold, and enables embedding extractions at scales only constrained by capacity and write speed of the underlying storage device. As PLM capabilities, dataset sizes, and computing infrastructures continue to grow, scalable methods like PEPE will become increasingly critical for enabling future large-scale PLM-embedding-based applications.

5. Disclosure statement

V.G. declares advisory board positions in aiNET GmbH, Enpicom B.V, Absci, Diagonal Therapeutics, and FairJourney Biologics. V.G. is a consultant for Adaptyv Biosystems, Proteinea, and LabGenius. V.G. is an employee of Imprint LLC.

6. Funding

This work was supported by grants from the Norwegian Cancer Society Grant (#215817, to VG), Research Council of Norway projects (#300740, #331890 to VG). This project has received funding (to VG) from the Innovative Medicines Initiative 2 Joint Undertaking under grant agreement No 101007799 (Inno4Vac). This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and EFPIA. This communication reflects the author's view and neither IMI nor the European Union, EFPIA, or any Associated Partners are responsible for any use that may be made of the information contained therein. Funded by the European Union (ERC, AB-AG-INTERACT, 101125630, to VG).

7. References

1. Rao, R. *et al.* MSA Transformer. *Synthetic Biology* (2021).
2. Lin, Z. *et al.* Evolutionary-scale prediction of atomic-level protein structure with a language model. *Science* **379**, 1123–1130 (2023).
3. Barton, J., Galson, J. D. & Leem, J. Enhancing antibody language models with structural information. *bioRxiv* (2024) doi:10.1101/2023.12.12.569610.
4. Hwang, Y., Cornman, A. L., Ovchinnikov, S. & Girguis, P. Deep learning of genomic contexts predicts protein co-regulation and function. *bioRxiv* 2023.04.07.536042 (2023) doi:10.1101/2023.04.07.536042.

5. Høie, M. H. *et al.* AntiFold: Improved antibody structure-based design using inverse folding. *arXiv [q-bio.BM]* (2024).
6. Hsu, C. *et al.* Learning inverse folding from millions of predicted structures. *bioRxiv* 2022.04.10.487779 (2022) doi:10.1101/2022.04.10.487779.
7. Vaswani, A. *et al.* Attention is all you need. *arXiv [cs.CL]* (2017).
8. Elnaggar, A. *et al.* ProtTrans: Towards cracking the language of life's code through self-supervised deep learning and high performance computing. *arXiv [cs.LG]* (2020).
9. Chen, B. *et al.* XTrimoPGLM: Unified 100B-scale pre-trained transformer for deciphering the language of protein. *bioRxiv* 2023.07.05.547496 (2023) doi:10.1101/2023.07.05.547496.
10. Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. BERT: Pre-training of deep bidirectional Transformers for language understanding. *arXiv [cs.CL]* (2018).
11. Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. (2018).
12. Tay, Y. *et al.* UL2: Unifying language learning paradigms. *arXiv [cs.CL]* (2022).
13. Howard, J. & Ruder, S. Universal Language Model Fine-tuning for text classification. *arXiv [cs.CL]* (2018).
14. Rao, R. *et al.* Evaluating protein transfer learning with TAPE. *arXiv [cs.LG]* (2019) doi:10.1101/676825.
15. Rives, A. *et al.* Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proc. Natl. Acad. Sci. U. S. A.* **118**, e2016239118 (2021).
16. Fan, W. *et al.* Computational protein science in the era of large Language Models

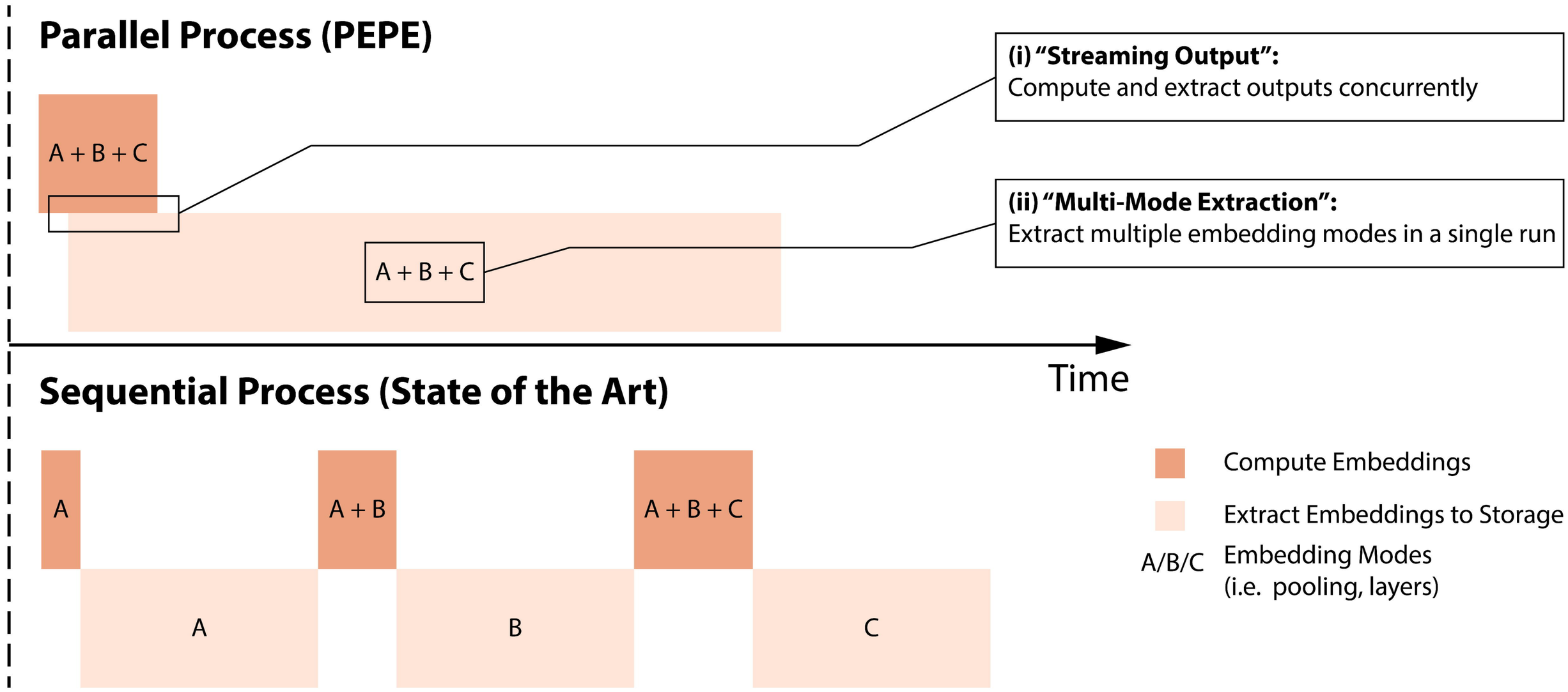
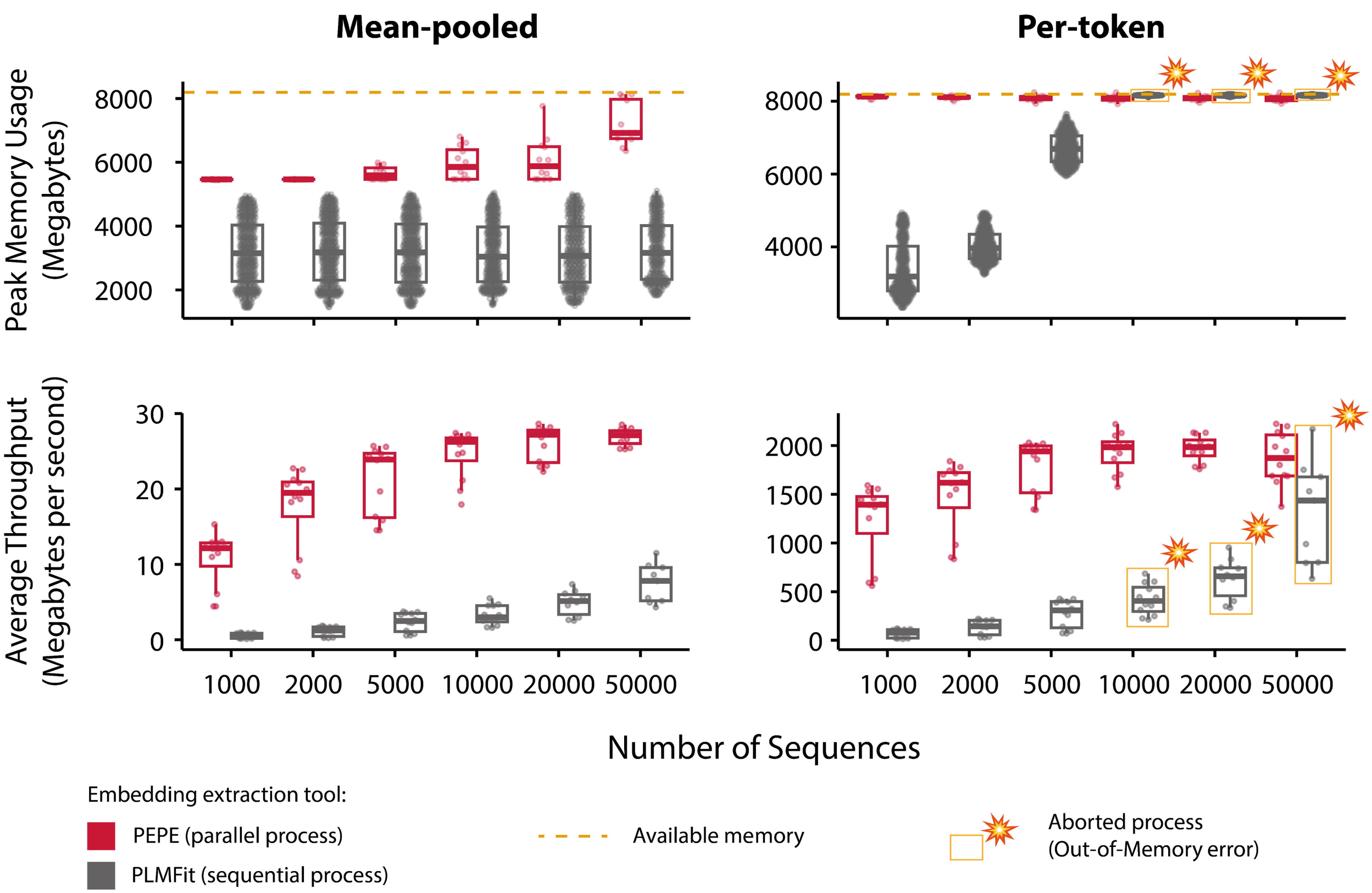
(LLMs). *arXiv [cs.CE]* (2025).

17. Ruffolo, J. A. & Madani, A. Designing proteins with language models. *Nat. Biotechnol.* **42**, 200–202 (2024).
18. Chandra, A., Tünnermann, L., Löfstedt, T. & Gratz, R. Transformer-based deep learning for predicting protein properties in the life sciences. *Elife* **12**, (2023).
19. Li, F.-Z., Amini, A. P., Yue, Y., Yang, K. K. & Lu, A. X. Feature reuse and scaling: Understanding transfer learning with protein language models. *bioRxiv* 2024.02.05.578959 (2024) doi:10.1101/2024.02.05.578959.
20. Gonzales, M. E. M., Ureta, J. C. & Shrestha, A. M. S. Protein embeddings improve phage-host interaction prediction. *PLoS One* **18**, e0289030 (2023).
21. Garbas, L., Ploner, M. & Akbik, A. TransformerRanker: A tool for efficiently finding the best-suited language models for downstream classification tasks. *arXiv [cs.CL]* (2024).
22. Luo, Z. *et al.* Interpretable feature extraction and dimensionality reduction in ESM2 for protein localization prediction. *Brief. Bioinform.* **25**, bbad534 (2024).
23. Wang, M., Patsenker, J., Li, H., Kluger, Y. & Kleinstein, S. H. Language model-based B cell receptor sequence embeddings can effectively encode receptor specificity. *Nucleic Acids Res.* **52**, 548–557 (2024).
24. Deutschmann, N. *et al.* Do domain-specific protein language models outperform general models on immunology-related tasks? *Immunoinformatics (Amst.)* **14**, 100036 (2024).
25. Yosinski, J., Clune, J., Bengio, Y. & Lipson, H. How transferable are features in deep neural networks? *arXiv [cs.LG]* (2014).
26. Pavlović, M. *et al.* The immuneML ecosystem for machine learning analysis of

adaptive immune receptor repertoires. *Nat. Mach. Intell.* **3**, 936–944 (2021).

27. Bikias, T., Stamkopoulos, E. & Reddy, S. T. PLMFit: benchmarking transfer learning with protein language models for protein engineering. *Brief Bioinform* **26**, (2025).
28. Wang, M., Kluger, Y., Kleinstein, S. H. & Gisela, G. b. AMULETY: A Python package to embed adaptive immune receptor sequences. *bioRxiv* 2025.03.21.644583 Preprint at https://doi.org/10.1101/2025.03.21.644583 (2025).
29. Dallago, C. *et al.* Learned embeddings from deep learning to visualize and predict protein sets. *Curr. Protoc.* **1**, e113 (2021).
30. Heinzinger, M. *et al.* Bilingual language model for protein sequence and structure. *NAR Genom. Bioinform.* **6**, lqae150 (2024).
31. Wolf, T. *et al.* HuggingFace's transformers: State-of-the-art natural language processing. *arXiv [cs.CL]* (2019).
32. Harris, C. R. *et al.* Array programming with NumPy. *Nature* **585**, 357–362 (2020).
33. Chinery, L. *et al.* Baselining the Buzz Trastuzumab-HER2 Affinity, and Beyond. *bioRxiv* 2024.03.26.586756 (2024) doi:10.1101/2024.03.26.586756.
34. Fox System Overview - University of Oslo. https://www.uio.no/english/services/it/research/platforms/edu-research/help/fox/system-overview.md.
35. Nickolls, J., Buck, I., Garland, M. & Skadron, K. Scalable parallel programming with CUDA. in *ACM SIGGRAPH 2008 classes* (ACM, New York, NY, USA, 2008). doi:10.1145/1401132.1401152.

A PEPE's Parallel Process Reduces Run Time and Eliminates Redundant Steps



B PEPE Enables Fast and Scalable Embedding Extraction With Minimal Memory Usage