Exascale Distillation Workflows for Protein Structure Prediction on El Capitan

Nikoli Dryden*, Tal Ben-Nun*, Pier Fiedorowicz*, Tom Benson*, Vinay S. Swamy†, Colin Kalicki†, Vinayak Gokhale‡
J. Austin Ellis‡, Nicholas Malaya‡, Bronis R. de Supinski*, Mohammed AlQuraishi†, Brian C. Van Essen*
*Lawrence Livermore National Laboratory †Columbia University ‡Advanced Micro Devices, Inc.

***Abstract*—Understanding the structure of proteins is critical to many biological applications. Recent deep learning approaches, such as AlphaFold, make high-quality structure predictions. However, training state-of-the-art models requires producing distillation datasets, which need several times more compute than training. We describe ElMerFold, an optimized distillation data generation workflow that scales to 11,200 nodes (44,800 APUs) on El Capitan by leveraging a combination of inference, workflow, I/O, and resiliency innovations. We demonstrate this for the OpenFold 3 monomer protein distillation dataset, which uses a complex workflow centered on a set of pretrained OpenFold 2 models to predict structures for ~41 million proteins. ElMerFold achieves a 45.4× speedup over the state-of-the-art, predicting 6718 structures/s, and reaches 1.99 Eflop/s peak (1.81 Eflop/s sustained including I/O) for FP32 model inference. As distillation and synthetic data become more common, these workflows will help drive scientific advancement.**

I. Justification

Sustained throughput of 6718 predicted structures/s (14.8M atoms/s) for generation of the OpenFold 3 monomer protein distillation dataset, including I/O and startup time, on 11,200 nodes (44,800 MI300A APUs) of El Capitan (45.4× speedup over state-of-the-art). Peak single-precision model inference performance of 1.99 Eflop/s (1.81 Eflop/s sustained).

II. Performance Attributes

Performance attribute	Our submission
Category of achievement	Time-to-solution, peak performance
Type of method used	Machine learning
Results reported on the basis of	Whole application including I/O
Precision reported	Single precision
System scale	Measured on full-scale
Measurement mechanism	Timers, FLOP count, performance modeling

III. Overview of the Problem

Understanding the structure of proteins and biological complexes is a critical enabler for many applications, including drug, antibody, and therapeutic design. However, experimental determination of structures is laborious, expensive, and often takes months, making high-throughput design and experimental loops infeasible. Accurate computational solutions to the protein folding problem—predicting the structure of a protein from its amino acid sequence—is a grand challenge in biology [1].

Community experiments, such as the Critical Assessment of Structure Predictions (CASP), have served to both assess and drive progress in this area. Historically, approaches to protein structure prediction have been either template-based, refining existing protein structures, or template-free *ab initio* methods, such as molecular dynamics [2]. Recently, deep learning-based approaches, exemplified by AlphaFold 1, 2, and 3 [3]–[5], AlphaFold-Multimer [6], RoseTTAFold [7], and OpenFold [8], have largely solved structure prediction for single-protein chains and made major advances in modeling multi-chain proteins and other biological complexes. Indeed, this work was recognized with the 2024 Nobel Prize in Chemistry. These models incorporate the protein sequence, co-evolutionary information from multiple sequence alignments (MSAs), and structure templates into end-to-end learned models that directly predict structures and provide calibrated confidence estimates.

However, although the state-of-the-art AlphaFold 3 [5] model provides open weights and a public inference server, its weights and outputs are placed under a restrictive license [9], and the training code and datasets are unavailable. These restrictions limit studies of different training procedures, incorporation of new training data, or adaptation of the model to new applications. OpenFold [8], a fully open re-creation of AlphaFold 2, yielded multiple insights into the data needs and the learning dynamics of the model, thus demonstrating the value of enabling the broader community to leverage the work.

While training models such as AlphaFold 3 from scratch is a major computational undertaking, currently estimated to require ~3 days, when including hyperparameter tuning, using all of El Capitan (the #2 supercomputer on the Top500 list [10]), AlphaFold 3 and other modern models (e.g., [11]) **use distillation datasets that require significantly more compute to produce than training the model**. *Distillation* [12] uses a pretrained model to produce high-quality data that is then used for additional training while also being independently valuable. This process can involve *self-distillation*, which uses the predictions to continue training of the original model, allowing a model to bootstrap a virtuous cycle of improvement. Similar approaches are also used when training frontier LLMs, which incorporate synthetic data and AI feedback [13], [14].

In particular, AlphaFold 3 was trained in part using ~41 million monomeric proteins from MGnify [15] with structures predicted by AlphaFold 2; producing a similar dataset using the public OpenFold distillation workflow was estimated to **require over one week of time using all of El Capitan**. Extending this process to additional datasets, such as multimers, would require an additional month or more. These extreme compute requirements have greatly limited the production of distillation

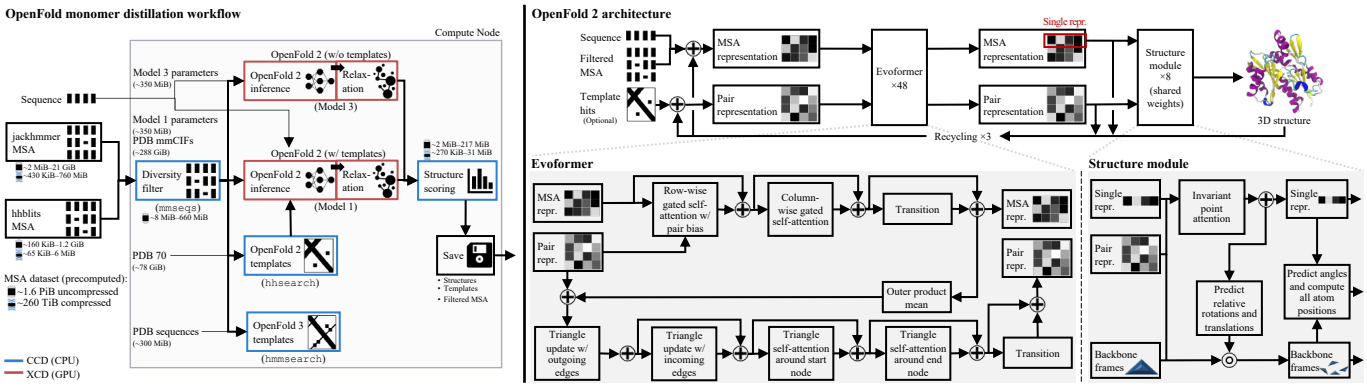


Fig. 1. Left: Overview of the OpenFold monomer protein distillation data generation workflow. Right: Overview of the OpenFold 2 model [4], [8].

datasets, and existing AlphaFold 3 reproductions (e.g., [16]–[18]) have used smaller datasets. In this work, we optimize and scale the workflow for generating the monomer protein distillation dataset on El Capitan as an exemplar of large-scale distillation data generation and as a precursor to training OpenFold 3. This effort requires a suite of novel optimizations and the holistic integration of El Capitan's unique capabilities to support the data storage, data movement, and diverse compute and memory requirements of the complex workflow.

IV. Current State of the Art

The workflow for producing the OpenFold 3 monomer distillation dataset involves the complex orchestration of multiple tools, centered on inference with pretrained OpenFold 2 models, as Fig. 1 illustrates. It uses ~41 million protein sequences from MGnify [15], for which we precomputed MSAs using both `jackhmmer` 3.3.2 [19] and `hhblits` 3.3.0 [20]. The two sets of MSAs are merged and filtered to retain only the most diverse alignments using MMseqs2 [21]. This combination serves as the input MSA for OpenFold 2, with which we run inferences with two separate models: one without structure templates and the other with structure templates computed using `hhsearch` over the PDB70 clustering database [20]. We additionally compute structure templates for OpenFold 3 using `hmmsearch` [19]. Finally, we score and save the predicted structures. The OpenFold 2 model is a complex Transformer-based architecture centered on a series of Evoformer blocks [4], which use a variety of attention mechanisms to integrate MSA and template information, followed by a structure module which introduces the 3D structure. Once predicted, the structure is refined through a relaxation step using gradient descent in the AMBER force field [22], implemented in OpenMM 8.3.1 [23].

OpenFold 2 provides a well-optimized inference implementation in PyTorch [24], [25], including performance and memory efficiency improvements over the original JAX implementation of AlphaFold 2. It additionally includes support for handwritten CUDA kernels for Evoformer attention operations through DeepSpeed4Science [26]. However, the kernel implementations are specific to NVIDIA GPUs, optimized for Ampere architectures, and either do not work or do not offer good performance on AMD GPUs or APUs. Similarly, while efficient attention implementations, such as FlashAttention [27], [28], are widely used, they do not support the particular attention operations needed by the Evoformer. Other works, such as FastFold [29] and ScaleFold [30], focus on improving the *training* phase and also target NVIDIA GPUs. Achieving good performance is further challenged by the variety of sequence lengths present in the distillation dataset, which vary over several orders of magnitude, see Fig. 2, and, consequently, have very different performance and memory characteristics, necessitating careful tuning. Any such optimization is critical: as opposed to LLM inference, which usually takes milliseconds per token, each OpenFold 2 inference run takes multiple seconds, and some take ***over an hour to predict the structure of a single protein***.

Inference servers, such as vLLM [31], Ollama [32], or `llama.cpp` [33], while effective for transformer models such as LLMs, do not support the OpenFold 2 model. Other standard inference optimizations, such as PyTorch's model compilation and exporting, either fail with the complicated model or yield no performance benefits. Additionally, existing inference solutions are unable to integrate with the broader workflow needed to generate the distillation data.

Finally, the I/O and data movement needs of this workflow require special consideration. The large input MSA dataset, ~1.6 PiB in size, is infeasible to move or replicate, even after compression, and must be on a large filesystem enabling fast, random read access by the data generation workflow. Additionally, for each prediction, a node requires access to multiple protein databases to perform searches and lookups. Thus, standard solutions to I/O in deep learning, such as data sharding or prefetching [34], are unsuitable for this workload.

As the distillation dataset is fixed, this is inherently a strong-scaling problem, which only a select few entities have run at scale: to our knowledge, only DeepMind has produced a distillation dataset on the scale of this monomer distillation dataset, and no performance numbers have been released. Other distillation datasets (e.g., [8]), are orders of magnitude smaller.

These characteristics mean that an optimized workflow, for a system with a highly-integrated I/O subsystem and massive compute and memory throughput, is required to make large-scale distillation data-generation workflows feasible and timely. Furthermore, the insatiable demand for higher-fidelity models and for larger, more complex datasets means that these workflows can fully utilize even the largest supercomputers.

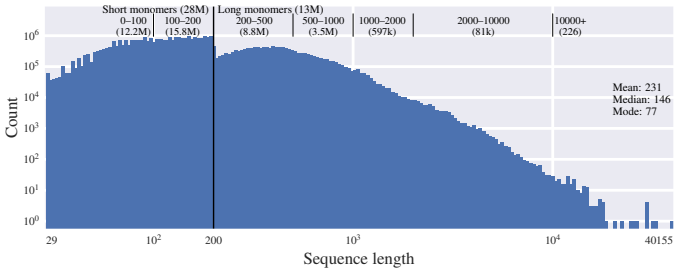


Fig. 2. Distribution of sequence lengths in the OpenFold 3 monomer protein distillation dataset, and sequence counts for different sequence length ranges.

V. Innovations Realized

Our innovations in generating the distillation set span the entire hierarchy of the El Capitan ecosystem—from maximizing the utilization of each APU, through ensuring each node fetches data efficiently for its constituents, to scheduling decisions to optimize sustained performance. We refer to the assembled suite of optimizations described here as **ElMerFold**.

Since the process of generating protein structures can yield variations on the same structure, we validate our modifications by using US-align [35] to compute TM-Scores [36] to ensure structures do not change. Our modifications produced no discernible change in output quality (median TM-Score of 1.00, always larger than 0.8) on a validated subset.

A. OpenFold 2 Inference Optimizations

The inference workflow of OpenFold 2 is a standalone process that broadly comprises four steps (Fig. 1): model parameter *initialization* for inference, *preprocessing* the input protein sequence and diversity-filtered MSAs for inference, running model *inference* on the input sequences, and performing structural *relaxation* on the model output. Our baseline is the public OpenFold 2 GitHub repository ported to ROCm (excl. CUDA optimized attention kernels) and run on El Capitan. Through a shift to a server-based architecture, and through performance model-driven and MI300A-specific optimizations, we yield a speedup of 15.2× when normalized to the sequence length distribution (maximum 39.9× for a single sequence), as §VII-A details. We describe key optimizations below.

Copy Elision and Memory Reduction on MI300A. The MI300A (Fig. 4) shares the same physical memory between the core complex dies (CCD, i.e., CPU) and the accelerator complex dies (XCD, i.e., GPU), which eliminates the need for separate "host" and "device" buffers. We use this feature to reduce overheads in the data pipeline using the LBANNv2 library [37], which supports allocating memory buffers suitable for interoperation between CCD and XCD cores and to manage the necessary synchronization for a memory buffer.

XCD Feature Preprocessing. On longer sequences, feature preprocessing becomes a bottleneck. We migrate some of the computations to the XCDs but keep the coordinate conversion to torsion angles on the CCD for numerical stability.

Inference Performance Characterization. We observe that the sequence length S heavily impacts the performance characteristics of inference, and that there are three regimes: 1) Short sequences are launch-bound. 2) Longer sequences are dominated by attention computations. 3) The longest sequences ($S \geq 2000$) are memory capacity-bound. We thus proceed with a sequence of optimizations to address these bottlenecks.

Overhead Reduction via Algebraic Regrouping. Certain operations in the OpenFold 2 model (e.g., matrix-matrix multiplication, template pointwise attention) are quadratic in S. For such operations, OpenFold 2 defines "chunks" of controllable size, which enable stacking or, inversely, breaking down operations to run sequentially. This enables trading off the memory footprint (smaller chunks) versus the number and size of kernels (larger chunks). Based on our performance and memory requirement modeling (§VI-A, §V-B), we devise heuristics for chunk sizes that maximize performance based on the sequence length while enabling long sequences to run.

Optimized Attention Kernels. OpenFold 2 has five variants of attention across multiple modules, which take over 60% of the total baseline runtime. The model uses a variant of the multi-head attention operator, $\mathrm{softmax}\left(\frac{1}{\sqrt{d}}QK^T + \sum_i B_i\right) \cdot V$, for each attention head. The addition of multiple bias terms B_i allows the model to encode protein-specific structural biases (e.g., graph structure, triangles). We initially developed a suite of optimized Triton [38] and DaCe [39] attention kernels, adapting the FlashAttention-2 [27], [28] fusion and tiling strategy, which significantly outperformed the PyTorch baseline (Fig. 6). However, these still underutilized the APU, and we proceeded to develop bespoke HIP kernels for the MI300A.

MFMA instructions. Triton only supports two of the five available FP32 matrix-fused-multiply-accumulate (MFMA) instructions [40] on the MI300A: 32×32×2 and 16×16×4. This results in Triton padding operations in kernels such as extra MSA attention. We benchmarked all five sizes, including batched MFMA instructions—unique to the AMD Matrix Core architecture—and use the batched 16 (4×4×1_16b) version as the core of a padding-free attention kernel that recovers the full compute potential of the device. Batched MFMAs also enable us to devise a *first-of-its-kind* shuffle-free partitioning for attention kernels, alleviating pressure on the Local Data Share (LDS) unit. Following this, we identified that the kernel was now bound by integer Vector Arithmetic Logic Unit (VALU) instructions, despite using wide load/store operations. These are due to load offsets and masked loads, resulting from the varying sequence length. We proceed to reduce those instructions.

Global-register path: Buffers and direct-to-LDS. By default on the MI300A, accessing global memory triggers `global` instructions. However, `buffer` memory instructions and their associated Buffer Resource Descriptors enable a wider range of operations which we take advantage of: 1) Buffer addressing enables linear reads with strides and offsets, with which we encode our attention loop offsets without explicit pointer advancing. 2) When accessed in strides of four, buffers support reading directly to the LDS, alleviating register pressure. 3) Buffers support range checking and clamping, allowing us to remove masking instructions. Combining these, we eliminate *all* VALU integer instructions in the core loop, leaving only four scalar ALU instructions.

Register-register path: LDS and DPP. Some attention opera-

tions (e.g., triangle) benefit from larger MFMA instructions, which incur cross-lane shuffling for reductions in softmax. By default, this operates via the LDS unit; however, certain instructions can use Data Parallel Primitives (DPP) to communicate across wavefront lanes. We select either the LDS or DPP path for each attention version to maximize performance.

Register allocation. We constrain the compiler using inline assembly to control General Purpose Register (GPR) allocation across scalar (SGPR), vector (VGPR), and matrix core accumulator (AGPR) registers, which we use to override suboptimal compiler decisions. In particular, we found that any use of AGPRs incurs transitional instructions copying from matrix core outputs to VGPRs for ALU operations, and so we eliminate them using the `-amdgpu-mfma-vgpr-form` LLVM flag.

Instruction reduction and pipelining. Having fixed our partitioning and data motion strategies, our kernel is now limited by instruction count and register pressure. We manually hoist operations (e.g., reciprocals) out of loops; use packed elementwise operations when possible; use the `max3` instruction, using a particular operation order (left-leaning chain) to overcome HIP compiler limitations; use `exp2` device intrinsics and save in-kernel multiplications by pre-scaling values into the $\log_2$-domain by changing the prior GEMM coefficients; and interleave MFMA fences to further pipeline VALU and MFMA operations. We note that several of these optimizations (e.g., `max3`) could not be implemented in abstractions such as Triton or DaCe without adding highly specialized passes.

Specialization. Lastly, we dispatch to tuned kernel variants based on the dimensionality and sequence length of the attention call. Overall, these kernels are **up to 2.3× faster** than the Triton kernels (§VII-B).

Pointwise Attention Kernels with DaCe. The above optimizations target compute-bound attention kernels; our Triton pointwise attention kernel is heavily memory-bound. Analysis showed that the tiling scheme and memory management Triton forces (placing intermediates in the LDS) are suboptimal. We instead implemented a bespoke DaCe [39] version of this kernel; note that the DeepSpeed4Science code lacks such a kernel and uses standard (unfused) attention.

The specialized kernel is designed for short sequence lengths, efficiently utilizes the register file, and performs cooperative loads. The end result is **21.6× faster** than the Triton kernel, and 48.9× faster than the original implementation (§VII-B).

Remaining performance. We used cycle-accurate instruction tracing [41] to confirm that **our kernels are either memory-bound or instruction issue-bound** (all memory transfers are hidden and the MFMA and VALU instructions are pipelined across the compute unit). Hence, further optimization would require precision or algorithmic changes, altering the output.

Reducing Launch Overhead. Following these kernel optimizations, sequences shorter than 500 residues are launch-bound due to Python interpreter overhead and the PyTorch runtime. We first measured the inter-kernel latency using standard PyTorch modules and Triton kernels as ~50 µs between each XCD activity (Table I; runtimes are the median of five runs). In an initial attempt to reduce this, we rewrote the core

TABLE I
INTER-KERNEL GAP ANALYSIS ($S = 61$, `MGYP003975845811`)

Version	Wall-Clock Time	Kernel Time	Gap
PyTorch + Triton	1924.2 ms	1729.6 ms	10.11%
PyTorch + C++ modules	1644.6 ms	1047.0 ms	36.34%
`torch.export`	949.4 ms	911.6 ms	3.98%
AOTInductor	671.6 ms	669.7 ms	~0.00%
`of2.cpp`	635.5 ms	629.7 ms	0.91%

`Attention` and `TriangleMultiplicativeUpdate` modules in C++, calling PyTorch's C++ `libtorch` directly (Table I, "PyTorch + C++ modules"). We also explored HIP graphs, but these require either binning by S, which would effectively introduce padding, or regeneration for each length.

To further reduce runtime, we considered the recent `torch.export`, introduced in PyTorch 2.1, and Ahead-of-Time Inductor (AOTInductor, which supports further inference optimizations, PyTorch 2.2+ with subsequent improvements). These further reduce the gap, but it is challenging to support *symbolic* sequence lengths with both, and neither support chunking (loops with symbolic extents are unsupported), making long proteins infeasible due to the memory requirements.

We therefore opted to port the entire OpenFold 2 model to C++ and *remove the PyTorch runtime*, directly calling vendor libraries and our kernels, and manage memory with a custom arena allocator. This reduced launch overhead to less than 1%, while supporting chunking and enabling other optimizations, such as improved layer normalization kernels and more extensive fusion. Further, control over the runtime enables us to use *specialized GEMM algorithm heuristics* rather than rocBLAS's defaults. We create a representative-residue-count-informed grid, over which we perform an offline search for the best rocBLAS GEMM solution for relevant sizes, and find the nearest neighbor from the table at runtime. We use this implementation (`of2.cpp`) for all runs.

Relaxation Optimizations. With inference optimized, relaxation becomes the largest bottleneck in the workflow. We found that the relaxation implementation had significant overhead from the Python interpreter, string operations, and type/framework conversions; it also lacked reproducibility due to nondeterministic reductions in OpenMM.

We therefore ported the relaxation process to C++ and HIP, directly using the *atom37* representation and canonical amino acids OpenFold 2 uses, and correct the protein structure, following the baseline routine (`pdbfixer`) exactly. For optimization, we implemented a force field reader that ingests the OpenMM AMBER99SB XML file, our own energy and gradient evaluator in HIP, and an L-BFGS optimizer that uses the same backtracking line search with strong Wolfe condition as OpenMM. The evaluator runs in FP64 without atomics to ensure accuracy and determinism.

We also developed two further innovations to improve relaxation: 1) We implement a 1.2 nm cutoff in the HIP kernel for proteins larger than 20,000 atoms, where forces become negligible. 2) By default, `pdbfixer` oxygen terminals (OXT) are set, followed by a soft-force field minimization. Instead, we analytically place OXT at a carboxylate-symmetric point

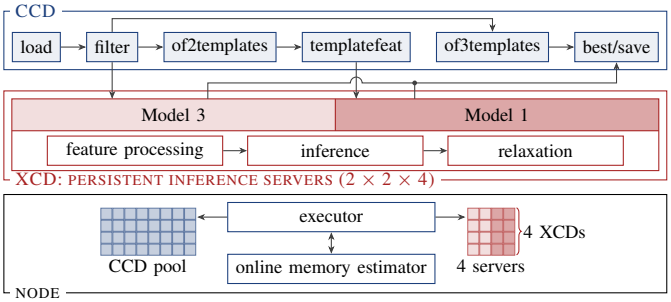


Fig. 3. Diagram of our on-node persistent inference server and task scheduling.

and leave the terminals unconstrained so they can be optimized in tandem during relaxation. In observed cases, this placement was within 0.3 Å of the final position, so it serves as a reasonable starting point. We additionally implemented an AVX-512 vectorized CCD-based version of relaxation with a custom thread pool, but found the XCD implementation to offer greater overall performance.

Our implementation is both bitwise reproducible for a fixed random seed and **8.5–90× faster than the original relaxation pipeline**, while maintaining high TM-scores, effectively eliminating relaxation as a bottleneck.

B. Distillation Workflow Optimizations

After optimizing inference, we turn to optimizing the broader workflow (Fig. 1) to ensure high utilization.

On-node Workflow Orchestration. The original OpenFold 2 distillation workflow executed the entire workflow sequentially despite significant opportunity for parallel execution and CCD/XCD overlap. To extract the maximum parallelism possible, we built a resource-aware graph scheduler to manage workflow task execution (Fig. 3). Each task consists of an operation (e.g., the diversity filter), dependencies (prior tasks that must successfully complete first), and required resources (e.g., CCD cores, memory, storage). An online resource tracker monitors the available resources and only allows tasks to execute when their dependencies and resources are satisfiable.

We manage scheduling at four levels: 1) Prior to startup, we statically assign sequences to nodes based on their length, ensuring that each node has an approximately equal mix of sequence lengths. 2) Each node greedily selects sequences for which to predict structures based on the availability of the initial resources needed for the workflow (storage for MSAs, memory). Once selected, the workflow for that sequence begins enqueueing tasks for execution. 3) We select eligible tasks to run from those queued whenever their resources are satisfied, picking arbitrarily to ensure progress even when the number of ready tasks exceeds available resources. 4) The scheduler monitors for tasks that are starved due to large memory requirements and places brief reservations to ensure they run, without stalling the workflow. This scheduling approach helps ensure consistently high utilization over time by overlapping the longer processing time and higher memory requirements of long sequences with multiple short sequences.

Persistent inference servers. The original workflow performed inference by invoking a Python subprocess which ran the OpenFold 2 PyTorch implementation on the sequence. This incurs significant overhead from Python initialization, library imports, ROCm initialization, and model initialization/weight loading, and also creates system-wide filesystem load (§V-C). We address this by implementing a persistent inference server (see Fig. 3) which is launched on each node during startup.

The server manages a set of `of2.cpp` processes across all four APUs, which each manage inference for a particular model. These persist to amortize startup across an entire run and carefully manage memory to avoid leaks or contamination across inferences. Our on-node scheduler sends inference tasks to the server when they are ready, and monitors for completions.

To further improve XCD utilization, we oversubscribe the XCDs by running up to four concurrent inference processes on the same device. This ensures that CCD-bound portions of one inference request (e.g., data ingestion) can overlap with XCD compute from other requests, and also better-utilizes the XCD hardware, achieving greater effective utilization and improving overall throughput. As the OpenFold 2 model weights are small (~350 MiB), the memory overhead is minimal. Oversubscription requires careful scheduling of overlapping inference tasks, particularly with regard to memory management. For especially long sequences, an individual inference task can consume all device memory, which would typically result in an out-of-memory (OOM) error. As device-managed memory is not accurately tracked by the OS kernel on MI300As, OOM errors can result in arbitrary user processes being killed, risking an inconsistent or unrecoverable application state.

To prevent OOMs, the inference server uses an online memory estimator to reserve memory for each inference on a device, as this is monotonic in the sequence length and input MSA depth. Inference tasks track their peak memory usage, allowing the estimator to refine its predictions.

We considered batched inference as a potential optimization, but ultimately rejected it due to three major drawbacks: 1) Padding input sequences to match the longest in a batch would waste compute with mismatched sequence lengths, requiring complex scheduling to address. 2) The cost of waiting to accumulate a batch would prohibitively impact throughput given the high variability in and unpredictability of sequence arrival times. 3) For longer sequences, batching is not possible, as inference is already near memory capacity.

Additional workflow optimizations. After the above optimizations were completed, we observed that CCD components of the workflow were becoming a significant bottleneck, limiting our executor's ability to pipeline work. We addressed this with a number of runtime and memory usage optimizations:

Filter. While the `mmseqs` diversity filter already used hand-tuned AVX2 intrinsics, it suffered from poor cache locality, especially for long sequences. We restructured the filter to ensure contiguous accesses were possible, and additionally reimplemented it using AVX-512, resulting in a speedup of 2–7× depending on the sequence length. Subsequently, we improved the filtering algorithm to avoid redundant work and more efficiently store data, achieving a further 7× speedup and 45% memory reduction for deep alignments.

OF2 templates. We determined that `hhsearch` spent significant time reparsing PDB70 for every query; by precomputing and caching steps that do not depend on the query, we achieve 1.2–2.1× speedups, benefiting short sequences the most due to the fixed overheads. We also found that its Viterbi algorithm allocated buffers sized to a worst-case upper-bound, and replaced this with a reusable, thread-local buffer sized to the number of steps, reducing memory usage by up to 2.5×.

OF3 templates. We found that the reference PDB sequence database used to construct this search had significant redundancy; we deduplicated it while ensuring identical results, obtaining a 4.1× speedup and reduced memory use.

Template features. This stage computed hits and constructed the PDB templates using the OF2 template results. Originally an inference step, we found this was almost entirely string parsing; we implemented an optimized PDB parser and hoisted the task out of the inference server, overlapping it.

C. I/O Optimizations

At the system level, our workflow leverages El Capitan's Lustre filesystem and Rabbit near-node storage (§VI-B) to minimize I/O overheads and ensure scalability.

MSA Storage. Our precomputed MSA dataset totals ~1.6 PiB; to reduce the storage impact, we compress MSAs with Zstandard when on Lustre, requiring only ~260 TiB. As storing MSAs as individual files would consume too much metadata, we pack them into archives that contain ~15 GiB of MSAs when uncompressed. To enable efficient random access, our archives are `.tars` of compressed MSAs (rather than a compressed `.tar`), combined with a precomputed index that maps the sequence ID to the MSA's archive, byte offset in the archive, and size. Thus, an MSA can be read after only a single file seek; the compressed data is streamed through a decompressor and written to Rabbit storage. We employ a similar approach to store output structures.

OpenFold Input Pipeline. We optimize OpenFold 2's input pipeline to eliminate unnecessary temporary files and reads and to improve data loading. In particular, it requires structure templates, which are loaded based on searches computed earlier in the pipeline. There are ~223k templates; we adopt a similar indexed archive approach, without compression, which significantly reduces data staging and template seek times.

Startup. Python startup time at scale is a major challenge; without dedicated optimization, large-scale Python workflows cannot start on El Capitan, as every Python process performs thousands of file system lookups to import its modules and shared libraries. We addressed this potential file storm in three ways: 1) We employ Spindle [42], a library that intercepts shared library loads, serves them from a central server, and also serves as a negative file system metadata cache, avoiding repeated failed lookups. 2) We enable Spindle's Python support, which provides similar capabilities for Python module loading. 3) We ensure the Python multiprocess context preloads relevant libraries and modules to reduce the need to reload them after forking. This requires careful management to prevent early initialization of ROCm and other libraries to avoid crashes.

Rabbit Near-node Storage. For each structure prediction, ~380 GiB of static auxiliary data is needed, including structure templates, model parameters, and protein databases. This data must be stored on Rabbit storage; the many small accesses are unsuitable for Lustre at scale, and using memory would significantly reduce capacity for other work. We extensively redesigned the I/O and data components of the workflow to support staging both persistent and transient data on a Rabbit-provided node-local file system mount. Each Rabbit is configured through Flux (§VI-B) to provide a 500 GiB read-only shared file system and a 600 GiB private XFS file system mounted on each node over its dedicated PCIe link. The file systems are striped across the 16 NVMe drives in the Rabbit; while this does result in some interference when different nodes write to their private file systems provided by the same Rabbit, the overall throughput improvement outweighed this. Each shared file system is pre-populated with the static data, staged by the Rabbit processor during startup, and shared by all nodes attached to the Rabbit, eliminating redundant storage.

Throughout workflow execution, the on-node scheduler tracks space usage on the Rabbit and ensures tasks are only started if the space they require is available. This enables us to offload almost all of the I/O load from the workflow to the Rabbits without consuming precious on-node memory.

D. Fault Tolerance and Resilience

Given the scale of El Capitan and the long runtimes of distillation workflows, failures are expected and must be handled; further, given startup overheads, any failures must be isolated and work redistributed without restarting.

While each node executes independently in our workflow, the Flux scheduler used by El Capitan [43] runs a broker process on every node, which participates in its tree-based overlay network (TBON). Should the broker on a node fail, the entire subtree of nodes rooted at that broker also fails. We adjust the topology of Flux's TBON to use a "mincrit" topology, which minimizes the number of critical nodes (nodes which cannot fail) while ensuring the tree fanout does not exceed 1024 at any level to maintain good TBON performance. We then dedicate the critical nodes (at most eleven in our runs) entirely to Flux and do not run any work on them, to mitigate the increased resource requirements of the brokers in this configuration and minimize any risk of application-level faults.

We use Flux's event subscription system to monitor for node failure events. When Flux detects one, a background process identifies the unfinished work that node was assigned, and automatically redistributes it among all remaining nodes. This strategy prevents any tasks from being missed due to failures.

Additionally, we harden the workflow against other transient on-node failures by monitoring for crashes and restarting tasks. Our inference servers run in separate processes specifically to isolate HIP errors, which can have process-wide impacts. Finally, we automatically reattempt data staging in case of early node failures or network issues during startup.

TABLE II
FLOP MODEL FOR OPENFOLD 2 INFERENCE

Component	Flop count†
F_I (input embed)	$16640 \cdot S^2 + 25088 \cdot MS + 22528 \cdot S$
F_R (recyc. embed)	$3843 \cdot S^2$
F_T (template embed)	$1568 \cdot S^3T + 456211 \cdot S^2T + 32832 \cdot S^2 + 29184 \cdot ST$
F_X (extra MSA)	$6208 \cdot S^3 + 9288 \cdot S^2X + 4998144 \cdot S^2 + 536992 \cdot SX + 32768 \cdot S$
F_{Evo} (Evoformer)	$48S(1552 \cdot S^2 + 1032 \cdot M^2 + 3090 \cdot SM + 2064 \cdot MT + 2392576 \cdot M + 3090 \cdot ST + 1249536 \cdot S + 1032 \cdot T^2 + 2392576 \cdot T) + 196608 \cdot S$
F_S (structure module)	$59904 \cdot S^2 + 30147288 \cdot S$
F_H (heads)	$16384 \cdot S^2 + 143872 \cdot S$
Total	$(R+1) \cdot (F_I + F_R + F_T + F_X + F_{Evo} + F_S) + F_H$

† For sequence length S, T templates, M MSA sequences, X extra MSA features, and R recycling iterations.

VI. How Performance Was Measured

Due to the complex nature of our workflow, performance measurement is challenging. In particular, OpenFold 2 runtime and flop count depend on the protein sequence length, which varies greatly (Fig. 2). We define three figures of merit:

1) Structures per second: The number of structures successfully predicted and written to Lustre per second, which directly measures progress on producing the distillation dataset; however, it can be inflated, as short monomers are significantly faster to predict than long monomers. There are at most two structures predicted per sequence (one for each model inference).

2) Atoms per second: The total atoms in successfully predicted structures per second, which better reflects sequence lengths.

3) Inference flop/s: The flop/s achieved for *model inference only*, excluding relaxation and other workflow operations. As precisely measuring flop counts in large runs is infeasible, we estimate them with our performance model (§VI-A).

Unless noted, our timing includes startup time (Rabbit configuration, job launch, data staging, unpacking, etc.) and I/O.

For smaller studies, we randomly selected ten sequences from each of the length bins in Fig. 2 (our "small dataset"), excluding length 10,000+ sequences (of which there are only 226). OpenFold 2 inference performance is given as the median of 5 runs. For individual kernel run time, measurements are the median of 10 runs with 10 additional warm-up steps.

A. Symbolic Performance Model

While standard approaches exist for estimating transformer flop counts [44], [45], they are insufficient for OpenFold, due to the model's complexity, its graph-oriented features, custom operations, and sequence length dependence. We therefore wrote a symbolic performance model as a modularized Python script using SymPy [46], which allows it to be parameterized by architectural features and the sequence length S.

Table II shows our general flop model for OpenFold 2. For the base model with templates (set $T = 0$ for no templates), the expression can be simplified to:

$$347{,}904 \cdot S^3 + 764{,}022{,}844 \cdot S^2 + 300{,}965{,}502{,}304 \cdot S.$$

The cubic dependency on sequence length supports our choice to emphasize long monomers with our atoms/s metric.

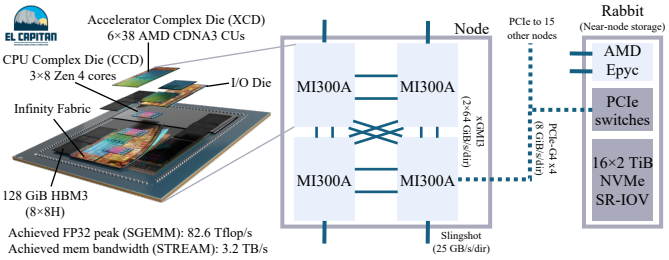


Fig. 4. Overview of the MI300A APU, an El Capitan node, interconnects, and the Rabbit near-node storage.

While this model is limited to inference (it does not model feature preprocessing, relaxation, nor the rest of the workflow), it is reasonably accurate. To validate it, we compared our predicted flop count with `rocprof`'s measured FP32 flops for our small dataset, and found it undercounts flops by ~1.8% overall (max 13.7% for short sequences).

B. El Capitan and the MI300A Architecture

El Capitan is the most recent Advanced Technology System of the Advanced Simulation and Computing Program of the National Nuclear Security Administration of the U.S. Department of Energy, sited at Lawrence Livermore National Laboratory. It has been ranked #2 on the Top500 list since June, 2026, (and was #1 for November, 2024, through November, 2025) with a sustained performance of 1.809 Eflop/s on HPL [10] and a sustained single-precision (FP32) performance of 3.69 Eflop/s. The system contains 11,520 nodes each with four AMD MI300A Accelerated Processing Units (APUs), for a total of 46,080 APUs, interconnected with HPE Slingshot. Figure 4 provides details of the MI300A and the node architecture. In particular, each APU has 128 GiB of HBM3 memory, for a total of 512 GiB of memory per node; no DRAM is present. This memory is coherent and physically shared between the CCDs and XCDs on the APU. Each APU has FP32 vector and matrix core units with 122.6 Tflop/s peak performance.

Data persistence on El Capitan is supported by capacity and performance tiers. Merced, a 401 PiB hard disk drive-based Lustre file system serves as the capacity tier. The performance tier consists of 720 HPE "Rabbit" modules [47], each containing 16 2 TiB NVMe SR-IOV drives. A Rabbit is connected to the Slingshot network and has PCIe connections to 16 compute nodes, allowing direct storage access via private or shared XFS file systems. Further, the Rabbit processor supports scheduling data ingestion and unpacking archives to place data in shared file systems for attached compute nodes.

El Capitan uses the Flux resource manager [43] for both job management and Rabbit configuration.

VII. Performance Results

We now evaluate the performance of our workflow. First, we show the overall benefit of our inference optimizations for single-node performance and analyze the performance of our kernel optimizations. We then conduct large-scale scaling and performance studies to evaluate overall performance.

Baseline. Our baseline is the public OpenFold 2 implementation, adapted to run on AMD platforms by the OF2 team. We

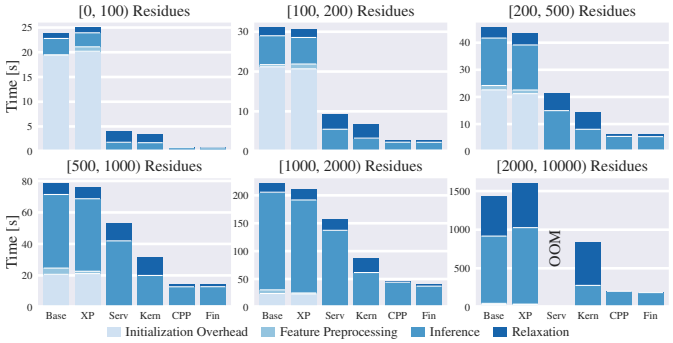


Fig. 5. Stacked bar plot depicting the inference timing breakdown following performance optimizations, in sequence length bins. Versions from left to right: **Base** = Baseline implementation, **XP** = XCD preprocessing, **Serv** = inference server, **Kern** = optimized kernels, **CPP** = C++ OpenFold model and relaxation, **Fin** = Final ElMerFold implementation.

evaluated the implementation on our small dataset (§VI) and on larger subsets, including small-scale scaling studies (up to 512 nodes of El Capitan, staging data to Rabbits). The median throughput was one structure per 75.5 s per node. For baseline runtime at scale, we extrapolated this performance assuming perfect scaling. We note this is an optimistic assumption which gives a conservative lower bound on our speedup.

A. *OpenFold 2 Inference Performance*

Figure 5 summarizes the on-node workflow performance. It provides a runtime breakdown of the inference process captured on the "small dataset". The breakdown shows the impact of certain optimizations varies greatly across the length bins.

The second version of the code (XP), which moves feature preprocessing to the APU XCDs, significantly improves preprocessing performance on all but the shortest sequences, with a median speedup of 4.21× for the longest sequences.

The third iteration of performance engineering (Serv) focuses on the switch to the server-based architecture. While eliminating Python and ROCm initialization and parameter loading overhead, none of the proteins in the final bin could run due to an out-of-memory error. The fourth version (Kern), in which we introduce the fused kernels that reduce activation memory, remediates this issue.

The fused kernels yield a noticeable improvement across all bins. Since most compute time was within model inference, this optimization both improves the overall runtime by 4.53× on the largest bin and enables computing the largest bin (as well as longer sequences, as Fig. 6 demonstrates).

Introducing the PyTorch-free `of2.cpp` and the C++ relaxation port (CPP) yield significant performance improvements. For inference, the gains are most pronounced on the smallest, launch-bound bin (2.7×). The larger bins also exhibit higher performance due to the improved cross-module fusion strategies. For relaxation, the fifth version delivers speedups of 8.5× for short sequences and up to 90× for the longest sequences. On the smaller bins (under 20,000 atoms), the speedup results from the lean runtime, HIP evaluation kernels, and zero format conversions, whereas the larger bins also benefit from the shorter optimization process due to the cutoff and improved starting conditions (§V-A).

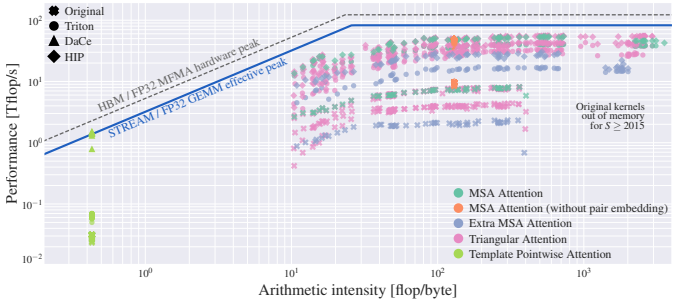


Fig. 6. Roofline plot of the different attention kernels in OpenFold 2.

Our final optimizations, which consist of tuning kernels and GEMM implementation selection heuristics, contribute a modest (5–8%) improvement across the board.

When weighted by the bin distribution (i.e., if each bin's result is multiplied by its overall size), the average normalized speedup over the baseline is **15.2**×. We now proceed to provide a more extensive evaluation of the fused kernels via an in-depth kernel performance analysis.

B. *Kernel Performance*

Our kernel optimization innovations were attained for two core computational operations—attention (on its five variants in OpenFold 2) and layer normalization.

Layer Normalization. For layer normalization, which is a memory-bound kernel, our performance improvements result from two modifications: 1) fusing the transposed, noncontiguous parallel reads into the normalization kernel itself; and 2) using the specific tensor dimensions to load memory once and retain it in the compute units across mean and variance computations. On a protein with sequence length 887, the performance improvements range between 1.72 and 7.36× (median 4.89×) per kernel.

MSA and Triangular Attention. Attention operators account for about 61% of the inference runtime. Thus, several optimizations and bespoke implementations target them (§V-A). We use the symbolic attention modules in our performance model (§VI-A) and a roofline analysis to guide our optimizations of the kernel. Figure 6 depicts the overall optimization efforts on attention kernels throughout all "small dataset" sequences and the four variants (the original OpenFold 2 version using rocBLAS, our Triton, DaCe, and bespoke HIP kernels).

We first run the STREAM benchmark [48] and square FP32 matrix multiplications of size 16,384×16,384 to obtain the effective (or achievable) peak. STREAM yielded 3.2 TB/s memory BW and rocBLAS yielded 82.62 Tflop/s. We annotate both the hardware (§VI-B) and the effective peak in Fig. 6.

As shown in the figure, the kernels can be memory- or compute-bound, depending on the sequence length. Our fused implementation and tile size heuristics considerably improve performance compared with the original implementation. For compute-bound kernels, Triton performance converges to 46% of the effective flop/s peak, whereas our bespoke HIP kernels converge to 68%, the latter bounded by the softmax operation between the two constituent matrix multiplications.

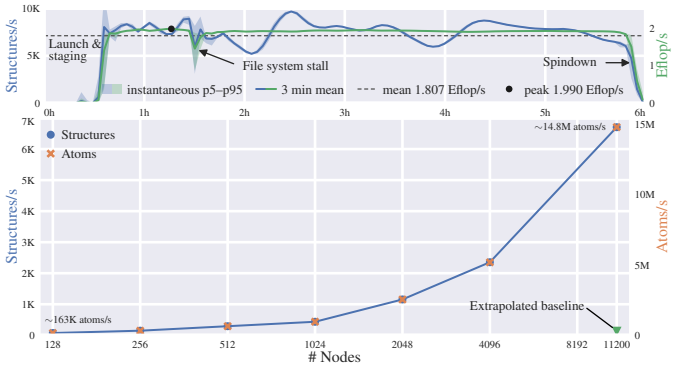


Fig. 7. Top: Timeline of a 6 h run on 11,200 nodes showing predicted structures and flop/s (in 3 min bins). Bottom: Sustained predicted structures and atoms per second achieved. All runs include startup and I/O time.

We also observe that the XCDs remain underutilized, even when the bespoke HIP kernels are used. Underutilization effects are, however, mitigated in the overall workflow by oversubscribing the XCDs (§V-B).

Pointwise Attention. Figure 6 shows that both the OpenFold 2 and Triton versions of pointwise attention fall short of the memory bound. At 0.43 flop/byte, Triton's lack of control over memory scheduling led to our DaCe kernel (§V-A), which uses DaCe's optimized cooperative memory movement capabilities.

We focus on a single instance of the kernel. Since the kernel is performed on batches of size S^2, chunking is almost always performed. We thus measure the performance of our heuristically tuned chunk size, 65,536, which is the most common occurrence of the kernel configuration in our runs. The figure indicates that the DaCe kernel, running at 3.23 TB/s, achieves the effective STREAM peak. For reference, the Triton implementation attains 149.42 GB/s on the same size. Having achieved the roofline, further optimization of the pointwise attention kernel would not be beneficial, and attempts to rewrite it directly in HIP resulted in similar or lower performance.

C. Sustained Workflow Performance

We evaluated the complete workflow at scale on El Capitan, with the largest run using 11,200 nodes (44,800 APUs) for 6 hours. Figure 7 (bottom) shows the strong scaling for generating the OpenFold 3 monomer distillation dataset in terms of atoms and structures/s, achieving **6718 structures/s** for the largest

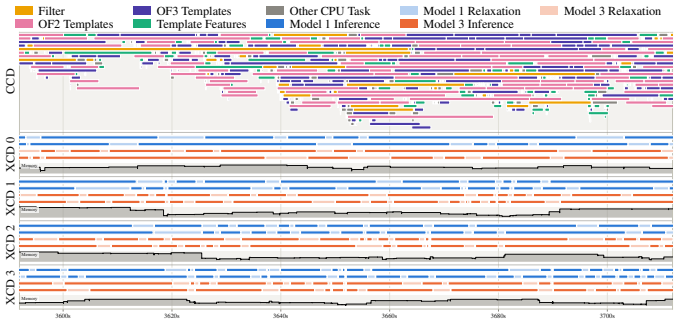


Fig. 8. Timeline snapshot from a single El Capitan node running ElMerFold, illustrating the diversity of task kind and memory requirements active at any one time. Boxes indicate the duration of tasks, and the colors correspond to different kinds of tasks (filtering, templates, inference, etc.).

TABLE III
RUNTIME AND THROUGHPUT SPEEDUP FOR k CONCURRENT INFERENCES FOR REPRESENTATIVE SEQUENCE LENGTHS (SPEEDUP IS $k\,t_1/t_k$)

	Wall-Clock Time (s)		
S	$k=1$	$k=2$	$k=4$
52	0.13	0.16 (1.61×)	0.23 (2.23×)
152	1.29	2.34 (1.10×)	4.54 (1.13×)
442	5.96	11.58 (1.03×)	23.07 (1.03×)
702	13.10	25.38 (1.03×)	50.73 (1.03×)
1158	34.68	68.02 (1.02×)	135.68 (1.02×)

run. Model inference reached a single-precision **peak of 1.99 Eflop/s** in this run, and **1.81 Eflop/s sustained** over the entire run. Our sustained performance is very close to our peak performance throughout much of the run, dropping only during startup, job drain, and a transient file system stall. The parallel efficiency is 91% when measured in both structures/s and atoms/s. While each node has independent computation, the workflow is loosely coupled through data operations on the Lustre filesystem, as well as the consequent network and metadata server traffic, so perfect scaling is not expected; our orchestration system is nevertheless able to hide much of this. We also annotate the extrapolated performance of the baseline code, assuming it scaled perfectly to 11,200 nodes; despite this optimistic assumption, the ElMerFold workflow is **45.4× faster**. This exceeds the 15.2× speedup of our inference and relaxation optimizations (§VII-A) due to the additional workflow optimizations (more aggressive overlap, etc.).

Figure 8 illustrates this with an execution timeline snapshot for a single node: the on-node orchestration ensures many tasks execute simultaneously, ensuring CCDs and XCDs are utilized and no particular workflow stage or input sequence becomes a bottleneck to inference. Our optimizations to other workflow components, particularly memory reductions, show their benefit here, as they enable more concurrent work, providing a deeper pipeline and buffer to absorb transient system stalls without idling XCDs. Throughout the snapshot (and, indeed, the sustained portion of the full run), the XCDs never fall idle. We also see the effect of our online memory estimator, which limits concurrent execution on the XCDs when long sequences require large amounts of memory.

In Table III, we quantify the effect of concurrent execution for representative sequence lengths, showing that while this increases the latency of each inference, the throughput of running k concurrent inferences always exceeds what we would obtain by running the inferences sequentially. This particularly benefits short sequences, which underutilize the APU (e.g., $S = 52$ uses only 28–45% of the CUs). Note that in a production run, concurrent inferences are usually of different lengths and very long sequences are not co-scheduled due to memory. Despite this, we find these results to be consistent with our run.

Using Rabbits was critical to achieving this performance. We evaluated our workflow without them, keeping all data on Lustre, but it achieved a peak performance of 0.33 structures/s on 16 nodes (0.14 structures/s on 32 nodes), and completed zero structures when using 128 nodes, due to the massive I/O load. Additionally, after the initial spinup of the workflow on

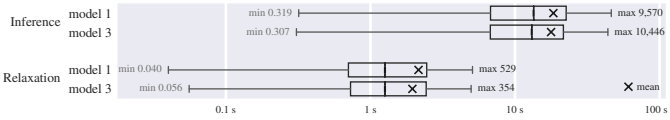


Fig. 9. Notched boxplots showing the distribution of model inference and relaxation runtimes from our 11,200-node run.

a node, the loading of MSAs from Lustre and the saving of predicted structures back to Lustre is entirely overlapped by other workflow tasks, excluding prolonged file system stalls.

Figure 7 (top) additionally shows a timeline of the number of structures predicted and ongoing inference flop/s throughout our largest run. Staging data to the Rabbits took 125 s (2.6 TiB/s peak aggregate Lustre bandwidth) and unpacking it a median of 174 s. Following this, we begin to launch jobs to nodes using `flux bulksubmit`; to avoid overwhelming the file system and Spindle, startup is staggered over ~7 min (in preliminary studies without our startup optimizations (§V-C), Python would fail to start at scale even after several hours). Overall, it takes ~20 min from job submission to the first structure output. The workflow begins to spin up across El Capitan, and sustains 6718 structures/s throughout the run, with small fluctuations due to varying sequence length mixes. Finally, the job automatically reduces its workload as it detects the end of its allocation. Nine node failures occurred during the job, after which their remaining work was automatically reallocated among the remaining nodes (§V-D); other faults were contained to individual nodes and recovered from. There was one dip in performance, ~1.5 h into the run, due to a Lustre metadata server failure, which temporarily stalled MSA load and structure save operations.

Figure 9 reports the distribution of runtimes for model inference and relaxation across our run. As our performance model (§VI-A) would suggest, model 3 (i.e., without templates) is ~3% faster than model 1 due to the lack of templates. The distribution is also heavily skewed due to the cubic dependency on sequence length and long tail of longer sequences.

Discussion. ElMerFold's integrated suite of optimizations delivers high sustained and peak performance, and is able to maintain near-peak performance through the run once startup completes while recovering from unexpected system anomalies. We also see this stressing the system in novel ways; e.g., relative to the runtime in Fig. 5, job startup is a significant fraction. While this would be amortized by a longer duration run, it also identifies issues in the underlying system software that can be addressed in the future.

Our current work has focused on FP32 computation, as the validated OpenFold 2 weights are in FP32 and were not tuned for low-precision inference. It is not clear the extent to which inference can be performed in lower precision, as the predicted 3D coordinates of the atoms in protein structures can lose meaningful accuracy. We experimented with two mixed-precision approaches: 1) Performing GEMM computations in TF32, which we found to negatively impact the resulting structures. 2) A variant of the Ozaki method [49] using BF16 to emulate the FP32 computations, which required excess memory and did not achieve significant speedups in practice.

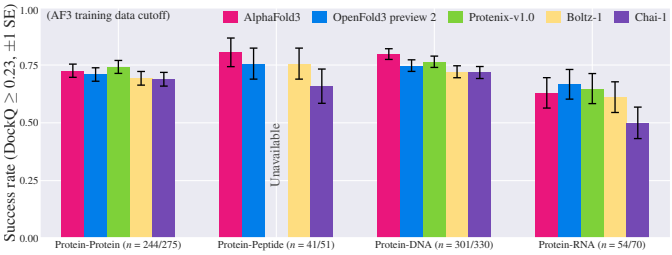


Fig. 10. Performance of structure prediction models on multimeric modalities of the FoldBench benchmark [50]. Figure courtesy of the OpenFold 3 team [51].

Within ElMerFold, there are several additional directions under consideration for further improvement of the workflow:

- As memory requirements for activations scale with $\mathcal{O}\left(S^2\right)$, and flops with $\mathcal{O}\left(S^3\right)$, model-parallel inference may be beneficial and would especially benefit distillation workflows with longer sequences (e.g., multimers or other complexes).
- In contrast, short monomers are memory-bound. The MI300A supports partitioning its XCDs into subgroups ("CPX" and "TPX" mode), which may enable additional performance relative to our current oversubscription approach.

VIII. Implications

The orchestration of many exaflops of compute on petabytes of data is a difficult task, and coupling that with a workflow and application that will produce truly useful scientific knowledge is extraordinarily challenging. In this effort, we assembled ElMerFold, an optimized workflow that uses innovations in high-performance computing to unlock innovations within scientific AI and computational biology. The El Capitan system was primarily designed for modeling and simulation codes, and this work has identified unique computational demands not exhibited by those applications. Taking a holistic approach that couples data management, I/O, and job resilience with the traditional task of optimizing to achieve as much of the attainable peak compute performance as is possible demonstrates new requirements that should be considered when designing future HPC systems, as well as a bright new path for AI at scale on a wider range of compute platforms.

A. *OpenFold 3 and the Benefits of Distillation Data*

The production of the OpenFold 3 monomer distillation dataset was the original impetus for ElMerFold, as detailed here. Compared to an initial projection of eight days using all of El Capitan—an optimistic estimate—ElMerFold can now produce the dataset in approximately four hours. Once the dataset was completed, it was used for training OpenFold 3, with a preview version available [51]. This version exhibits capabilities comparable to AlphaFold 3 and other reproductions for monomeric and multimeric protein structures and some other modalities (see Fig. 10, [51]), in a completely open model. Further, as detailed in [51], the inclusion of additional distillation data is most beneficial for particularly challenging modalities, such as for antibody/antigen docking: compared to an earlier preview of OpenFold 3 [52], which used only a subset of the monomer distillation data, the latest preview significantly improves mean DockQ for the AlphaFold 3 antibody-antigen

test set [5] (ranked ~0.12 to ~0.21, oracle ~0.26 to ~0.31). This distillation data has already benefited other open models, including OpenBind-0 [53] and OpenDDE [54].

We also note that the production of distillation data is not a one-time task. ElMerFold has been used to produce additional data to supplement the existing monomer distillation data, and can generalize to future models, such as OpenFold 3.

B. Distillation Workflows for Scientific Machine Learning

Indeed, distillation data is not just limited to protein structures. The primary challenge in developing novel machine learning models for many scientific fields (e.g., physics, chemistry, biology) is the availability of high-quality data. While experimental measurement and high-precision simulations remain the gold standard for data, the ability to distill knowledge from a model serves as a powerful complement to bootstrap more capable models. AlphaFold 3 and OpenFold 3 serve as key examples of this, and the associated distillation workflows are representative of the complexity, with multiple ML models combined with traditional modeling and simulation toolkits, of current and future data generation workflows.

C. Agentic Coding for Performance Engineering

Over the course of the the ElMerFold project, the capabilities of agentic coding systems significantly improved, and the ElMerFold team made extensive use of them, primarily Claude Fable 5 and GPT-5.6 Sol. While unable to "one-shot" any significant optimizations, under expert guidance and with extensive review, such agents serve as an incredible force multiplier for performance engineering, autonomously implementing, profiling, and validating potential optimizations in hours that would have taken a human days or weeks. Enabling this requires the agent to have access to the target hardware for benchmarking, comprehensive benchmark and test suites, and tooling (e.g., profilers, debuggers) in order to provide the agent with rapid feedback to iterate on. For example, in ElMerFold, the HIP attention kernels, `of2.cpp` port, and C++ relaxation were implemented this way. This work provides a demonstration that teams can use these workflows to produce high-performance code specialized to their particular problem characteristics and hardware platforms.

D. Rethinking HPC Systems Design in Light of Scientific ML

Higher-level abstractions for performance. Several of our key inference optimizations used high-level, domain-specific languages (Triton, DaCe) to implement kernels, supplemented by bespoke intrinsics-level code and inline assembly to deliver the best possible performance. Abstractions have multiple advantages over lower-level implementations in languages like CUDA or HIP: they require significantly less effort to implement; the same implementation can run on diverse architectures (e.g., AMD and NVIDIA) and generations (e.g., AMD MI250X to MI300A); and implementations can easily take advantage of compiler and language improvements without modification. This mitigates "hardware lottery" [55] effects and improves out-of-the-box performance on new hardware for many applications. However, these abstractions can leave performance on the table compared with an optimal implementation, and our work identifies relevant gaps. We are currently working on incorporating our findings (e.g., Buffer Resource Descriptors) into DaCe to generalize these optimizations for other applications. Other bespoke optimizations that required circumventing the compiler create a unique co-design opportunity with the AMD compiler team, improving future versions of HIP and `hipcc`.

Rabbits enable flexible I/O offload. Many scientific AI applications require significant auxiliary and intermediate data to operate, and often have complicated access patterns (e.g., the genetic database searches conducted in ElMerFold). Global parallel file systems do not scale for these workloads. The Rabbit near-node storage systems on El Capitan provide an additional, flexible tier in the storage hierarchy to accommodate these workloads, while balancing overall system cost by not overprovisioning resources (e.g., node local NVMe) that other workloads do not require. On El Capitan specifically, where the MI300A APUs lack DRAM, the Rabbits serve as a second tier of storage, demonstrating the effectiveness of flexible I/O accelerators for adapting existing hardware to new workloads. Future uses of Rabbits may also extend their integrated compute capability to support on-demand asynchronous data movement and preprocessing for applications.

Scientific ML workflows have diverse compute requirements. Large-scale supercomputers have historically been designed around the needs of modeling and simulation codes; more recently, dedicated "AI supercomputers" have also been designed. These workloads have very different characteristics (e.g., precision requirements, accelerator support, etc.); yet, as ElMerFold demonstrates, scientific distillation data workflows closely couple both deep learning models and more traditional tooling. Splitting these onto different systems would impose major performance hits and require storing or moving large amounts of intermediate data; these concerns will be especially acute for agent-driven workflows [56]. These workflows will necessitate balanced systems capable of both low- and high-precision flops and CPU and GPU acceleration.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-2004815). Nikoli Dryden, Tal Ben-Nun, Pier Fiedorowicz, Tom Benson, & Brian Van Essen were funded in part by LLNL LDRD #24-SI-008. Nikoli Dryden was funded in part by LLNL LDRD #25-ERD-042. Vinay Swamy was funded by NLM grant T15LM007079. Some funding provided by NIH grant 1R35GM150546. Experiments were performed at the Livermore Computing facility. The authors thank Brian Behlendorf, Adam Bertsch, Ryan Day, Scott Futral, Elsa Gonsiorowski, John Gyllenhaal, Kalan Kier, Matt LeGendre, Py Watson, and the many other Livermore Computing staff and administrators who supported this work; HPE, AMD, and the El Capitan Center of Excellence; Simone Conti; the Flux team, especially Mark Grondona; and the OpenFold team.

References

[1] K. A. Dill, S. B. Ozkan, M. S. Shell, and T. R. Weikl, "The protein folding problem," *Annu. Rev. Biophys.*, vol. 37, no. 1, 2008.

[2] R. Pearce and Y. Zhang, "Toward the solution of the protein structure prediction problem," *J. Biological Chemistry*, vol. 297, no. 1, 2021.

[3] A. W. Senior *et al.*, "Improved protein structure prediction using potentials from deep learning," *Nature*, vol. 577, no. 7792, 2020.

[4] J. Jumper *et al.*, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, no. 7873, 2021.

[5] J. Abramson *et al.*, "Accurate structure prediction of biomolecular interactions with AlphaFold 3," *Nature*, vol. 630, no. 8016, 2024.

[6] R. Evans *et al.*, "Protein complex prediction with AlphaFold-Multimer," *bioRxiv*, 2021.

[7] M. Baek *et al.*, "Accurate prediction of protein structures and interactions using a three-track neural network," *Science*, vol. 373, no. 6557, 2021.

[8] G. Ahdritz *et al.*, "OpenFold: Retraining AlphaFold2 yields new insights into its learning mechanisms and capacity for generalization," *Nature Methods*, vol. 21, no. 8, 2024.

[9] DeepMind, 2024. [Online]. Available: https://github.com/google-deepmind/alphafold3/blob/main/WEIGHTS_TERMS_OF_USE.md

[10] TOP500, "June, 2026, TOP500 list," 2026. [Online]. Available: https://top500.org/lists/top500/2026/06/

[11] V. Zambaldi *et al.*, "De novo design of high-affinity protein binders with AlphaProteo," *arXiv preprint arXiv:2409.08022*, 2024.

[12] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *NeurIPS Deep Learning and Representation Learning Workshop*, 2014.

[13] A. Grattafiori *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[14] Y. Bai *et al.*, "Constitutional AI: Harmlessness from AI feedback," *arXiv preprint arXiv:2212.08073*, 2022.

[15] L. Richardson *et al.*, "MGnify: the microbiome sequence data analysis resource in 2023," *Nucleic acids research*, vol. 51, no. D1, 2023.

[16] Chai Discovery Team *et al.*, "Chai-1: Decoding the molecular interactions of life," *BioRxiv*, 2024.

[17] J. Wohlwend *et al.*, "Boltz-1: Democratizing biomolecular interaction modeling," *BioRxiv*, 2025.

[18] S. Passaro *et al.*, "Boltz-2: Towards accurate and efficient binding affinity prediction," *BioRxiv*, 2025.

[19] "HMMER: biosequence analysis using profile hidden Markov models," 2024. [Online]. Available: http://hmmer.org/

[20] M. Steinegger *et al.*, "HH-suite3 for fast remote homology detection and deep protein annotation," *BMC Bioinformatics*, vol. 20, 2019.

[21] M. Steinegger and J. Söding, "MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets," *Nature biotechnology*, vol. 35, no. 11, 2017.

[22] V. Hornak *et al.*, "Comparison of multiple Amber force fields and development of improved protein backbone parameters," *Proteins: Structure, Function, and Bioinformatics*, vol. 65, no. 3, 2006.

[23] P. Eastman *et al.*, "OpenMM 8: molecular dynamics simulation with machine learning potentials," *The Journal of Physical Chemistry B*, vol. 128, no. 1, 2023.

[24] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[25] J. Ansel *et al.*, "PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *ASPLOS*, 2024.

[26] S. L. Song *et al.*, "DeepSpeed4Science initiative: Enabling large-scale scientific discovery through sophisticated AI system technologies," *arXiv preprint arXiv:2310.04610*, 2023.

[27] T. Dao *et al.*, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," *Advances in Neural Information Processing Systems*, vol. 35, 2022.

[28] T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," in *ICLR*, 2023.

[29] S. Cheng *et al.*, "FastFold: Optimizing AlphaFold training and inference on GPU clusters," in *PPoPP*, 2024.

[30] F. Zhu *et al.*, "ScaleFold: Reducing AlphaFold initial training time to 10 hours," in *Design Automation Conference*, 2024.

[31] W. Kwon *et al.*, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[32] "Ollama," 2025. [Online]. Available: https://ollama.com/

[33] "llama.cpp," 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp

[34] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning I/O," in *Supercomputing*, 2021.

[35] C. Zhang, M. Shine, A. M. Pyle, and Y. Zhang, "US-align: universal structure alignments of proteins, nucleic acids, and macromolecular complexes," *Nature Methods*, vol. 19, no. 9, 2022.

[36] Y. Zhang and J. Skolnick, "Scoring function for automated assessment of protein structure template quality," *Proteins: Structure, Function, and Bioinformatics*, vol. 57, no. 4, 2004.

[37] Lawrence Livermore National Laboratory, "LBANN GitHub Repositories," 2025. [Online]. Available: https://github.com/lbann

[38] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

[39] T. Ben-Nun *et al.*, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Supercomputing*, 2019.

[40] AMD, "AMD Instinct MI300 Instruction Set Architecture," 2025. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf

[41] ——, "Using thread trace," 2026. [Online]. Available: https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/develop/how-to/using-thread-trace.html

[42] W. Frings *et al.*, "Massively parallel loading," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013.

[43] D. H. Ahn *et al.*, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, vol. 110, 2020.

[44] J. Kaplan *et al.*, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[45] M. Isaev, N. McDonald, L. Dennison, and R. Vuduc, "Calculon: a methodology and tool for high-level co-design of systems and large language models," in *Supercomputing*, 2023.

[46] A. Meurer *et al.*, "SymPy: Symbolic computing in Python," *PeerJ Computer Science*, vol. 3, 2017.

[47] H. Devarajan *et al.*, "Chasing the rabbit: A systematic exploration of software-defined HPC storage," in *2026 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2026.

[48] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[49] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, "DGEMM using tensor cores, and its accurate and reproducible versions," in *International Conference on High Performance Computing*, 2020.

[50] S. Xu *et al.*, "Benchmarking all-atom biomolecular structure prediction with FoldBench," *Nature Communications*, 2025.

[51] OpenFold3 Team, "OpenFold3-preview2 technical report," 2026. [Online]. Available: https://portal.openfold.omsf.io/reports/of3p2_technical_report.pdf

[52] ——, "OpenFold3-preview technical report," 2025. [Online]. Available: https://github.com/aqlaboratory/openfold-3/blob/main/assets/of3p1_technical_report.pdf

[53] OpenBind Consortium, "OpenBind-0: Advancing Open Molecular Structure Prediction," 2026. [Online]. Available: https://openbind.uk/news/blog-openbind-0-advancing-open-molecular-structure-prediction/

[54] Aureka AI OpenDDE Project, "Folding, reasoning, and scaling with open-source drug discovery engine," *arXiv preprint arXiv:2607.03787*, 2026.

[55] S. Hooker, "The hardware lottery," *CACM*, vol. 64, no. 12, 2021.

[56] H. Menon *et al.*, "Multi-agent collaboration for automated design exploration on high performance computing systems," *arXiv preprint arXiv:2603.11515*, 2026.