Received 2 June 2023, accepted 19 June 2023, date of publication 27 June 2023, date of current version 11 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3289836

RESEARCH ARTICLE

Scalable Empirical Dynamic Modeling With Parallel Computing and Approximate k-NN Search

KEICHI TAKAHASHI[1], (Member, IEEE), KOHEI ICHIKAWA[2], (Member, IEEE), JOSEPH PARK[3], (Senior Member, IEEE), AND GERALD M. PAO[4]
[1]Cyberscience Center, Tohoku University, Sendai 980-8578, Japan
[2]Nara Institute of Science and Technology, Ikoma, Nara 630-0192, Japan
[3]Comprehensive Nuclear-Test-Ban Treaty Organization, 1400 Vienna, Austria
[4]Okinawa Institute of Science and Technology, Onna, Okinawa 904-0412, Japan

Corresponding authors: Keichi Takahashi (keichi@tohoku.ac.jp) and Gerald M. Pao (gerald.pao@oist.jp)

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant JP20K19808; and in part by the Joint Usage/Research Center for Interdisciplinary Large-Scale Information Infrastructures (JHPCN), Japan, under Project jh220050.

ABSTRACT Empirical Dynamic Modeling (EDM) is a mathematical framework for modeling and predicting non-linear time series data. Although EDM is increasingly adopted in various research fields, its application to large-scale data has been limited due to its high computational cost. This article presents kEDM, a high-performance implementation of EDM for analyzing large-scale time series datasets. kEDM adopts the Kokkos performance-portable programming model to efficiently run on both CPU and GPU while sharing a single code base. We also conduct hardware-specific optimization of performance-critical kernels. kEDM achieved up to 6.58× speedup in pairwise causal inference of real-world biology datasets compared to an existing EDM implementation. Furthermore, we integrate multiple approximate k-NN search algorithms into EDM to enable the analysis of extremely large datasets that were intractable with conventional EDM based on exhaustive k-NN search. EDM-based time series forecast enhanced with approximate k-NN search demonstrated up to 790× speedup compared to conventional Simplex projection with less than 1% increase in MAPE.

INDEX TERMS Empirical dynamic modeling, high-performance computing, time-series analysis, performance portability, high-performance data analytics.

I. INTRODUCTION

Empirical Dynamic Modeling (EDM) [1] is a mathematical framework for modeling and predicting non-linear time series data. Although EDM is increasingly adopted in various fields such as neuroscience [2], ecology [3], medicine [4] and geophysics [5], applications to large datasets have been limited due to the high computational cost. Since current EDM implementations are not designed with performance as a primary goal, several studies [6], [7] improved the EDM algorithm or implementation to accelerate the computation. We developed mpEDM [8], a prototype implementation of EDM that utilizes GPU-centric supercomputers, by taking advantage of both intra- and inter-node parallelism, and GPU offloading.

Although mpEDM demonstrated massive speedup over a conventional implementation, several challenges remain. First, mpEDM maintained different implementations for different hardware such as CPUs and GPUs. This design is simple, but incurs high maintenance costs and requires development effort when porting it to a novel hardware. Second, some parts of the EDM computation could not be efficiently implemented using ArrayFire, the library we used to offload computation to the GPU. Thus, the computation could not be fully executed on the GPU. Furthermore, we used exhaustive k-nearest neighbor (k-NN) search in mpEDM, but the cost of exhaustive k-NN search grows rapidly even if parallel hardware is employed.

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Alawneh.

In this article, we develop kEDM, a performance-portable implementation of EDM using the Kokkos programming model to address these challenges in mpEDM. By implementing all kernels with Kokkos, kEDM efficiently runs on diverse hardware while having a single code base. Furthermore, we integrate approximate k-NN search algorithms into EDM to enable the analysis of extremely large datasets that were intractable with exhaustive k-NN search.

This article is an extension to our previous conference paper [9] where we presented a prototype of kEDM and conducted a preliminary performance evaluation. The key updates from the conference paper are summarized as follows:

- We present a high-performance implementation of S-map, an EDM algorithm that quantifies the non-linearity of a dynamical system.
- We analyze the impact of each optimization technique we propose using microbenchmarks.
- We integrate approximate k-NN search algorithms into EDM to enable the analysis of extremely large datasets.

The rest of the article is organized as follows. Section II briefly introduces the basic concept of EDM and algorithms that we mainly target, which are Simplex projection, Convergent Cross Mapping, and S-map. Section III describes the design and implementation of kEDM, our performance-portable implementation of EDM based on the Kokkos programming model. Section IV evaluates the performance of kEDM using both synthetic and real-world datasets, analyzes the impact of performance tuning, and investigates the trade-off between speed and accuracy when using approximate k-NN search in EDM. Section VI concludes this article.

II. BACKGROUND

In this section, we first discuss the basic concept behind EDM. We then introduce popular EDM methods that we target: Simplex projection for short-term forecasts, S-map for quantification of non-linearity, and CCM for causal inference.

A. EMPIRICAL DYNAMIC MODELING

Empirical Dynamic Modeling (EDM) [1] is a mathematical framework for modeling non-linear time series data. It is based on the *Takens' theorem* [10], [11]. Takens' theorem states that the attractor manifold of a dynamical system can be reconstructed from time lags of time series observed from the dynamical system. Fig. 1 illustrates the concept of state space reconstruction. Fig. 1(a) shows the original state space of a dynamical system. Fig. 1(b) shows the state space reconstructed from time lags. Takens' theorem shows that given enough time lags, the reconstructed manifold preserves the topological features of the original manifold, *i.e.*, the reconstructed manifold is *diffeomorphic* to the true manifold.

1) SIMPLEX PROJECTION

Simplex projection is an EDM method that makes short-term forecasts [12]. Let $\mathbf{x} \in \mathbb{R}^L$ denote the *library* time

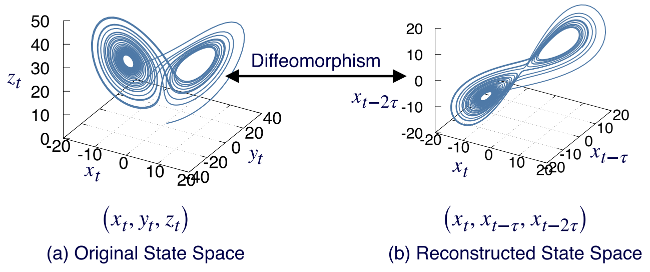


FIGURE 1. Concept of state space reconstruction.

series. The time-delayed embedding of a point $\mathbf{x}_t \in \mathbb{R}$ in the E-dimensional state space $\mathbf{X}_t \in \mathbb{R}^E$ is given as:

$$\mathbf{X}_t = \left(\mathbf{x}_t, \mathbf{x}_{t-\tau}, \ldots, \mathbf{x}_{t-(E-1)\tau}\right), \tag{1}$$

where τ is the time lag. For a given observation $\mathbf{y}_t$ and its E-dimensional time-delayed embedding $\mathbf{Y}_t = (\mathbf{y}_t, \mathbf{y}_{t-\tau}, \ldots, \mathbf{y}_{t-(E-1)\tau})$, Simplex projection predicts the T_p steps ahead value of $\mathbf{y}_{t+T_p}$. First, let us assume $\mathbf{n} \in \mathbb{N}^{E+1}$ is a vector of nearest neighbor indices. $\mathbf{n}_i$ is the time index of the i-th nearest neighbor of $\mathbf{X}_t$, and $\mathbf{X}_{\mathbf{n}_i}$ is the embedded vector of the i-th nearest neighbor. The Euclidean distance is used to measure the distance between embedded points, *i.e.*, $d(\mathbf{X}_i, \mathbf{X}_j) = \|\mathbf{X}_i - \mathbf{X}_j\|_2$. Simplex projection predicts the future of $\mathbf{y}_t$ by considering how the $E+1$ nearest neighbors of $\mathbf{X}_t$, which define an E-dimensional simplex, move over time. Specifically, the T_p steps ahead prediction of $\mathbf{y}_t$ is given as a weighted average of the $E+1$ nearest neighbors in the state space projected T_p steps ahead :

$$\hat{\mathbf{y}}_{t+T_p} = \sum_{i=1}^{E+1} \frac{\mathbf{w}_i}{\sum_{i=1}^{E+1} \mathbf{w}_i} \cdot \mathbf{X}_{\mathbf{n}_i+T_p}, \tag{2}$$

where

$$\mathbf{w}_i = \exp\left\{-\frac{d(\mathbf{X}_t, \mathbf{X}_{\mathbf{n}_i})}{\min\limits_{1\le j\le E+1} d(\mathbf{X}_t, \mathbf{X}_{\mathbf{n}_j})}\right\}. \tag{3}$$

Here, $\mathbf{w} \in \mathbb{R}^{E+1}$ are the weights assigned to each neighbor. The weights are assigned based on the distance between the predicted point and its neighbor, *i.e.*, closer neighbors are given higher weights.

2) S-MAP

Sequentially Locally Weighted Global Linear Maps (*S-map*) is another short-term forecast method [13]. Its main purpose is to estimate state space Jacobians and is used to quantify the non-linearity of a system. Unlike Simplex projection that uses the k-nearest library points of the predicted point in the state space, S-map uses *all* library points, weighting the points by their distances from the prediction point, and applying a localization kernel to the weights reflecting a specific state space localization of the linear mapping. This localization in the state space reflects the "locally weighted global linear maps".

Specifically, S-map builds a linear model for each predicted point as follows. Given a vector of regression coefficients $\hat{\mathbf{c}} \in \mathbb{R}^{E+1}$, the linear model is defined as:

$$\hat{\mathbf{y}}_{t+T_p} = \hat{\mathbf{c}}_1 + \sum_{i=1}^{E} \hat{\mathbf{c}}_{i+1} \mathbf{Y}_{t-i\tau}. \tag{4}$$

The model linearly combines lagged observations $\mathbf{Y}_t$ scaled by the regression coefficients $\mathbf{c}$. Note that $\mathbf{c}$ is a function of time step t and varies over time. The regression coefficients are estimated as a solution of the following least-squares problem:

$$\hat{\mathbf{c}} = \underset{\mathbf{c}}{\operatorname{argmin}} \|\mathbf{WAc} - \mathbf{Wb}\|_2^2, \tag{5}$$

$$\mathbf{A} = \begin{bmatrix} 1 & \mathbf{x}_{1-\tau} & \mathbf{x}_{1-2\tau} & \cdots & \mathbf{x}_{1-(E-1)\tau} \\ 1 & \mathbf{x}_{2-\tau} & \mathbf{x}_{2-2\tau} & \cdots & \mathbf{x}_{2-(E-1)\tau} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mathbf{x}_{N-\tau} & \mathbf{x}_{N-2\tau} & \cdots & \mathbf{x}_{N-(E-1)\tau} \end{bmatrix}, \tag{6}$$

$$\mathbf{b} = \begin{bmatrix} \mathbf{x}_{1+T_p} & \mathbf{x}_{2+T_p} & \cdots & \mathbf{x}_{N+T_p} \end{bmatrix}^\top, \tag{7}$$

$$\mathbf{W} = \operatorname{diag}(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_N), \tag{8}$$

where $\mathbf{w}$ are the weights defined as:

$$\mathbf{w}_i = \exp\left\{-\frac{\theta d(\mathbf{Y}_t, \mathbf{X}_i)}{\bar{d}(\mathbf{Y}_t, \mathbf{X}_i)}\right\}. \tag{9}$$

The parameter $\theta (\geq 0)$ is a localization factor that controls how much near neighbors are given higher weights than distant neighbors (*i.e.,* the degree of state dependence) in making a prediction. Note that if $\theta = 0$, S-map is equivalent to an autoregressive model. To quantify the non-linearity of a system, the prediction skill is evaluated with respect to θ. If the prediction accuracy increases with θ, the system is considered non-linear.

3) CONVERGENT CROSS MAPPING (CCM)

Convergent Cross Mapping (CCM) identifies and quantifies the causal interaction between two time series variables. [14] The generalized Takens' theorem [11] implies that it is possible to cross map between time series, *i.e.*, predict one time series from another, if both time series are observed from the same dynamical system. Given a pair of time series $\mathbf{x}$ and $\mathbf{y}$, CCM reconstructs a subsampled manifold from $\mathbf{x}$ and uses it to predict $\mathbf{y}$ with the Simplex projection or S-map. The causality is tested by gradually increasing the subset size and evaluating if the prediction accuracy of $\mathbf{y}$ (measured by Pearson's correlation of predicted and actual values) increases. If the prediction accuracy of $\mathbf{y}$ increases with the subset size of $\mathbf{x}$, it indicates that $\mathbf{y}$ left some information in $\mathbf{x}$, and thus it is considered $\mathbf{y}$ causes $\mathbf{x}$.

B. mpEDM

mpEDM[1] is a proof-of-concept implementation of EDM for GPU-based HPC systems we developed in our previous work [8]. Prior to developing mpEDM, we profiled cppEDM,[2] the *de facto* standard implementation, and identified that the k-NN search in the state space is the main bottleneck in most EDM methods. We thus offloaded the k-NN search to GPU using the ArrayFire [15] library. Using the AI Bridging Cloud Infrastructure (ABCI)[3] supercomputer, mpEDM demonstrated up to 1,530× speedup over cppEDM on a dataset recorded from zebrafish brain containing 53,053 time series each with 1,450 time steps.

Although mpEDM successfully accelerated EDM using HPC, several challenges remain. First, mpEDM maintained different implementations for different architectures. This design incurs high maintenance costs and requires development effort when porting mpEDM to a novel HPC hardware. Second, mpEDM was limited by ArrayFire. mpEDM used ArrayFire to offload computation to GPU. However, some parts of EDM computation, specifically the lookup of nearest neighbor points, could not be efficiently implemented using ArrayFire. Thus, the lookups were executed on the host CPU even if a GPU is available on the system.

III. PROPOSAL

In this section, we describe the design and implementation of kEDM. We first describe the overall design of kEDM, and introduce the Kokkos performance-portable programming model we use to implement kEDM. We then discuss the implementation of Simplex projection and S-map along with performance tuning. Lastly, we describe the integration of approximate k-NN search into EDM to enhance the scalability of EDM.

A. OVERALL DESIGN

To address the challenges in mpEDM, we develop a new implementation of EDM named kEDM.[4] kEDM uses the Kokkos [16], [17] performance portability framework, and runs on both CPUs and GPUs while sharing the same code base. Thus, porting to a new hardware can be completed with minimal effort. Furthermore, since all kernels including lookups of neighbors are implemented using Kokkos, the entire application can be executed on the GPU. We also include a Python binding based on pybind11.[5]

The runtime of exhaustive k-NN search becomes prohibitively long if the time series is extremely long even if high-performance CPUs or GPUs are employed. To address this problem, we take advantage of approximate k-NN search algorithms that yield massive speedup over exhaustive search with minor loss in accuracy.

B. KOKKOS

Kokkos [16], [17] is a performance-portable programming model. Kokkos is implemented as a C++ library, and allows

[1]https://github.com/keichi/mpEDM

[2]https://github.com/SugiharaLab/cppEDM

[3]https://abci.ai/

[4]https://github.com/keichi/kEDM

[5]https://github.com/pybind/pybind11

users to develop programs that run efficiently on diverse hardware. Parallel computation in Kokkos is expressed using a combination of three abstractions: (1) execution space, (2) execution pattern, and (3) execution policy. The execution space represents *where* the computation is executed. As of writing this article, Kokkos supports CUDA, HIP, HPX, OpenACC, OpenMP, OpenMP Target Offloading and SYCL as execution spaces. The execution pattern represents *what* the computation is, *i.e.*, computational kernel. The execution pattern is represented with one of `parallel_for`, `parallel_reduce` and `parallel_scan`, and a C++ functor.

The execution policy defines *how* the computation is executed. The most basic execution policy is `RangePolicy`, which is used to parallelize a loop that iterates over a one-dimensional range of indices. `TeamPolicy` and `TeamThreadRange` execution policies are used in conjunction to parallelize a nested loop. `TeamPolicy` is applied to the outer loop, and launches teams of threads. `TeamThreadRange` is applied to the inner loop, and launches threads within a team. The use of `TeamPolicy` and `TeamThreadRange` allows to take advantage the *hierarchical parallelism* of the underlying hardware: on a GPU, teams and threads map to thread blocks and threads, respectively, and on a CPU, teams map to physical cores and threads map to hardware threads. `ThreadVectorRange` is the third level of parallelism, which maps to individual vector lanes. `TeamPolicy` also gives access to scratch memory, which is an abstraction of fast on-chip memory, *e.g.*, shared memory on GPUs.

Views are fundamental data types in Kokkos that represent reference-counted multidimensional arrays. They are associated to memory spaces, an abstraction of where the data resides, and have polymorphic data layouts so that the memory layout can be transparently changed. By default, a view is stored row-major on CPU and column-major on GPU, so that accesses are blocked on CPU and coalesced on GPU.

Listing 1 shows a vector add kernel implemented in Kokkos. In this example, a `parallel_for` with `RangePolicy` iterates over the one-dimensional range $0 \leq i < n$. Listing 2 shows a matrix vector multiply kernel that uses hierarchical parallelism. The outer `parallel_for` with `TeamPolicy` launches a league of `m` teams that each computes one element in the output vector `y`. The inner `parallel_reduce` with `TeamThreadRange` computes the dot product between a row in `A` and `x`. Subsequently, a single thread in each team writes the dot product to the corresponding location in `y`.

In addition to Kokkos, we considered several performance portable programming models such as OpenMP Target Offloading, OpenACC and SYCL. We chose Kokkos because previous research [18], [19], [20] suggested that Kokkos delivers portable performance on a wider variety of devices compared to its alternatives. Furthermore, multiple production applications such as ExaWind [21], Albany [22]

```
parallel_for(RangePolicy<ExecSpace>(n),
KOKKOS_LAMBDA(int i) {
    y(i) = a * x(i) + y(i);
});
```

Listing 1. Vector addition using `RangePolicy`**.**

```
parallel_for(TeamPolicy<ExecSpace>(m, AUTO),
KOKKOS_LAMBDA(const member_type &member) {
    int i = member.league_rank();
    float sum = 0.0f;

    parallel_reduce(TeamThreadRange(member, n),
    [=] (int j, float &tmp) {
        tmp += A(i, j) * x(j)
    }, sum);

    single(PerTeam(member),
    [=] () { y(i) = sum; });
});
```

Listing 2. Matrix vector multiplication using `TeamPolicy` **and** `ThreadVectorRange`**.**

Uintah [23] and LAMMPS [24] have been successfully ported to Kokkos, demonstrating its practicality.

C. SIMPLEX PROJECTION

In our previous work [8], we identified that the k-NN search in the state space is the major bottleneck in Simplex projection. We thus design a parallel k-NN search using Kokkos. An exact (exhaustive) k-NN search is composed of calculating the distances between all embedded points, and then performing a top-k search to find the closest k points from each point.

Algorithm 1 is a pseudocode of the pairwise distances kernel. This kernel calculates the squared distance between every pair of embedded points $\mathbf{X}_t$. A trivial implementation would first embed the time series $\mathbf{x}$ into the state space and generate $\mathbf{X}$ as defined in Eq. (1). However, this approach incurs excessive memory accesses. We thus do not create the embedding $\mathbf{X}$ in the memory explicitly but directly compute the distances between embedded points from the original time series $\mathbf{x}$. The i-loop (lines 1–8 in Algorithm 1) and j-loop (lines 2–7) iterate over each element in $\mathbf{D}$ and calculate $\mathbf{D}_{i,j}$, *i.e.*, the distance between $\mathbf{X}_i$ and $\mathbf{X}_j$. The inner-most k-loop (lines 4–6) iterates over the embedded dimensions and accumulates the distance in each dimension into $\mathbf{D}_{i,j}$.

To parallelize this algorithm, we use hierarchical parallelism and assign the outer-most i-loop to thread teams and inner j-loop to individual threads. Taking advantage of the fact that $\mathbf{x}_{i-k\tau}$ is L times reused in the j-loop, we use team-private scratch memory to cache the values of $\mathbf{x}_{i-k\tau}$. On CPUs, vectorization is critical to attaining high performance. A simple approach would be to vectorize the inner-most k-loop by using `ThreadVectorRange` offered by Kokkos or compiler auto-vectorization. However, this

Algorithm 1 Pairwise distances

```
Input: Library time series $\mathbf{x} \in \mathbb{R}^L$
Output: Pairwise distance matrix $\mathbf{D} \in \mathbb{R}^{L \times L}$
// TeamPolicy
1  parallel for $i \leftarrow 1$ to $L$ do
       // TeamThreadRange
2      parallel for $j \leftarrow 1$ to $L$ do
3          $\mathbf{D}_{i,j} \leftarrow 0$
4          for $k \leftarrow 1$ to $E$ do
5              $\mathbf{D}_{i,j} \leftarrow \mathbf{D}_{i,j} + (\mathbf{x}_{i-k\tau} - \mathbf{x}_{j-k\tau})^2$
6          end
7      end
8  end
```

is not profitable since the embedding dimension is usually small ($E \leq 20$) in realistic use cases. We thus use Kokkos SIMD types [25] to explicitly vectorize the j-loop. SIMD types are an abstraction around native SIMD instructions (*e.g.*, AVX-256, AVX-512 and SVE) and allow hardware-independent explicit vectorization.

Algorithm 2 show the pseudocode of the top-k search kernel that runs on the distance matrix computed by the pairwise distance kernel. The top-k search kernel is particularly challenging to implement in a performance-portable manner because state-of-the-art top-k search algorithms [26], [27] are usually optimized for a specific hardware. Thus, we design and implement a top-k search algorithm that works on both CPU and GPU efficiently. The basic idea of this algorithm is to maintain a list of top-k items while scanning the input. To hold the top-k items, we use a sorted list. The average time complexity of inserting a new element into a sorted list $O(k)$. Although there are data structures with lower time complexity such as a max-heap with $O(\log k)$ average time complexity for insertion, we found out that a sorted list is the fastest for a small k (≤ 20) required in EDM.

In our algorithm, each thread team finds the top-k elements from one row of the distance matrix (lines 1–10 in Algorithm 2). Each thread within a thread team maintains a local sorted list on team-private scratch memory that holds the top-k elements it has seen so far. Threads read the distance matrix in a coalesced manner and push the distances and indices to their local lists (lines 2–4). Once all elements are processed, one leader thread in each thread team merges the local lists within the team and writes the final top-k elements to global memory (lines 5–9). Once the top-k elements are found, $\mathbf{D}$ is converted from squared distance to Euclidean distance by taking the square root. $\mathbf{D}$ is then converted to exponential scale and normalized such that the sum of each row equals to one according to Eqs. (2) and (3).

D. CCM

When performing pairwise CCM between a large number of time series, the number of k-NN search queries can be drastically reduced by performing multiple CCMs that use

Algorithm 2 Top-k search

```
Input: Pairwise distance matrix $\mathbf{D} \in \mathbb{R}^{L \times L}$
Output: Top-k distance matrix $\mathbf{D} \in \mathbb{R}^{L \times (E+1)}$ and index
        matrix $\mathbf{I} \in \mathbb{N}^{L \times (E+1)}$
// TeamPolicy
1  parallel for $i \leftarrow 1$ to $L$ do
       // TeamThreadRange
2      parallel for $j \leftarrow 1$ to $L$ do
3          Insert $(\mathbf{D}_{i,j}, j)$ into local list
4      end
       // Single
5      for $j \leftarrow 1$ to E+1 do
6          for $j \leftarrow 1$ to # of threads in the team do
7              $(\mathbf{D}_{i,j}, \mathbf{I}_{i,j}) \leftarrow$ Pop element from thread $j$'s list
8          end
9      end
10     Normalize $\mathbf{D}$
11 end
```

the same library time series in parallel [8]. This is because in Eq. (2), $\mathbf{w}$ and $\mathbf{n}$ only depend on the library time series and do not depend on the predicted time series. Thus, $\mathbf{w}$ and $\mathbf{n}$ can be precomputed and reused for many predicted time series. If $\mathbf{w}$ and $\mathbf{n}$ are reused, the k-NN search becomes no longer a bottleneck, and the lookup of the predicted time series becomes the primary bottleneck.

Algorithm 3 shows the pseudocode of the lookup kernel. The basic idea of the lookup kernel is to perform the lookups of many predicted time series in batch. This reduces the number of k-NN search, but also increases the chance that the distances and indices of the neighbors are kept cached. The outer i-loop (lines 1–8 in Algorithm 3) iterates over the predicted time series, and assigned to thread teams. The inner j-loop (lines 2–7) loop iterates over the time steps, and assigned to threads within the team. The body of the j-loop

Algorithm 3 Lookup

```
Input: Predicted time series $\mathbf{Y} \in \mathbb{R}^{L \times N}$, normalized
       top-k distance matrix $\mathbf{D} \in \mathbb{R}^{L \times (E+1)}$ and index
       matrix $\mathbf{I} \in \mathbb{R}^{L \times (E+1)}$ computed from the library
       time series
Output: Predicted time series $\hat{\mathbf{Y}} \in \mathbb{R}^{L \times N}$
// TeamPolicy
1  parallel for $i \leftarrow 1$ to $N$ do
       // TeamThreadRange
2      parallel for $j \leftarrow 1$ to $L$ do
3          $\hat{\mathbf{y}}_{j,i} \leftarrow 0$
4          for $k \leftarrow 1$ to $E + 1$ do
5              $\hat{\mathbf{y}}_{j,i} \leftarrow \hat{\mathbf{y}}_{j,i} + \mathbf{D}_{j,k} \cdot \mathbf{y}_{\mathbf{I}_{j,k} - k\tau}$
6          end
7      end
8  end
```

Algorithm 4 S-map

```
   Input: Library time series $\mathbf{x} \in \mathbb{R}^L$ and predicted time
          series $\mathbf{y} \in \mathbb{R}^L$
   Output: Pairwise distance matrix $\mathbf{D} \in \mathbb{R}^{L \times L}$
   // TeamPolicy
1  parallel for $i \leftarrow 1$ to $L$ do
       // TeamThreadRange
2      parallel for $j \leftarrow 1$ to $L$ do
3          $\mathbf{D}_{i,j} \leftarrow 0$
4          for $k \leftarrow 1$ to $E$ do
5              $\mathbf{D}_{i,j} \leftarrow \mathbf{D}_{i,j} + (\mathbf{x}_{i-k\tau} - \mathbf{y}_{j-k\tau})^2$
6          end
7          $\mathbf{D}_{i,j} \leftarrow \sqrt{\mathbf{D}_{i,j}}$
8      end
       // TeamThreadRange
9      parallel for $j \leftarrow 1$ to $L$ do
10         $w \leftarrow \exp\left\{-\frac{\theta \mathbf{D}_{j,i}}{\bar{\mathbf{D}}_{j,i}}\right\}$
11         $\mathbf{A}_{j,0,i} \leftarrow w$
12         for $k \leftarrow 1$ to $E$ do
13             $\mathbf{A}_{j,k,i} \leftarrow w \cdot \mathbf{x}_{j-k\tau}$
14         end
15         $\mathbf{b}_{j,i} \leftarrow w \cdot \mathbf{x}_{j+T_p}$
16     end
17 end
18 Batch solve $\operatorname{argmin} \|\mathbf{Ax} - \mathbf{b}\|_2$ and store solution to $\mathbf{c}$
   // RangePolicy
19 parallel for $i \leftarrow 1$ to $L$ do
20     $\mathbf{y}_i \leftarrow \mathbf{c}_{0,i}$
21     for $k \leftarrow 1$ to $E$ do
22         $\hat{\mathbf{y}}_{i+T_p} \leftarrow \mathbf{c}_{k,i} \cdot \mathbf{y}_{i-k\tau}$
23     end
24 end
```

is essentially identical to Eq. (2). Since the lookup requires indirect (random) accesses to the predicted time series $\mathbf{y}$, we cache the time series onto team-private scratch memory.

E. S-MAP

As described in Section II-A2, S-map builds a linear model for each predicted point and estimates its parameters using the least-squares method. Thus, the main bottleneck of S-map is to solve a large number of least-squares problems. In kEDM, we use `sgels`, a function provided by LAPACK[6] that finds solutions to least-squares problems. To alleviate the kernel launch overhead and increase the utilization of the hardware, we use the `cublasSgelsBatched` function provided by cuBLAS,[7] which solves many least-squares problems in parallel. On the CPU, we simply call `sgels` repeatedly since standard LAPACK implementations do not provide a batched version of `sgels`.

[6]https://netlib.org/lapack/
[7]https://docs.nvidia.com/cuda/cublas/

Listing 4 shows the pseudocode of S-map. Lines 1 to 17 prepare $\mathbf{A}$ and $\mathbf{b}$ according to Eqs. (6)–(8). Note that $\mathbf{A}$ and $\mathbf{b}$ are three dimensional arrays where their first dimension corresponds to batches. Since the size of $\mathbf{A}$ is $O(L^2E)$, $\mathbf{A}$ does not fit in memory if L or E is large. We thus calculate the size of $\mathbf{A}$ and control the batch size so that $\mathbf{A}$ fits in memory. Line 18 invokes the least-squares solver, and lines 19–24 make predictions using the estimated linear models.

F. APPROXIMATE EDM FOR LARGE-SCALE DATASETS

Even if multi-threading or GPU offloading is used, the runtime of EDM becomes prohibitive for long time series since the time complexity of k-NN search is $O(L^2)$. To solve this problem, we take advantage of *Approximate Nearest Neighbor (ANN) search* [28] algorithms to accelerate the costly k-NN search in EDM. ANN search algorithms provide a large speedup compared to an exhaustive k-NN search with a small loss in accuracy. In this article, we consider the following three popular ANN search algorithms:

- *Inverted File Index (IVF)*: IVF [29] is a simple index that clusters the training points using the k-means method. At search time, only the points within the cluster nearest from the query point are searched.
- *k-dimensional Tree (k-d Tree)*: k-d Tree [30] is a classical tree-based index that hierarchically partitions the space into cells. It is constructed by recursively splitting the training points by a hyperplane perpendicular to an axis. At search time, the tree is traversed and only a small number of cells are searched to find the neighbors.
- *Hierarchical Navigable Small World (HNSW)* [31]: HNSW is a recently proposed graph-based index based on Navigable Small World (NSW) graphs [32], [33]. An NSW graph is a graph where a greedy search algorithm can find a path between two vertices with a number of hops polylogarithmic to the number of vertices. This property can be leveraged for k-NN search by building an NSW graph where vertices correspond to data points and searching over this graph.

We use the Faiss [26] library to implement IVF and HNSW, and use the nanoflann [34] library to implement k-d Tree, since these libraries are used in many previous studies and are generally considered well-optimized and mature.

IV. EVALUATION

In this evaluation, we first assess the performance of k-NN search and lookup kernels in kEDM using microbenchmarks, and analyze the performance benefit of tuning. We then evaluate the performance of CCM and S-map using kEDM. Finally, we analyze the runtime and accuracy trade-off of Simplex projection accelerated with approximate k-NN search.

The evaluation experiments are conducted on both CPU and GPU. The evaluation environment is a computing server equipped with two sockets of AMD EPYC 7742 CPUs, 1 TB of DDR4 SDRAM and one NVIDIA A100 PCIe card

(40 GB model). We use the AMD Optimizing C/C++ Compiler (AOCC) 4.0.0 and NVIDIA CUDA Toolkit 11.8 to compile kEDM for CPU and GPU, respectively. As for the BLAS/LAPACK implementation, we use AMD Optimizing CPU Libraries (AOCL) BLIS and libFLAME on the CPU, and cuBLAS on the GPU. All experiments on the CPU are conducted on a single socket of EPYC 7742 with core binding enabled.

A. MICROBENCHMARKS

Fig. 2 shows the runtime of the k-NN search of kEDM and mpEDM on A100 as a function of the embedding dimension E. When $L = 10^3$, the pairwise distances kernel of kEDM is significantly faster than that of mpEDM, and the speedup reaches up to 26.7×. In mpEDM, the time-delayed embeddings were performed on the CPU, and the embedded vectors were passed to ArrayFire's kNN search function. Contrastingly, kEDM performs the time-delayed embedding during the distance calculation and thus requires much fewer memory accesses. On the other hand, the partial sort kernel is slower than mpEDM. This is because the overhead of maintaining the priority queues becomes dominant if the total number of time steps is small, and each thread only processes a small number of elements. However, since the distance calculation is the bottleneck in mpEDM, the k-NN search is up to 6.32× faster on kEDM.

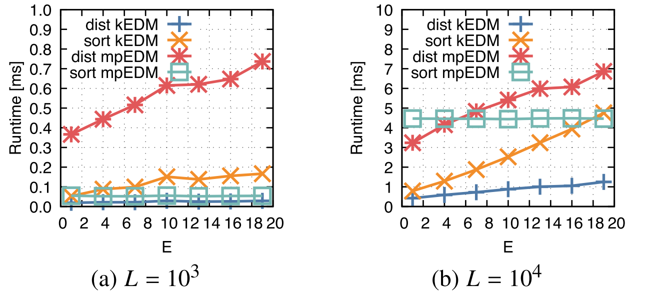


FIGURE 2. k-NN search runtime on A100.

When $L = 10^4$, the pairwise distance kernel is up to 7.62× faster than mpEDM. The partial sort kernel is 5.67× faster when $E = 1$, but the speedup becomes smaller as E increases. This is because a larger E requires a more shared memory to hold the local priority queues, and results in lower multiprocessor occupancy. Fig. 3 shows the runtime of k-NN search on EPYC. Overall, kEDM achieves comparable performance as mpEDM on EPYC. This suggests that the overhead imposed by Kokkos is minimal, and our performance-portable implementation can attain similar performance as a conventional implementation for CPU.

Fig. 4 shows the runtime of the lookup kernel (without cross-correlation calculation) on A100 and EPYC 7742. We generate 10^5 synthetic target time series each having 10^3 or 10^4 time steps, and measure the time to perform batch lookups from a library time series with the same number

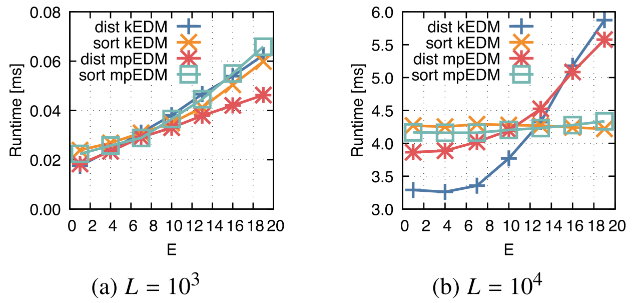


FIGURE 3. k-NN search runtime on EPYC 7742.

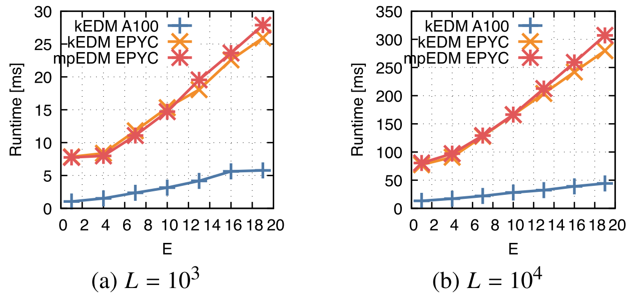


FIGURE 4. Lookup runtime on A100 and EPYC 7742.

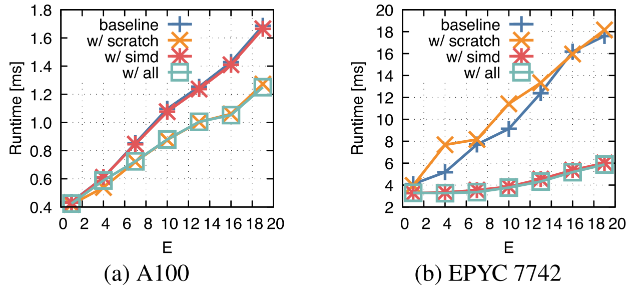


FIGURE 5. Impact of performance tuning on the pairwise distances kernel.

of time steps. kEDM on A100 is consistently faster than on EPYC, and the speedup ranges from 3.89× to 7.50×.

To assess the impact of performance tuning described in Section III-C, we compare the performance of kernels with and without the performance tuning applied. Fig. 5 shows the performance of the pairwise distances kernel with and without using the scratch memory and SIMD types. On A100, using the SIMD types yield no measurable change in performance as expected. This is because the SIMD types are translated to scalar types on GPUs. Using scratch memory clearly improves the performance of the kernel by up to 1.34×. On EPYC, scratch memory has no performance benefit, but SIMD types yield significant performance improvement. The speedup reaches 3.0× at maximum.

Fig. 6 shows the performance of the lookup kernel with and without using the scratch memory. The performance benefit of using scratch memory is drastic on A100, reaching up to a 17.2× speedup. Interestingly, scratch memory slightly improves the performance on EPYC as well, even though

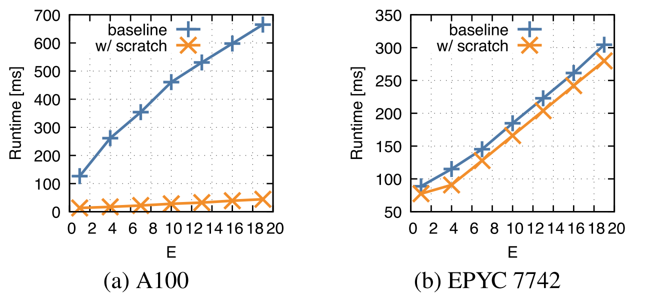


FIGURE 6. **Impact of performance tuning on the lookup kernel.**

CPUs do not have a manually managed cache. This might be because accessing the predicted time series in advance loads the time series into the cache hierarchy. These results demonstrate that the Kokkos programming model allows optimization for a specific hardware while keeping the performance penalty on other hardware minimal.

B. CCM

We prepare the following six real-world datasets with diverse number and length of time series that reflect a variety of use cases, and measure the runtime of kEDM for completing pairwise CCM calculations. At the time of the development of mpEDM and kEDM the target datasets for analysis were mostly biological. For this reason, we used whole genome RNA sequencing (transcriptome) datasets that we generated ourselves, as well as neuroscience whole brain Calcium imaging, datasets that represent the main areas of experimental biological big data at this time. All datasets are recorded in single-precision floating point numbers. The number of time series and time steps in each datasets are described in Table 1.

- *Fish1_Normo* is a subset of 154 representative neuron behaviors of the so called default state behavior of zebrafish larvae collected by lightsheet microscopy of fish transgenic with nuclear-localized GCAMP6f, a calcium indicator that we generated ourselves (unpublished).
- *Fly80XY* is a *Drosophila melanogaster* (fruit fly) whole brain lightfield microscopy GCAMP6f recording, where distinct brain areas were identified by independent component analysis with the fly left right and forward walking speed behaviors collected on a styrofoam ball [35].
- *Genes_MEF* contains the gene expression profiles of all genes and small RNAs generated by Illumina[8] short read sequencing from mouse embryo fibroblast genes over 96 time steps of two cycles of serum induction and starvation stimulation generated by the authors (unpublished).
- *Subject6* and *Subject11* are whole brain light sheet microscopy GCAMP6f recordings at whole brain scale and single neuron resolution of larval zebrafish responding to visual stimuli [36].
- *F1* is a subset of whole brain 2-photon microscopy a larval zebrafish where seizures were biochemically induced with recordings of three phases: control conditions, pre-seizure and full seizure [37].

[8]https://www.illumina.com/

Table 1 shows the runtime for performing a pairwise CCM on each dataset. Fig. 7 shows the speedup of each implementation on each platform over mpEDM executed on AMD EYPC 7742. The results clearly demonstrate that kEDM outperforms mpEDM in most cases. In particular, kEDM shows significantly higher (up to 6.58×) performance than mpEDM on A100. This performance gain is obtained from the optimized k-NN search and GPU-enabled lookups. Datasets with few number of time steps or time series, *i.e.*, Fish1_Normo and Genes_MEF, do not benefit from GPU acceleration on both kEDM and mpEDM. This is because the runtime of each kernel is short, and the reduction of kernel runtime cannot offset the runtime overhead of offloading computation to the GPU.

C. S-MAP

Fig. 8 shows the runtime of S-map using kEDM. If $L = 10^3$, A100 is slower than EPYC. This is because the computation is not large enough to fully utilize the GPU. The GPU speedup becomes larger as the number of time steps and embedding dimension become larger. The GPU speedup becomes 12.8× when $L = 10^4$ and $E = 20$.

D. APPROXIMATE EDM

In this evaluation, we assess the trade-off between runtime and accuracy when using ANN search in EDM. We create artificial time series data with varying lengths by simulating the Lorenz attractor using the Runge-Kutta method [38], and compare the runtime and accuracy of Simplex projection across different ANN algorithms. We also provide the performance when using exact k-NN search as a baseline. Each ANN search algorithm has different hyperparameters that affect its search speed and accuracy. We use the default values provided by Faiss and nanoflann in all algorithm to make a fair comparison.

1) RUNTIME

Since the dimensionality of data heavily impacts the performance of ANN search [39], we measure the runtime of approximate EDM in two cases, where the embedding dimension is 1 and 20. The measurement results are summarized in Tables 2 and 3 in the Appendix.

Fig. 9 shows the runtime of Simplex projection with respect to the time series length when the embedding dimension is one. The time series length L is varied from 2^{10} (=1,024) to 2^{20} (=1,048,576). On the CPU, exact search is the fastest only if $L = 2^{10}$, and quickly becomes the slowest due to its quadratic time complexity. k-d Tree is the fastest algorithm for most time series lengths. When $L = 2^{20}$, k-d Tree attains

TABLE 1. Runtime for performing pairwise CCM on real-world datasets.

			NVIDIA A100			AMD EPYC 7742		
Dataset	# of Time Series	# of Time Steps	kEDM	mpEDM	Speedup	kEDM	mpEDM	Speedup
Fish1_Normo	154	1,600	6s	11s	2.67×	3s	4s	1.33×
Fly80XY	82	10,608	10s	35s	3.56×	22s	30s	1.36×
Genes_MEF	45,318	96	231s	236s	1.02×	96s	139s	1.45×
Subject6	92,538	3,780	3,727s	24,525s	6.58×	12,145s	11,571s	0.95×
Subject11	101,729	8,528	11,125s	56,814s	5.10×	43,812s	38,542s	0.88×
F1	8,520	29,484	5,592s	18,001s	3.21×	23,001s	19,950s	0.87×

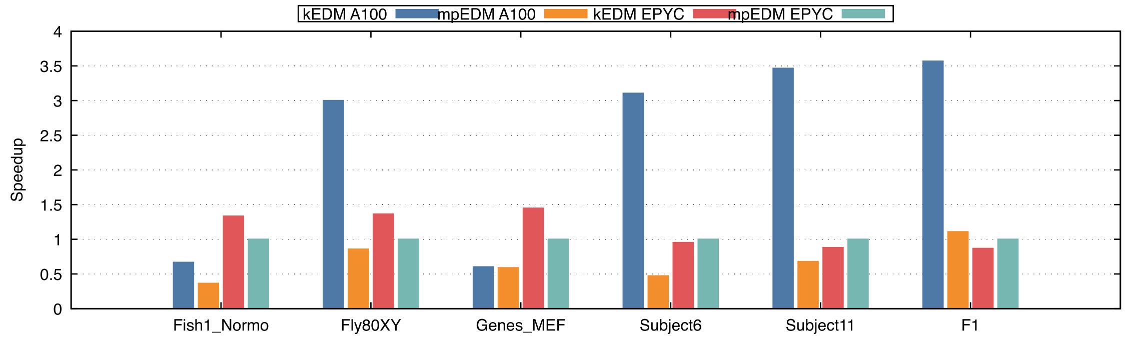
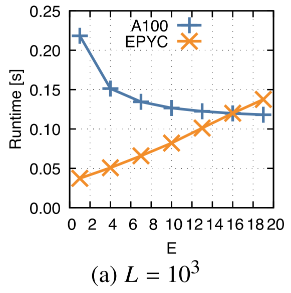


FIGURE 7. Speedup for performing pairwise CCM on real-world datasets.



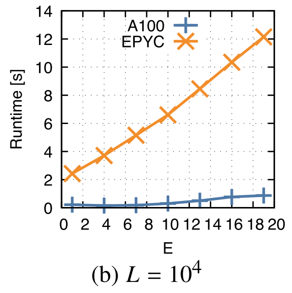


FIGURE 8. S-map runtime.

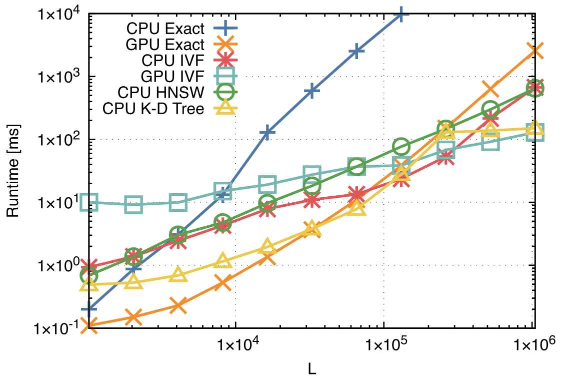


FIGURE 9. Runtime of approximate Simplex projection ($E = 1$).

a 3,406× speedup over exact search. It is also 4.4× and 4.3× faster than IVF and HNSW, respectively. This indicates that k-d Tree is highly efficient for low-dimensional data.

On the GPU, exact search is faster than IVF when the time series is short. IVF becomes faster than exact search when $L \geq 2^{18}$, and attains 19.9× speedup over exact search when $L = 2^{20}$. IVF clusters the data points using the k-means algorithm and then performs an exact search within the cluster closest to the query point. This reduces the number of points to search in the exact search, but the time complexity remains quadratic. The overhead for this clustering step does not pay off until the time series becomes long.

Fig. 10 shows the runtime of Simplex projection when the embedding dimension is 20. On the CPU, k-d Tree is no longer the fastest one. This is because k-d Tree struggles with high-dimensional data because of the *curse of dimensionality* [40], where the number of visited cells increases exponentially with respect to the data dimension. IVF and HNSW both achieve similar performance, but HNSW is the fastest when the time series becomes long ($L \geq 2^{19}$). The speedups of HNSW and IVF over exact search are 775.1× and 325.6×, respectively, when $L = 2^{20}$. On the GPU, the trend is similar to the low-dimensional case. IVF is initially slower than exact search, but surpasses exact search when $L \geq 2^{17}$.

These results clearly show that there is no single ANN search algorithm that always performs faster than all the other algorithms. Therefore, the ANN search algorithm should be switched depending on the characteristics of the input dataset. A reasonable choice would be to use k-d Tree for lower embedding dimensions and HNSW for higher embedding dimensions on a CPU. On a GPU, exact search should be used for shorter time series and IVF for longer time series. In the next section, we will compare the ANN search algorithms from the aspect of accuracy.

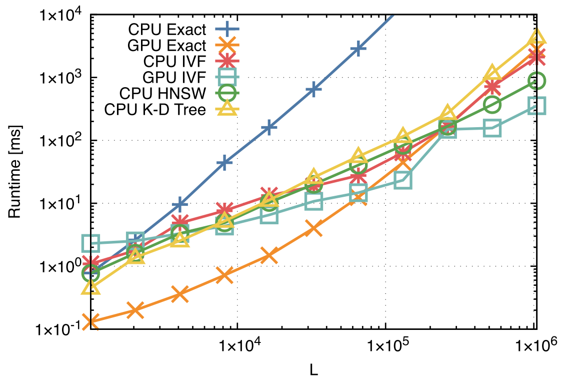


FIGURE 10. Runtime of approximate Simplex projection ($E = 20$).

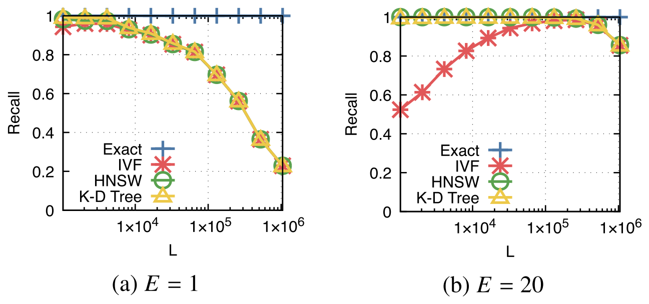


FIGURE 11. Recall of nearest neighbors (higher is better).

2) ACCURACY

We now assess the accuracy of ANN search and Simplex projection. Fig. 11 shows the recall of ANN search, which is defined as the ratio of found neighbors by the ANN search algorithm to the number of true neighbors. The true nearest neighbors are obtained using exact nearest neighbor search on the GPU. When $E = 1$, the recall rate drops quickly as L increases. Interestingly, the difference in recall among the three ANN algorithms is very small. When $E = 20$, the decline in recall of HNSW and k-d Tree is small compared to the $E = 1$ case. This is because more neighbors are found compared to the $E = 1$ case, and the probability that the true neighbors are included in the found neighbors becomes higher. The recall of IVF is initially low, but increases as the time series become longer. As described in Section III-F, IVF clusters the training points, and then searches only within the cluster that is closest from the query point. Therefore, if the number of points per cluster is few, the probability that the closest cluster does not contain the true neighbors becomes high, and thus the recall drops.

To evaluate the impact of errors incurred by approximate k-NN search on the prediction accuracy of Simplex projection, we measure the Mean Absolute Percentage Error (MAPE) of Simplex projection and compare it with the conventional Simplex projection using exact k-NN search. Fig. 12 shows the relative MAPE of Simplex projection when using each ANN search algorithm instead of exact k-NN search.

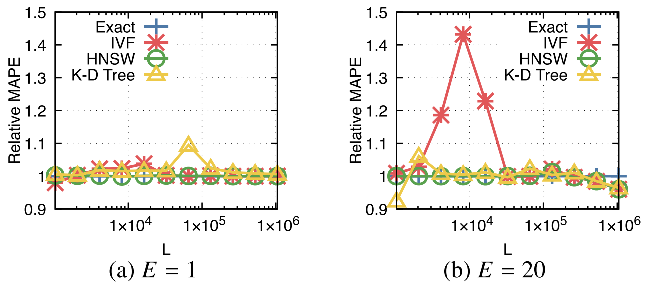


FIGURE 12. Normalized MAPE of approximate Simplex projection (lower is better).

Interestingly, the increase in MAPE is minimal even if the recall of k-NN search clearly drops such as in the $E = 1$ case. This can be explained as follows. If the embedded points are densely distributed on the attractor manifold, there are a large number of points surrounding the predicted point. Since the key idea of Simplex is to use an ensemble of points similar to the predicted point in the state space, if the ANN search can find some of the similar points around the predicted point, Simplex projection can still provide robust accuracy if those points are not exact k-nearest neighbors.

Out of the ANN search algorithms, HNSW is the most stable algorithm in terms of MAPE. The increase in MAPE compared to conventional Simplex projection is consistently below 1% for the variety of time series lengths and embedding dimensions we evaluated. Considering the comparison of runtime in the previous section, HNSW takes a good balance between speed and accuracy and is the best choice on the CPU.

V. LIMITATIONS AND FUTURE WORK

Our evaluation in this article is limited to two processors: NVIDIA A100 GPU and AMD EPYC 7742 CPU. Even though kEDM is developed using the Kokkos performance portable-programming model and thus expected to achieve reasonable performance on other processors as well, the exact speedup is yet to be investigated. Performance evaluation on other recent high-performance processors such as AMD Instinct GPUs [41] or NEC Vector Engines [42] is a future work.

The evaluation results indicate that the choice of the best ANN search algorithm depends on several parameters such as the number of time steps and embedding dimension. In addition, it also depends on whether a GPU is available on the system, and the performance of the CPU and GPU. Currently, the ANN search algorithm needs to be manually selected by the user by considering these factors, which might be a burden for some users. We thus plan to develop an automatic algorithm selection mechanism in the future that chooses the fastest ANN search algorithm on a target hardware for a given dataset.

In terms of the implementation, not all features available in cppEDM are implemented in kEDM. These features include exclusion radius (excludes nearest neighbors if their relative time index is within a threshold), conditional embedding

TABLE 2. Runtime of approximate Simplex projection ($E = 1$).

L	CPU Exact	GPU Exact	CPU IVF	GPU IVF	HNSW	k-d Tree
1024	0.20	0.11	0.93	10.02	0.69	0.49
2048	0.87	0.15	1.38	9.16	1.37	0.53
4096	3.08	0.23	2.45	9.95	3.04	0.69
8192	13.21	0.53	4.29	15.09	4.77	1.15
16384	128.79	1.36	7.85	19.11	9.69	1.95
32768	593.39	3.67	10.98	27.51	18.30	3.79
65536	2529.81	11.00	13.42	36.89	36.88	7.80
131072	9720.97	38.31	23.67	38.49	76.80	27.68
262144	44034.59	150.72	53.32	67.95	146.56	129.67
524288	161425.80	633.88	215.56	91.73	299.35	137.95
1048576	512171.50	2571.73	671.09	129.14	647.90	150.35

TABLE 3. Runtime of approximate Simplex projection ($E = 20$).

L	CPU Exact	GPU Exact	CPU IVF	GPU IVF	HNSW	k-d Tree
1024	0.78	0.13	1.09	2.30	0.77	0.45
2048	2.61	0.20	1.78	2.52	1.61	1.38
4096	9.56	0.36	4.86	3.31	3.32	2.54
8192	44.22	0.72	7.61	4.35	4.89	5.22
16384	160.50	1.49	13.28	6.53	10.30	11.21
32768	648.15	4.03	18.69	10.75	19.99	25.86
65536	2878.60	12.48	27.65	14.73	40.48	55.65
131072	13960.90	44.69	64.19	23.28	83.20	115.78
262144	57941.15	172.83	166.24	149.51	167.44	269.19
524288	208946.20	695.92	715.70	156.84	369.37	1169.87
1048576	686427.30	2802.78	2107.57	355.07	885.56	4405.45

(selectively embeds library points), pluggable solvers (allows swapping the linear model used in S-map to other models such as Ridge, LASSO and ElasticNet regression models), and plotting of results. However, there are no fundamental challenges in implementing these features in kEDM.

VI. CONCLUSION

We presented the design and implementation of kEDM, a high-performance implementation of EDM for analyzing large-scale time series datasets on CPU and GPU. kEDM achieved up to 6.58× speedup in pairwise causal inference of real-world biology datasets compared to mpEDM. This result demonstrates the effectiveness of our approach, *i.e.,* developing the software using a performance-portable programming model and applying hardware-specific optimizations to performance-critical kernels.

Furthermore, we integrated approximate k-NN search into EDM to enable the analysis of extremely large datasets. Simplex projection enhanced with approximate k-NN search was 790× faster Simplex projection using exact k-NN search with less than 1% increase in MAPE. A key insight we obtained is that Simplex projection is surprisingly robust to low k-NN search recall rate. This property might apply to other EDM algorithms as well, and could be further exploited in the future to achieve significant speedup.

The massive speedup delivered by kEDM will allow the research community to apply EDM to previously intractable big data. Conventionally, the number of time steps or time series had to be significantly reduced by subsampling or dimensionality reduction to analyze large-scale datasets using EDM within feasible time. However, this inherently causes loss of information and produces potentially inaccurate results. We believe that kEDM will widen the applicability of EDM to various research fields involving the analysis of non-linear time series data.

APPENDIX
APPROXIMATE EDM RUNTIME

Table 2 compares the runtime of approximate Simplex projection using different ANN search algorithms and varying the number of time steps when $E = 1$. Similarly, Table 3 compares the runtime of approximate Simplex projection when $E = 20$.

ACKNOWLEDGMENT

The authors would like to thank Dominic R. W. Burrows at the MRC Centre for Neurodevelopmental Disorders, Institute of Psychiatry, Psychology and Neuroscience, King's College London, London, U.K., for providing the F1 dataset used in the performance evaluation. They also would like to thank Jorge Aldana at the Salk Institute for Biological Studies for providing assistance with the computing servers.

REFERENCES

[1] C.-W. Chang, M. Ushio, and C.-H. Hsieh, "Empirical dynamic modeling for beginners," *Ecol. Res.*, vol. 32, no. 6, pp. 785–796, Nov. 2017.

[2] S. Liu, M. Ye, G. M. Pao, S. M. Song, J. Jhang, H. Jiang, J.-H. Kim, S. J. Kang, D.-I. Kim, and S. Han, "Divergent brainstem opioidergic pathways that coordinate breathing with pain and emotions," *Neuron*, vol. 110, no. 5, pp. 857–873, Mar. 2022.

[3] M. Ushio, C.-H. Hsieh, R. Masuda, E. R. Deyle, H. Ye, C.-W. Chang, G. Sugihara, and M. Kondoh, "Fluctuating interaction network and time-varying stability of a natural fish community," *Nature*, vol. 554, no. 7692, pp. 360–363, Feb. 2018.

[4] S. Liu, D.-I. Kim, T. G. Oh, G. M. Pao, J.-H. Kim, R. D. Palmiter, M. R. Banghart, K.-F. Lee, R. M. Evans, and S. Han, "Neural basis of opioid-induced respiratory depression and its rescue," *Proc. Nat. Acad. Sci.*, vol. 118, no. 23, Jun. 2021, Art. no. e2022134118.

[5] J. Park, G. M. Pao, G. Sugihara, E. Stabenau, and T. Lorimer, "Empirical mode modeling: A data-driven approach to recover and forecast nonlinear dynamics from noisy data," *Nonlinear Dyn.*, vol. 108, no. 3, pp. 2147–2160, May 2022.

[6] H. Ma, K. Aihara, and L. Chen, "Detecting causality from nonlinear dynamics with short-term time series," *Sci. Rep.*, vol. 4, no. 1, pp. 1–10, Dec. 2014.

[7] B. Pu, "Exploiting multiple levels of parallelism of convergent cross mapping," Ph.D. dissertation, Dept. Comput. Sci., Univ. Saskatchewan, Saskatoon, SK, Canada, 2019.

[8] W. Watanakeesuntorn, K. Takahashi, K. Ichikawa, J. Park, G. Sugihara, R. Takano, J. Haga, and G. M. Pao, "Massively parallel causal inference of whole brain dynamics at single neuron resolution," in *Proc. IEEE 26th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2020, pp. 196–205.

[9] K. Takahashi, W. Watanakeesuntorn, K. Ichikawa, J. Park, R. Takano, J. Haga, G. Sugihara, and G. M. Pao, "kEDM: A performance-portable implementation of empirical dynamic modeling using Kokkos," in *Proc. Pract. Exp. Adv. Res. Comput.*, Jul. 2021, pp. 1–8.

[10] F. Takens, "Detecting strange attractors in turbulence," in *Dynamical Systems and Turbulence, Warwick 1980*, vol. 898. Berlin, Germany: Springer, 1981, pp. 366–381.

[11] E. R. Deyle and G. Sugihara, "Generalized theorems for nonlinear state space reconstruction," *PLoS ONE*, vol. 6, no. 3, Mar. 2011, Art. no. e18295.

[12] G. Sugihara and R. M. May, "Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series," *Nature*, vol. 344, no. 6268, pp. 734–741, Apr. 1990.

[13] G. Sugihara, "Nonlinear forecasting for the classification of natural time series," *Phil. Trans. Roy. Soc. London A, Phys. Eng. Sci.*, vol. 348, no. 1688, pp. 477–495, 1994.

[14] G. Sugihara, R. May, H. Ye, C.-H. Hsieh, E. Deyle, M. Fogarty, and S. Munch, "Detecting causality in complex ecosystems," *Science*, vol. 338, no. 6106, pp. 496–500, Oct. 2012.

[15] J. Malcolm, P. Yalamanchili, C. McClanahan, V. Venugopalakrishnan, K. Patel, and J. Melonakos, "ArrayFire: A GPU acceleration platform," *Proc. SPIE*, vol. 8403, May 2012, Art. no. 84030A.

[16] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 805–817, Apr. 2022.

[17] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, Dec. 2014.

[18] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Assessing the performance portability of modern parallel programming models using TeaLeaf," *Concurrency Comput.*, vol. 29, no. 15, pp. 1–15, 2017.

[19] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance portability across diverse computer architectures," in *Proc. IEEE/ACM Int. Workshop Perform., Portability Productiv. HPC (P3HPC)*, Nov. 2019, pp. 1–13.

[20] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking performance portability on the yellow brick road to exascale," in *Proc. IEEE/ACM Int. Workshop Perform., Portability Productiv. HPC (P3HPC)*, Nov. 2020, pp. 1–13.

[21] M. A. Sprague, S. Ananthan, G. Vijayakumar, and M. Robinson, "ExaWind: A multifidelity modeling and simulation environment for wind energy," *J. Phys., Conf. Ser.*, vol. 1452, no. 1, Jan. 2020, Art. no. 012071.

[22] I. Demeshko, J. Watkins, I. K. Tezaur, O. Guba, W. F. Spotz, A. G. Salinger, R. P. Pawlowski, and M. A. Heroux, "Toward performance portability of the Albany finite element analysis code using the Kokkos library," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 2, pp. 332–352, Mar. 2019.

[23] J. K. Holmen, A. Humphrey, D. Sunderland, and M. Berzins, "Improving Uintah's scalability through the use of portable Kokkos-based data parallel tasks," in *Proc. Pract. Exp. Adv. Res. Comput. Sustainability, Success Impact*, Jul. 2017, pp. 1–8.

[24] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. I. Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS—A flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comput. Phys. Commun.*, vol. 271, Feb. 2022, Art. no. 108171.

[25] D. Sahasrabudhe, E. T. Phipps, S. Rajamanickam, and M. Berzins, "A portable SIMD primitive using Kokkos for heterogeneous architectures," in *Proc. Int. Workshop Accel. Program. Using Directives*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 12017, 2020, pp. 140–163.

[26] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Trans. Big Data*, vol. 7, no. 3, pp. 535–547, Jul. 2021.

[27] A. Shanbhag, H. Pirk, and S. Madden, "Efficient top-K query processing on massively parallel hardware," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 1557–1570.

[28] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data—Experiments, analyses, and improvement," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 8, pp. 1475–1488, Aug. 2019.

[29] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 1, pp. 117–128, Jan. 2011.

[30] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[31] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, Apr. 2020.

[32] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces," in *Proc. 5th Int. Conf. Similarity Search Appl. (SIASP)*, 2012, pp. 132–147.

[33] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Inf. Syst.*, vol. 45, pp. 61–68, Sep. 2014.

[34] J. L. Blanco and P. K. Rai. (2014). *Nanoflann: A C++ Header-Only Fork of FLANN, a Library for Nearest Neighbor (NN) With KD-Trees*. [Online]. Available: https://github.com/jlblancoc/nanoflann

[35] S. Aimon, T. Katsuki, T. Jia, L. Grosenick, M. Broxton, K. Deisseroth, T. J. Sejnowski, and R. J. Greenspan, "Fast near-whole–brain imaging in adult Drosophila during responses to stimuli and behavior," *PLOS Biol.*, vol. 17, no. 2, pp. 1–31, Feb. 2019.

[36] X. Chen, Y. Mu, Y. Hu, A. T. Kuan, M. Nikitchenko, O. Randlett, A. B. Chen, J. P. Gavornik, H. Sompolinsky, F. Engert, and M. B. Ahrens, "Brain-wide organization of neuronal activity and convergent sensorimotor transformations in larval zebrafish," *Neuron*, vol. 100, no. 4, pp. 876–890, Nov. 2018.

[37] D. Burrows, G. Diana, B. Pimpel, F. Moeller, M. Richardson, D. Bassett, M. Meyer, and R. Rosch, "Single-cell networks reorganise to facilitate whole-brain supercritical dynamics during epileptic seizures," *bioRxiv*, Oct. 2021. [Online]. Available: https://www.biorxiv.org/content/early/2021/10/16/2021.10.14.464473

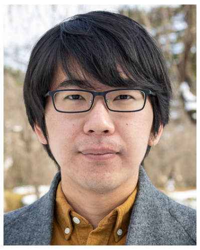
[38] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge, U.K.: Cambridge Univ. Press, 2007.

[39] M. Aumüller, E. Bernhardsson, and A. Faithfull, "ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Inf. Syst.*, vol. 87, Jan. 2020, Art. no. 101374.

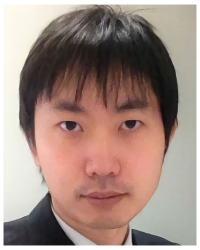
[40] P. Ram and K. Sinha, "Revisiting kd-tree for nearest neighbor search," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 1378–1388.

[41] A. Smith and N. James, "AMD Instinct^{TM} MI200 series accelerator and node architectures," in *Proc. IEEE Hot Chips 34 Symp. (HCS)*. Washington, DC, USA: IEEE Computer Society, Aug. 2022, pp. 1–23.

[42] K. Takahashi, S. Fujimoto, S. Nagase, Y. Isobe, Y. Shimomura, R. Egawa, and H. Takizawa, "Performance evaluation of a next-generation SX-Aurora TSUBASA vector supercomputer," in *Proc. 38th Int. Conf. High Perform. Comput. (ISC High Performance)*, 2023, pp. 359–378.



KEICHI TAKAHASHI (Member, IEEE) received the M.S. and Ph.D. degrees in information science from Osaka University, Osaka, Japan, in 2016 and 2019, respectively. In 2018, he was a Visiting Scholar with the Oak Ridge National Laboratory, Oak Ridge, TN, USA. He was an Assistant Professor with the Nara Institute of Science and Technology, Nara, Japan, from 2019 to 2021. He is currently an Assistant Professor with Tohoku University, Sendai, Japan. His research interests include high-performance computing and parallel distributed computing.



KOHEI ICHIKAWA (Member, IEEE) received the B.E., M.S., and Ph.D. degrees from Osaka University, in 2003, 2005, and 2008, respectively. He was a Postdoctoral Fellow with the Research Center of Socionetwork Strategies, Kansai University, from 2008 to 2009. He was an Assistant Professor with the Central Office for Information Infrastructure, Osaka University, from 2009 to 2012. He is currently an Associate Professor with the Division of Information Science, Nara Institute of Science and Technology (NAIST), Japan. His current research interests include distributed systems, virtualization technologies, and software-defined networking.

JOSEPH PARK (Senior Member, IEEE) received the B.S. and M.S. degrees in ocean engineering and the Ph.D. degree in electrical engineering from Florida Atlantic University. He is currently the Chief of Engineering and Development with the Comprehensive Nuclear-Test-Ban Treaty Organization, Vienna, Austria. His technical background is centered on acoustics, nonlinear dynamics, signal processing, control and systems engineering, numerical modeling, oceanography, ocean and structural engineering, sensors and instrumentation, information theory, electronics, and software development.

GERALD M. PAO received the B.S. degree in molecular biology from the University of California at San Diego and the Ph.D. degree in molecular biology from the Salk Institute working on cancer biology and virology. He was a Postdoctoral Fellow with the Salk Institute working on epigenetics of cancer and then in applied mathematics with the Scripps Institution of Oceanography before becoming a Staff Scientist with the Salk Institute. He was subsequently a Research Track Director (Research Fellow) for high throughput screening data science and gene therapy with Vertex Pharmaceuticals before coming back to academia. He is currently an Assistant Professor with the Biological Nonlinear Dynamics Data Science Unit, Okinawa Institute of Science and Technology. His research interests include big data nonlinear dynamics and causal inference in complex networks using high-performance computing.