Auto-HPCnet: An Automatic Framework to Build Neural Network-based Surrogate for High-Performance Computing Applications

Wenqian Dong
Florida International University and
University of California, Merced
Miami, Florida, USA
wdong@fiu.edu

Gokcen Kestor
Pacific Northwest National Laboratory
Richland, Washington, USA
gokcen.kestor@pnnl.gov

Dong Li
University of California, Merced
Merced, California, USA
dli35@ucmerced.edu

ABSTRACT

High-performance computing communities are increasingly adopting Neural Networks (NN) as surrogate models in their applications to generate scientific insights. Replacing an execution phase in the application with NN models can bring significant performance improvement. However, there is a lack of tools that can help domain scientists automatically apply NN-based surrogate models to HPC applications. We introduce a framework, named Auto-HPCnet, to democratize the usage of NN-based surrogates. Auto-HPCnet is the first end-to-end framework that makes past proposals for the NN-based surrogate model practical and disciplined. Auto-HPCnet introduces a workflow to address unique challenges when applying the approximation, such as feature acquisition and meeting the application-specific constraint on the quality of final computation outcome. We show that Auto-HPCnet can leverage NN for a set of HPC applications and achieve 5.50 speedup on average (up to 16.8 speedup and with data preparation cost included) while meeting the application-specific constraint on the final computation quality.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**; **Neural networks**; **Model development and analysis**.

ACM Reference Format:
Wenqian Dong, Gokcen Kestor, and Dong Li. 2023. Auto-HPCnet: An Automatic Framework to Build Neural Network-based Surrogate for High-Performance Computing Applications. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages.

1 INTRODUCTION

Many scientific simulations are complex and often used as a tool to obtain accurate and high-fidelity information about physical systems. However, those simulations often involve physics-based, closed-formed expression that can be evaluated in a finite number of standard operations. Those operations, which may come from a numerical solvers, can be time-consuming and difficult to be ported to emerging hardware (e.g., GPU-like accelerators).

The neural network-based surrogate model can address the above problem and recently shows its power in a wide range of HPC applications (e.g., earth systems [1], climate science [2, 3], turbulence modeling [4–6], computational physics [7], cyberphysical systems [8], material discovery [9–11], quantum chemistry [12, 13], biological sciences [14–16], and hydrology [17, 18]). The neural network-based surrogate model replaces a numerical solver or an execution phase (e.g., PCG [19] and FFT [20]) in the application with a neural network (NN) model. The NN model uses the same input/output as the solver or the execution phase, but brings large performance improvement to the application without violating numerical simulation correctness and stableness [19–22]. Using NN-based surrogate models, domain scientists are able to port the application to GPU (or other accelerators) that run NN workloads efficiently, even though the original application does not have any accelerator-based implementation. Even better, the NN-based surrogate models can be optimized by deterministic choices, such as the number of layers in NN and neuron size, to balance model prediction accuracy and cost.

Although using NN-based surrogate models to accelerate scientific simulations is promising, there is a lack of efforts that can automate the process of applying the NN-based surrogate models to scientific simulations (HPC applications). In practice, once the domain scientist selects a numerical solver in an HPC application to be accelerated by NN techniques, she has to manually (1) find appropriate input/output features of the to-be-replaced code region, (2) inject various input instances to the code region and collect the application's responding results (to be used as the ground truth to train the NN-based surrogate model), (3) determine the type of the NN model, (4) determine the NN topology (e.g., #layers and #neurons), and (5) apply the NN-based surrogate model to the target application and test the model performance. The above process is labor-intensive, and could be repeated multiple times before the NN-based surrogate model is finalized. As a result, there is a large gap between domain scientists and the usage of the NN-based surrogate model.

In this paper, we introduce Auto-HPCnet, to democratize the usage of NN-based surrogate models in HPC applications. Auto-HPCnet is the first end-to-end framework that makes past proposals for building NN-based surrogate models practical and disciplined.

HPDC '23, June 16–23, 2023, Orlando, FL, USA.

ACM ISBN 979-8-4007-0155-9/23/06…$15.00

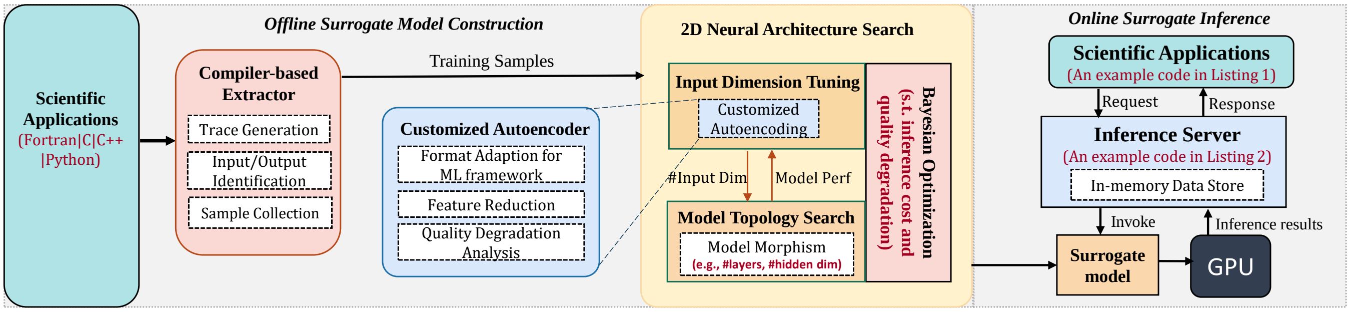


Figure 1: AutoHPCnet's workflow

Auto-HPCnet is based on our observations on multiple challenges of applying NN-based surrogate models in practice.

First, identifying and collecting the input/output features of the NN-based surrogate model is difficult. In a surrogate model, we must keep the same inputs/outputs as those of the to-be-replaced code region. The input variables (input features) are read inside the code region to update other variables; the output variables are updated in the code region and used after the code region. However, traditional NN-based surrogate models require domain scientists to provide highly relevant or well-designed input/output features, based on their extensive domain knowledge. Also, using the knowledge to identify features can introduce redundant input features, which makes the NN model heavy, or lose important output features, which leads to the crash or failure of the application.

Second, removing redundancy elements (e.g., zero elements) in sparse input variables of the NN model to construct an efficient but accurate NN model is a challenge. In HPC applications, we observe that input variables are usually sparse matrices stored in the format of Coordinate list (COO), Compressed Sparse Row (CSR), or Compressed Row Storage (CRS). However, despite being state-of-the-art DNN frameworks, PyTorch and TensorFlow have limited support for NN model training on sparse inputs, specifically when it comes to techniques like backpropagation. The existing DNN frameworks do not provide gradient descent functions to process the sparse matrices with common formats like CSR. Because of the lack of support for sparse matrices, it is inevitable to introduce transformations between the sparse format (COO, CSR, or CRS) and the dense format (a full matrix padding with zero elements). Such a transformation includes unrolling and re-construction processes, which introduces computation inefficiency. Also, the dense format causes storage inefficiency. For example, in NPB conjugate gradient application [23], the size of a single sparse matrix can increase 14× after re-construction.

Third, the feature reduction and selection of NN model topology are tightly coupled, and how to coordinate the two processes to minimize execution time and maximize the accuracy of the NN model is a challenge. The NN model topology refers to the number of network layers, the type of each layer (e.g., fully connected, convolution, deconvolution, or recurrent), and the number of neurons in each layer. Both the number of features and NN model topology impact model execution time and accuracy. The number of features determines the topology of the first layer in the NN model and impacts the design of the following layers; on the other hand, the topology selection of the NN model reflects feature eligibility. The existing Neural Architecture Search (NAS) methods [24–26] do not consider such interaction between the number of features and NN topology. Also, most of them consume text, audio, and images as model input, and have difficulty consuming data structures in HPC applications as input.

Fourth, how to enable the automatic construction of NN-based surrogate models and allow the user to efficiently explore the usage of surrogate models in a given HPC application is a challenge. This challenge includes how to build the whole workflow of finding NN models and making the workflow easy to use; this challenge also includes how to maximize the performance benefit while minimizing the user efforts, and how to integrate the automation into the user's decision-making process of using the NN-based surrogate models.

To address the above problems, we propose, Auto-HPCnet, a framework to construct NN-based surrogate models for HPC applications. Fig. 1 depicts Auto-HPCnet. **(1)** To address the first problem on the identification of input/output variables of the code region to use the NN model, Auto-HPCnet introduces a set of LLVM-based tools (labeled as *Compiler-based Extractor* in Fig. 1). Those tools instrument load and store instructions to trace memory read/write operations and enable the generation of a tree-based data dependency graph based on dynamic profiling; the tools also automatically analyze the graph to identify input/output and generate training samples based on the identified inputs/outputs.

(2) To address the second problem on the large sparse matrix of the NN model, Auto-HPCnet introduces an autoencoder-based feature reduction, i.e., *Customized Autoencoder* in Fig. 1. This mechanism provides painless support for the sparse matrix. During the offline training, the autoencoder adopts a gradient checkpoint technique to address the GPU memory limitation, which stores snapshots of the autoencoder parameters at the forward time to save memory space. During the online usage, the autoencoder uses a "TensorFlow embedding API" to directly take sparse matrices as input without unrolling effort.

(3) To address the third problem of coordinating feature reduction and selection of NN model topology, Auto-HPCnet introduces a *2D neural architecture search*. This strategy is automated. At the high level of this search process (input dimension tuning), we use the Bayesian optimization to decide the number of features; at the low level (the model topology search), we use another Bayesian optimization to decide the NN topology using an existing AutoML framework (particularly Autokeras [24]). The low level of the search is based on the decision of the high level. The two levels work iteratively and coordinately to consider the impact of both feature reduction and NN topology. Furthermore, we consider both execution time and correctness of using the NN-based surrogate model during the 2D neural architecture search. We introduce a user-given threshold as an application-specific metric and incorporate the metric into the search of the NN model, which ensures the correctness of the application's final output.

(4) Putting together a set of tools and strategies, Auto-HPCnet builds a workflow that relieves the domain scientist from labor-intensive work to apply NN-based surrogate models to HPC applications. The paper makes the following contributions.

- We introduce a framework that enables an automatic construction and use of NN-based surrogate models in HPC applications.
- We introduce a workflow and a set of techniques in Auto-HPCnet to address unique challenges when applying the NN-based surrogate method to HPC applications.
- We demonstrate the effectiveness of Auto-HPCnet by applying it to a set of HPC applications. Our experiments show that with Auto-HPCnet, the applications can achieve 5.50× speedup on average (up to 16.8× and with data preparation cost included) without loss in the final computation quality by replacing execution phases with NN-based models. We show that in terms of speedup, Auto-HPCnet can generate NN models outperforming those models generated by the state-of-the-art methodologies including a competitive AutoML framework (Autokeras), a manual NN construction tool (ACCEPT), and a computation-approximation strategy (i.e., the loop perforation).

2 BACKGROUND AND MOTIVATION

2.1 Characteristics of HPC Code Regions

In this paper, we investigate the use of neural network-based surrogates as a means of approximating specific code regions within HPC applications. The selection of the code region to be replaced is left to the discretion of the user, and can range from a computation loop, a set of functions, to the entire application. In this section, we provide some insights on which/why HPC applications can benefit from NN-based surrogates.

Many HPC applications contain computation-intensive code with inevitable data dependencies, which can hinder parallel performance. For instance, many HPC applications in fields such as mathematics, physics, chemistry, and other natural sciences require the solution of large systems of linear equations. The solving procedure for these systems is often the dominant contributor to runtime, but as the systems are often sparse, it is not practical to invert them. Practitioners therefore often rely on iterative algorithms to converge to a useful solution, which can be time-consuming. Moreover, these iterative algorithms, such as the Preconditioned Conjugate Gradients (PCG) method, are closed-form optimization solvers and are difficult to parallelize efficiently on general-purpose platforms like CPUs and GPUs [27, 28].

Algorithm 1 Algorithm of PCG solver in fluid simulation

```
1:  Initialize $x_0$, $r_0 = b - Ax_0$, $p_0 = r_0$
2:  for $n \leftarrow 1$ to $N$ do
3:      // compute the preconditioner-vector product $Ap_i$
4:      $Ap_i = A * p_i$
5:      $alpha = (r_{i-1} * r_{i-1})/p_i * Ap_i$
6:      // update the solution
7:      $x_i = x_{i-1} + alpha * p_i$
8:      // update the residual
9:      $ri = r_{i-1} - alpha * Ap_i$
10:     $beta = (r_i * r_i)/(ri - 1 * r_{i-1})$
11:     $pi = r_i + beta * p_{i-1}$
12:     if $\| ri \|$<tolerance:
13:         break
14: end for
15: return $x$
```

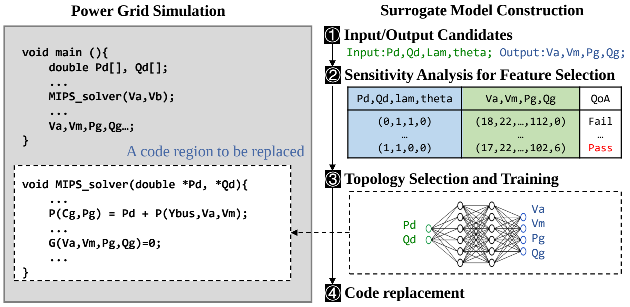
Algorithm 1 is an example of the PCG method used in fluid simulation, and it dominates the runtime of the application. The algorithm has a Read-After-Write (RAW) data dependency at Line 9 (r_i is computed using the updated value of x_i from the previous iteration) and Line 11 (p_i is computed using the updated value of r_i from the previous iteration). This type of dependency can lead to a series of performance problems when attempting to parallelize the code on GPUs, such as how to leverage massive parallelism, how to coalesce memory transactions, and how to avoid memory-bank conflicts.

NN models can serve as an alternative to computation-intensive code regions in HPC applications with data dependencies. NN models can be easily parallelized on emerging hardware like GPUs, due to their ability to leverage massive parallelism, coalesce memory transactions, and avoid memory-bank conflicts. Furthermore, NN models can also be implemented on other emerging hardware such as FPGAs and Sambanova, further increasing the potential for performance benefits and acceleration in HPC applications. This versatility in hardware implementation further motivates practitioners to use NN models as a tool for acceleration in these types of applications.

HPC applications can tolerate approximation. HPC applications can tolerate computation inaccuracy caused by approximation. This has been demonstrated in existing literature, such as [29–33]. In many cases, the accuracy of the final result is not as crucial as the speed at which it is obtained. For example, in fluid simulations, the goal may be to obtain a qualitative understanding of the fluid behavior rather than an exact solution. In such cases, the use of approximate methods, such as NN-based surrogates, can be an effective way to accelerate the computation while still providing useful results.

One example of this is the use of NN models to surrogate the PCG method in fluid simulations[19, 34]. The PCG method is highly computation-intensive, especially when the computation problems are large and sparse. By using an NN-based surrogate, it is possible to approximate the PCG method and reduce the computation time, while still obtaining accurate results. This is particularly beneficial when the simulation is performed on emerging hardware, such as GPUs, which can take advantage of the parallelism of the NN-based surrogates to achieve faster computation.

Also, many HPC applications have a threshold to determine when the final application outcome is acceptable or when the simulation should be terminated. In fluid simulation, the convergence of the


Power Grid Simulation
Surrogate Model Construction
① Input/Output Candidates
② Sensitivity Analysis for Feature Selection
③ Topology Selection and Training
④ Code replacement
A code region to be replaced
Pd
Qd
Va
Vm
Pg
Qg

```
void main (){
    double Pd[], Qd[];
    ...
    MIPS_solver(Va,Vb);
    ...
    Va,Vm,Pg,Qg…;
}
```

```
void MIPS_solver(double *Pd, *Qd){
    ...
    P(Cg,Pg) = Pd + P(Ybus,Va,Vm);
    ...
    G(Va,Vm,Pg,Qg)=0;
    ...
}
```

```
Input:Pd,Qd,Lam,theta; Output:Va,Vm,Pg,Qg;
```

Pd,Qd,lam,theta	Va,Vm,Pg,Qg	QoA
(0,1,1,0)	(18,22,…,112,0)	Fail
…	…	…
(1,1,0,0)	(17,22,…,102,6)	Pass

Figure 2: An example of applying the surrogate model.

solution obtained by the PCG method may be deemed as acceptable even if the residual is not exactly zero. The threshold-based approach also exists in molecular dynamics [35], numerical weather prediction [36], and computational electromagnetics [22], etc. This threshold-based approach allows HPC applications to accept the NN-based surrogate model.

2.2 Traditional v.s. NN Approximation

In the field of numerical code optimization, traditional approximation techniques such as loop perforation, tiling, and interpolation have been widely utilized. Loop perforation, as described in literature such as [37, 38], involves removing or skipping certain iterations of a loop in order to reduce the number of operations that need to be performed. Tiling, as outlined in [39], breaks a large computation into smaller, more manageable chunks known as tiles, which can then be processed independently, resulting in greater parallelism and improved cache utilization. Interpolation, as presented in [40], is a technique for estimating the value of a function at a point where the function is not explicitly defined, utilizing known data points, or sample points, to infer the value of the function at other points.

Despite the widespread utilization of these traditional approximation techniques, they do possess certain limitations. Firstly, their applicability is often restricted to specific types of problems or code, resulting in a lack of generality. Secondly, these techniques typically rely on simple mathematical functions to approximate the behavior of the original code, which can result in a significant loss of accuracy. Lastly, the parameter tuning of these approximation techniques requires expert knowledge, which can make them difficult for non-experts to utilize.

In contrast, NN-based approximation methods (i.e., NN surrogates) have been proposed as a potential solution to these limitations. NNs possess the ability to approximate complex functions with **high accuracy**, as they are able to learn underlying patterns in the data and generalize well to new examples. Also, NNs can handle non-linear relationships and adapt to changes in the input data. Additionally, neural networks can be trained on **a wide variety of tasks** and generalized to new tasks with similar characteristics, resulting in ease of integration into existing codebases and accessibility to a wide range of users. Furthermore, the use of pre-trained models and libraries such as Tensorflow or Pytorch make NN surrogate methods **easy to use** even for those without expertise.

2.3 Traditional Workflow of Surrogate Model Construction

Fig. 2 shows an example of surrogate model construction, which includes four steps:

(1) The user chooses a code region for replacement and manually examines it to identify the input and output variables of the code region. In the power-grid simulation, the "MIPS solver" is identified as the most time-consuming part and replaced with a surrogate model. The manual identification of input and output variables is error-prone, especially when the number of input/output variables is large.

(2) The user conducts a sensitivity analysis to remove redundant features to build an efficient but accurate approximation model. A large amount of the input usually requires more computation units (e.g., neurons and layers) to maintain the performance of prediction accuracy. To mitigate the overhead of NN models, scientists [22] usually adopt feature selection to reduce the space of input variables and remove the implicate redundancy.

(3) The user manually constructs a surrogate model to replace a code region. The surrogate construction is a process of selecting a network topology that balances accuracy and cost (e.g., execution time, Flops). A larger, more complex surrogate model has the potential to offer better model accuracy but is likely to be slower during inference than a smaller model.

(4) After the surrogate model is well-trained, the user needs to either integrate the NN model into the application using cross-compiling, or take the NN model as an offline calculator. This is because, the codebases of the replaced numerical solvers are typically written in Fortran/C/C++ and run on HPC platforms (HPC) via OpenMP and/or MPI parallelization, while the emerging ML and data analytics libraries of NN-based surrogate model are typically written in Python.

The current state-of-the-art approaches for HPC applications perform these steps manually [19–22, 34, 41, 42], which is labor-intensive and motivates us to create a framework to automate this process.

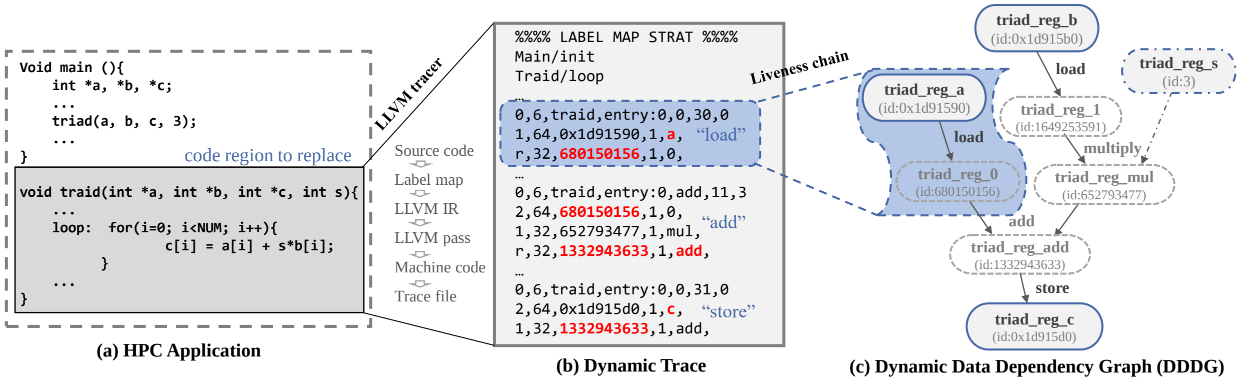
3 DATA ACQUISITION

Auto-HPCnet first identifies the input/output features and collects training samples based on identified features.

3.1 Compiler-based Feature Extraction

Given a code region selected by the user, Auto-HPCnet classifies variables within the code region as input variables, output variables, and internal variables. *Input variables* are declared outside of the code region and referenced in the code region. *Output variables* are written in the code region and read after the code region. Other variables that the code region writes to or reads from are internal variables. Given a target code region, Auto-HPCnet uses the following steps to acquire input and output variables.

Step1: Trace generation. Auto-HPCnet integrates an LLVM tool, LLVM-Tracer [43] which is an LLVM instrumentation pass to generate a dynamic LLVM instruction trace. This trace stores metadata for each instruction, such as the instruction type, names of registers, and operand values. Fig. 3 shows an example to depict trace generation.



```
Void main (){
    int *a, *b, *c;
    ...
    triad(a, b, c, 3);
    ...
}
```

```
void traid(int *a, int *b, int *c, int s){
    ...
    loop:  for(i=0; i<NUM; i++){
                    c[i] = a[i] + s*b[i];
           }
    ...
}
```

```
%%%% LABEL MAP STRAT %%%%
Main/init
Traid/loop
…
0,6,traid,entry:0,0,30,0
1,64,0x1d91590,1,a,  “load”
r,32,680150156,1,0,
…
0,6,traid,entry:0,add,11,3
2,64,680150156,1,0,
1,32,652793477,1,mul, “add”
r,32,1332943633,1,add,
…
0,6,traid,entry:0,0,31,0
2,64,0x1d915d0,1,c,  “store”
1,32,1332943633,1,add,
```

(a) HPC Application

(b) Dynamic Trace

(c) Dynamic Data Dependency Graph (DDDG)

Figure 3: An example of acquiring input and output variables.

Fig. 3(b) shows the trace for an example code. Auto-HPCnet extends LLVM-Tracer to reduce the trace size to simplify the identification process of input/output variables. In particular, during the trace generation, Auto-HPCnet recognizes loop structures in the code region. If a loop has no control flow divergence across iterations of the loop and accessed (array) variables across iterations remain the same, then Auto-HPCnet does not generate the whole trace for the loop. Instead, only the trace for one iteration is generated.

Step2: Identification of input and output variables. We construct a dynamic data dependency graph (DDDG) from the instruction trace based on the existing method [44]. In DDDG, vertices are the values of variables obtained from registers or memory; Edges are LLVM instructions (or operations) transforming input values into output values of variables. With DDDG, the root nodes represent inputs, and the leaf nodes represent outputs. Only taking the outputs from the DDDG is not sufficient. There may exist some intermediate variables used for the following code region. Hence, Auto-HPCnet uses a combination of liveness analysis [45] and use-def chain [46] to examine other output features.

We extend the construction of DDDG in [44] to fit the needs of the surrogate model from two perspectives. *First*, we group variables for effective feature reduction. In particular, the number of input variables recognized by DDDG can be large. Some of those variables can come from the same array; The large number of input variables can come from multiple arrays. During the feature reduction phase, some individual variables used as individual input features can be selected together for reduction, even though they come from different arrays. Using those variables as individual input features loses the array semantics, which leads to either ineffective feature reduction or lower accuracy in the surrogate model. Hence, during the identification of input variables, if some variables come from the same array, then the array (not individual variables) is used as the input feature of the surrogate model. *Second*, we parallelize the construction of DDDG to shorten the construction time and make it user-friendly. In particular, instead of processing instructions one by one, we process a group of instructions by multiple threads at the same time, which allows us to explore thread-level parallelism to accelerate instruction analysis when there is less dependence between instructions within the group.

Step3: Generating Training Samples. Training a surrogate model needs many training samples to ensure the model is sufficiently trained. A training sample is a pair of input features and output features, where the input and output features come from the values of the input and output variables of the target code region respectively. In cases that the user cannot find enough input problems to generate training samples, Auto-HPCnet allows the user to introduce perturbation into the values of input variables. The perturbation follows a specific distribution, such as the Gaussian distribution, i.e, $X' \sim \mathcal{N}(\mu, \sigma^2)$, where X’ is the randomized new sample given a predefined mean μ and variance σ. The distribution can be chosen by the user based on the application domain knowledge.

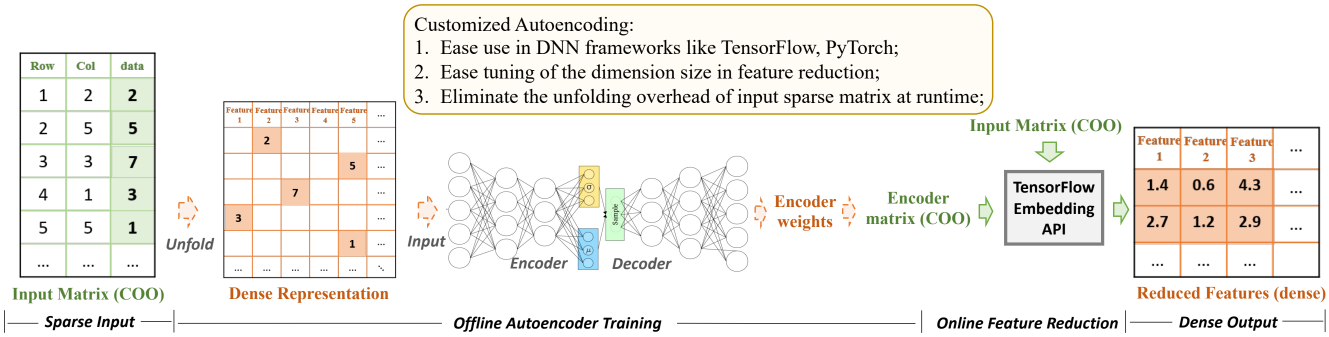
3.2 Dynamic Analysis v.s. Static Analysis

The NN-based surrogate has a common assumption that the identification of input/output variables (or features) is invariant across different executions of the code region. Based on this assumption, a dynamic execution trace can be used to identify input/output for a *specific* NN model for approximation. If the execution path divergence leads to different input/output, we should use a different NN model. In other words, an NN model can only be used to approximate the code segment with certain input/output (following certain distributions that lead to the same execution path), but not all input/output.

Static analysis has the risk of introducing redundant features (some input/output variables identified by static analysis may not be referenced in a specific execution), which leads to larger NN models and loss of performance benefit. Dynamic analysis can avoid this problem, but the NN model generated based on dynamic analysis can only work for certain inputs following a certain distribution. In this paper, we choose dynamic analysis over static analysis to maximize performance benefits.

4 INPUT ANALYSIS

We depict how Auto-HPCnet uses autoencoder with customized designs to reduce the dimensionality of sparse input matrix in this section.



Customized Autoencoding:
1. Ease use in DNN frameworks like TensorFlow, PyTorch;
2. Ease tuning of the dimension size in feature reduction;
3. Eliminate the unfolding overhead of input sparse matrix at runtime;

Row	Col	data
1	2	2
2	5	5
3	3	7
4	1	3
5	5	1
...	...	...

Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	...
	2				...
				5	...
		7			...
3					...
				1	...
...	...	...	...	...	⋱

Feature 1	Feature 2	Feature 3	...
1.4	0.6	4.3	...
2.7	1.2	2.9	...
...	...	...	

Figure 4: The workflow of applying Autoencoder in Auto-HPCnet

4.1 Autoencoders for Feature Reduction

Autoencoders aims to use a deep learning architecture to capture key representational information by mapping the high-dimensional data into a low-dimensional space [47]. Unlike traditional feature extraction technologies, Autoencoder provides an unsupervised method to do feature reduction without requiring priori knowledge on application domains.

An autoencoder consists of an encoder and a decoder. ❶ **Encoder.** The encoder typically has an hourglass-shaped architecture in which the high-dimensional data is compressed into a low-dimensional latent space that preserves semantic relationships. The encoder takes a whole matrix as input for feature reduction. ❷ **Decoder.** The decoder uses a horn-shaped network to reconstruct the reduced features back to the original representation (raw inputs). The weights of the autoencoder (both encoder and decoder) are tuned together to minimize a loss function, which typically penalizes deviations between the input of encoder and the output of decoder. Once the autoencoder is trained, the encoder is used to reduce features discussed in Section 5.

4.2 Customized Design for Sparse Input

Fig. 4 depicts the workflow of applying the autoencoder. In Autoencoder, we adopt the following designs to address problems when dealing with sparse inputs.

First, during offline training, we adopt a gradient checkpoint technique [48] to address the GPU memory limitation. In particular, since the existing DNN frameworks have limited support for DNN training on sparse matrix formats, if we unroll those sparse inputs to dense representation during the autoencoder training, the memory consumption to store the dense representation becomes a bottleneck. To address this problem, we adopt the gradient checkpoint technique [48], which stores snapshots of Autoencoder parameters at a forward time and recomputes those parameters at a backward time. The gradient checkpoint technique trades the computational cost (recalculation time) for GPU memory usage (parameter storage).

Second, at online feature reduction, we provide an API for sparse matrix formats without any decompression effort. After Autoencoder is trained and no optimization is applied at online usage, the user still has to take the input matrix with dense format (which requires decompression). Here, we apply a "TensorFlow embedding API". This API conducts matrix-multiplication in the format of sparse representation (e.g., CSR) and saves the multiplication result into the dense format. Autoencoder takes this API to implement the matrix-multiplication function at the first layer of Autoencoder, which helps the Autoencoder directly take the sparse matrix as an input without decompression. By doing so, we provide painless support for sparse matrices in the HPC application by eliminating both the temporal cost (format transformation of decompression) and spatial cost (memory usage of storing the dense representation) in the feature reduction process.

Third, we develop a metric to quantify the quality degradation in real-time and the user can define a lower bound of quality constraint to guide the encoding process. In traditional methods like K-means and PCA, it is hard to quantitatively measure the quality degradation before and after the feature reduction, because the size of the output matrix is not the same as the size of the input matrix. Here, we take the advantage that the autoencoder can reconstruct an output matrix (which has the same size as the input matrix) to do an element-by-element comparison. We propose a metric to evaluate the difference between the original input and the reconstructed matrix (i.e., the decoder output).

$$\sigma_y = \frac{1}{N}\sum_{i=1}^{N}\begin{cases} 0 & \text{if } |y_i - x_i| \le \mu |x_i| \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

We denote the matrix σ_y, defined by comparing each element y_i in the reconstructed matrix to its corresponding value x_i in the original sparse matrix and calculate the proportion of those elements that are at least in a feasible range away from x. The user determines a scaling factor μ based on the application domain requirement. Computing the similarity σ_y is lightweight and can be done on-the-fly during the autoencoder training. Based on the metric, the user can configure the lower bound (shown in Table 1) for different tasks.

4.3 Workflow of Applying Autoencoding

The workflow of autoencoder (shown in Fig. 4) has two parts, (1) offline training happened during the Bayesian optimization and (2) online usage when the HPC application is running. The offline training happens in high-level Bayesian optimization. The high-level Bayesian optimization is a loop structure (discussed in detail in Section 5.2).

In each iteration of the loop, Auto-HPCnet trains a new autoencoder. Across iterations, different autoencoders generate different numbers of features (the number of features is determined by the low-level Bayesian optimization and the Gaussian process in the high-level Bayesian optimization). After an autoencoder is trained in an iteration of the high-level Bayesian optimization, the encoder is then used in the low-level Bayesian optimization for encoder-model inference to generate reduced features for the NN architecture search. After the hierarchical Bayesian optimization is done, the autoencoder trained in the last iteration of the high-level Bayesian optimization is used for online usage.

5 2D NEURAL ARCHITECTURE SEARCH

Auto-HPCnet uses a two-dimensional neural architecture search to jointly decide NN architecture (i.e., NN topology) and the number of features. In this section, we first formulate our optimization problem. Then, we discuss algorithm details for the Bayesian optimization.

5.1 Search Space

We describe the formulation of the 2D neural architecture search as follows: given an input dataset and bound on the output error, find the best surrogate model that (i) meet the error bound and (ii) minimize the cost. This can be formulated as the following constrained optimization problem.

Problem Formulation. *Given:*

- K: a set of tunable input dimension, and
- θ: a set of tunable surrogate topology parameters.

Find the best K', θ' such that

- $f_c(K', \theta')$ is minimized
- $f_e(K', \theta') \leq \epsilon$.
- $T(f)$ is minimized
- $SE(f) \leq SE_{obj}$.

where θ' includes #kernel sizes, #channel, #pooling size, #unpooling size, and #residual connection of each layer. We involve three functions here.

- Output: Let $f(K', \theta')$ represent the function of the NN-based surrogate model to generate output.
- Quality degradation: Let $f_e(K', \theta')$ be the magnitude of the final computational quality degradation of the application.
- Cost: Let $f_c(K', \theta')$ be the cost of computing the output at runtime. This can be the running time, energy or other execution metric to be optimized.

A solution to the optimization problem is a point that lies within the feasible region $f_e(K', \theta') \leq \epsilon$ and minimizes the objective function $f_c(K', \theta')$.

5.2 Hierarchical Bayesian Optimization

The Bayesian optimization [49] has been used in architecture parameter tuning for machine learning models [50], in which Bayesian optimization searches among different combinations of architecture parameters. Many Bayesian optimization processes use a Gaussian process model to model the objective function f and an acquisition function to decide where to do the next evaluation. The traditional Bayesian optimization is an iterative process consisting of three steps: update, generation, and evaluation; (1) Update: train a Gaussian process model with a combination of an optimization vector and performance (in the case of NN architecture search, the performance is the NN model cost f_c and error f_e); (2) Generation: generate a new combination to observe by optimizing an acquisition function; and (3) Evaluation: apply the new combination to the optimization target (in the case of NN architecture search, this means we use the optimization vector to train an NN model) and measure performance.

Algorithm 2 Hierarchical Bayesian optimization

```
Require: Input dataset $D$; Acceptable quality degradation $f_e$;
1: Given search space: $\theta = \{\theta_1, .., \theta_n\} \cup K = \{k_1, k_2, ..., k_m\}$;
2: do   // Outer Loop
3: $k_i$ = initRandom();
4: $D(k_i)$ = FeatureReduction($k_i$);
    5: do   // Inner Loop:
    6: $\theta_i$ = initModel($D(k_i)$);
    7: $f_e, f_c$ = problem.evaluate($\theta_i$);
    8: model = GaussianProcess();
    9: model.update();
    10: while run out of searching time.
    11: Return the performance of best NN model $f'$ in terms of $f'_c$ and $f'_e$.
12: model = GaussianProcess($_n$, $f'$);
13: model.update();
14: while $f_c*$ is minimized and $f_e* < \epsilon$.
15: return the optimal input size $i*$ and topology parameters $k*$.
```

Algorithm. The optimization vector includes two types of parameters: (1) the feature reduction knobs K and (2) the NN architecture parameters θ. In the Gaussian process used in the Bayesian optimization, the optimization vector has to be vectorized in a Euclidean space because the Bayesian optimization must measure the distance between different optimization vectors. Because of the difference in physical properties, arithmetically adding the two types of parameters loses the parameter semantics, which leads to a suboptimal selection of parameters. To address the above problem, we introduce a two-level optimization strategy, which separates the optimization processes for the two types of parameters, but coordinates the two separate processes for optimal selection of those parameters.

Algorithm 2 depicts the strategy in our hierarchical Bayesian optimization. There is a two-level loop in the algorithm. The *high-level loop* (Lines 2-13) searches for the optimal setting K_i of the input dimension. The *low-level loop* (Lines 5-10) searches the best architecture parameters θ_i of the surrogate model. These two loops can interact with each other: the high-level loop generates an input sample with the dimension of K_n, which is then applied in the low-level loop during the architecture parameter search (Line 6); the low-level loop returns the performance (f_c and f_e) of the best model to the high-level loop (Line 11). Then the high-level loop makes a response based on an acquisition function to determine the next promising search point (i.e., k_{n+1}) and trains a corresponding autoencoder.

In the high-level loop, we apply the customized autoencoder (in Section 4.2) to conduct the feature reduction. In the low-level loop, we apply Autokeras [26] for model architecture search. In each iteration of the search (in either low-level loop or high-level loop), we apply the regular Bayesian optimization, which iteratively searches for the optimal by the three steps: update, generation, and evaluation. The whole search process is terminated, if the required

Table 1: Configurations in AutoHPCnet.

Search-level	–searchType	(1) “autokeras" (start with the Autokera’s default topology) (2) “userModel" (start with a user-given topology) (3) “fullInput" (no feature reduction applied)
	–bayesianInit	Initial samples for bayesian algorithm
	–encodingLoss	Acceptable encoding loss
	–qualityLoss	Acceptable quality loss
Model-level	–initModel	Surrogate model type (default=MLP)
	–preprocessing	Training data preprocessing
	–numEpoch	Number of epochs to train
	–trainRatio	Split dataset into training and validation
	–batchSize	Batch size
	–lr	Learning rate

performance f_e and f_c is achieved or a continuing search does not lead to enough improvement of the performance.

6 IMPLEMENTATION

We discuss the implementation details in this section.

6.1 Interaction with Users

To allow the user to annotate a code region for approximation with a surrogate model without hassle, Auto-HPCnet introduces two directives to mark the boundary of the code region. The annotation controls the LLVM instrumentation for trace generation, which in turn controls the usage of the surrogate model. After the annotation and building the application with LLVM, the user is expected to run a script to trigger the following workflow: (1) running the application to generate the LLVM instruction trace, (2) analyzing the trace to identify input and output variables in Auto-HPCnet, and (3) uses a script to run the application N times to collect training samples (at each time, the script triggers a perturbation of input variables to collect the output of the code region).

After collecting training samples, AutoHPCnet employs the 2D neural architecture search (incorporating the customized Autoencoder). As shown in Table 1, Auto-HPCnet gives two sets of configurations to accommodate the needs of different users. The first set, named as search-level, allows the user to easily configure the parameters of Bayesian optimization algorithm. The initial model architecture (i.e., “searchType”) can be specified based on the user’s knowledge to accelerate the search process. Also, the user is able to specify the initial sample size and the objectives of the Bayesian optimization (i.e., the acceptable encoding loss and acceptable quality loss). The initial samples are used to generate a Gaussian process model for exploring the best solution in Bayesian optimization. The second level, named model-level, is used to tune the hyperparameters (e.g., batch size and learning rate) for the surrogate model training. Also, the user can search for a specific type of neural network architectures (e.g., multi-layer perceptron or CNN).

Besides the above, Auto-HPCnet has a checkpoint mechanism that allows the user to stop and restore the model architecture search. Auto-HPCnet also allows the user to easily save and share the Autoencoder and the surrogate model across applications.

6.2 Quality-Oriented Optimizations

We summarize the implementation details in Auto-HPCnet with the awareness of the final computational outcome quality in this section.

Listing 1: Example of HPC simulation for surrogate request

```
#include “autoHPCnet_client.h”

// Initialize a Client object
autoHPCnet::Client client(false);
// Put the input features on the database
client.put_tensor(in_key, autoHPCnet);
// Run model already in the database
client.run_model(“AI-CFD-net”, {in_key}, {out_key});

// Get the result of the model
client.unpack_tensor(out_key, autoHPCnet);
```

Listing 2: Example of invoking surrogate model

```
from autoHPCnet import Client
from smartsim.database import Orchestrator
# import other packages …

# create and start a database
orc = Orchestrator(port=REDIS_PORT)
exp.generate(orc)
exp.start(orc, block=False)

# get input from database
sparse_tensor = client.get_tensor(input_feature)

# feature reduction and format transformation
compact_tensor = client.autoencoder(sparse_tensor)

# load a pretrained model from file
client.set_model_from_file("AI-CFD-net", "./saved_net.pt",
"TORCH", "GPU")

# Run model and retrieve outputs
client.run_model("AI-CFD-net", inputs=compact_tensor,
outputs=output_tensor)
```

Such awareness separates us from the existing AutoML tools like Google AutoML [25] and Autokeras [24].

Error-bounded Feature Reduction in AutoEncoder. To meet the constraint on quality degradation in the HPC application, the feature reduction, which is used to optimize the surrogate model in our work (Section 4.2), must consider the impact of the feature reduction on the final computation quality. Auto-HPCnet implements the API ”$Autoencoder.evl(\#inputs, \#compaction)$”. This API measure the quality degradation before ($\#inputs$) and after (#outputs) the feature reduction, using the metric defined in Eqn 1.

Auto-HPCnet also has the innovation of coupling the feature reduction and surrogate model construction through the 2D neural architecture search (Section 5).

6.3 Online Inference Invocation

To allow the user to easily integrate the surrogate model into an application, Auto-HPCnet provides two libraries: (1) a lightweight library working as a request client and compiled into the application to request the surrogate inference, and (2) a server client library to conduct NN inferences on GPU. Auto-HPCnet use SmartSim Orchestrator [51] to set up a in-memory storage to enable data sharing between the HPC application written in Fortran, C, C++, or Python, and NN models written with TensorFlow, Keras, or Pytorch.

Auto-HPCnet and SmartSim Orchestrator are coupled to work together. When launched through Orchestrator, applications using

Table 2: Applications for Evaluation.

Type	**Application:** replaced function	Description	Quality of Interest (QoI)
I	**CG**:CG_solver	Conjugate Gradient	Solution of linear equations
	FFT:FFT_solver	Fast Fourier Transform	Output sequence of FFT
	MG:MG_solver	Multi-Grid method	The final residual of the solver
II	**Blackscholes**:$BlkSchlsEqEuroNoDiv$	Investment pricing	The computed price
	Canneal:$Annealing$	VLSI routing	Routing cost
	fluidanimation:$NS\_equation$	Fluid dynamics	Particle distance
	streamcluster:$Dimension\_reduction$	Online clustering	Cluster center distance
	X264:$Encoding$	Video encoding	Structure similarity
III	**miniQMC:**$Determinant$	Quantum Monte Carlo	Particle energy
	AMG:PCG_solver	Solver of linear systems	Solution of linear systems
	Laghos: $SolveVelocity$	Compressible gas dynamics	Velocity Divergence

the Auto-HPCnet clients are directly connected to any Orchestrator launched in the same experiment.

This is because AutoHPCnet client adopts a Redis module (i.e., RedisAI [52]), provides the NN runtimes, creating a library agnostic middleware between the HPC application and NN libraries. Because of this middleware, the user of AutoHPCnet can smoothly switch between the NN framework and HPC application. This method greatly reduces the deployment complexity and overhead of adding the surrogate model in the HPC application.

Making Inference Call in Auto-HPCnet. Listing 1 shows an example of requesting NN inferences based on Auto-HPCnet client in a C-based application. The client first sends the input tensors to the inference server (Line 5) and then makes an inference call to the server. Using the above approach simplifies implementations: the user only needs to change a few lines of code in the application.

SmartSim Implementation of Inference Call. Listing 2 shows how to spin up a database with SmartSim Orchestrator and invoke a CNN model using the Auto-HPCnet client. In Listing 2, the "exp.start(orc, block=Flase)" (Line 8) uses the SmartSim library to launch a in-memory data storage. The server receives the inference request and fetches the input data from the storage (Line 11). Then, the server loads the pre-trained autoencoder (Line 14) and surrogate model (Line 17) to make an inference on GPU (Line 21). Despite writing in Python, we execute all surrogate models in C runtime.

7 EVALUATION

Platform. We conduct all experiments on an NVIDIA DGX-1 cluster with 8 nodes, and each node is equipped with two Intel Xeon E5-2698 v4 CPUs (40 cores in total running at 2.20GHz) and eight NVIDIA TESLA V100 (Volta) GPUs. We use CUDA 10.1/cuDNN 7.0 [53] and Tensorflow 2.3 [54] for model training and inference.

Applications. Table 2 lists applications we evaluate. We comprehensively cover three types of applications, which have been widely studied in HPC. Type-I includes numerical solvers that are often the most time-consuming in HPC applications. Type-II includes a set of general applications from the PARSEC parallel benchmark suite [55]. Those applications are evaluated in previous efforts [56–58] for approximate computing. Type-III is the representative of large-scale HPC applications. Type-III comes from the Exascale Computing Project (ECP) Proxy Applications Suite 4.0 [59].

Quality of Interest (QoI). To evaluate the final computation quality of the application, we assume that the user provides application-specific QoI that can be used to quantify the difference between the solution of the surrogate model and the exact solution. Table 2 lists the QoI of each application. The QoI differs among applications.

7.1 Auto-HPCnet Effectiveness

We use two metrics to evaluate Auto-HPCnet effectiveness: speedup and prediction hit rate. The speedup is used to evaluate the performance of Auto-HPCnet, and the prediction hit rate is used to evaluate the quality of the surrogate models generated by Auto-HPCnet. Equation 2 defines the speedup. We report the speedup of the whole application (instead of only the NN-replaced code region).

$$Speedup = \frac{T_{Numerical\_solver}}{T'_{NN\_infer} + T'_{Data\_load} + T_{Other\_part}} \tag{2}$$

where $T_{Numerical\_solver}$ represents the execution time of the application using the original code (e.g., a traditional numerical solver). T'_{NN_infer} is the inference time of the surrogate model generated by Auto-HPCnet and T'_{Data_load} is the data communication overhead for loading the NN model input to GPU. T_{Other_part} refers to the execution time of the rest part (the code regions without applying the NN surrogate model).

Equation 3 defines the prediction hit rate (i.e., $HitRate$), which refers to the ratio of the number of input problems that can reach the quality requirement with the NN surrogate models, to the total number of input problems (N):

$$HitRate = \frac{1}{N}\sum_{i=1}^{N}(1, \text{ if } |V'_i - V_i| \leq \mu|V_i|) \tag{3}$$

Where V is the user-specified QoI, V'_i is the calculated QoI after the surrogate model is applied to the application with the ith input problem, and V_i is the calculated QoI without applying the surrogate model to the application with the ith input problem. The difference between V'_i and V_i should be smaller than $\mu|V_i|$ in order to claim that applying the surrogate model to the application with the ith input problem generates a high-quality application outcome that meets the user's quality requirement.

$\sum_{i=1}^{N}(1, \text{ if } |V'_i - V_i| \leq \mu|V_i|)$ in Equation 3 counts the total number of input problems that can meet the user's quality requirement after applying the surrogate model. μ is a parameter set by the user (see Section 5). In our evaluation, μ is set as 10%, which is aligned with the existing efforts [56, 57, 60] for neural network-based computation approximation.

Using the two metrics, we evaluate Auto-HPCnet with 11 applications. Each application use 2,000 input problems for evaluation. Fig. 5 shows the results.

Performance. There is $1.89\times$ - $16.8\times$ speedup with a harmonic mean of $5.50\times$ across all three types of application, compared with the application performance on CPU (using all 40 cores). Among all applications, Blackscholes has the largest speedup. The large speedup comes from the fact that the surrogate model removes all control flows in the original code, and Auto-HPCnet is able to offload BlkSchlsEqEuroNoDiv (the most computation-intensive part) to GPU.

To further study the performance, we compare the performance of using the original code on GPU and using the surrogate models generated by Auto-HPCnet on GPU. Table 3 shows the results for AMG, a production code that can run on either CPU or GPU. To run AMG on GPU, we use AMGX [61]. We observe that Auto-HPCnet leads to $4.14\times$ better performance than AMGX. To look into why the surrogate model on GPU performs better, we measure the number

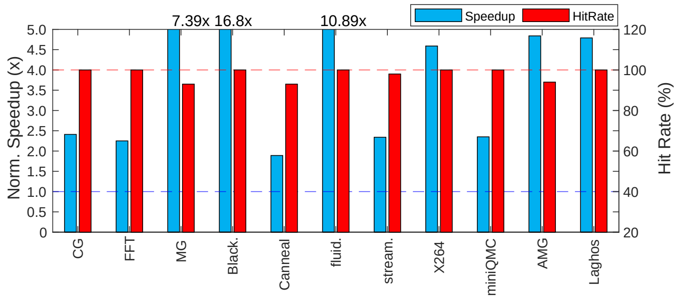


Figure 5: Speedup and prediction HitRate in Auto-HPCnet.

Table 3: Compare the performance of Auto-HPCnet on GPU with the performance of the original code on GPU. The results are for AMG.

Methods	CPU-only	Original code on GPU	Auto-HPCnet on GPU
Floating-Point Operations	30.66G	72.82G	21.97G
L2 level cache-miss rate	37.47%	26.31%	17.81%
Mem Bandwidth (MB/s)	3523.15	7518.85	6735.54
Wall clock time (seconds)	2.47	2.11	0.51

of Floating-Point (FP) operations, last-level cache miss rate, and (global) memory bandwidth consumption, shown in Table 3. With the surrogate model, the number of FP operations and last-level cache miss rate are reduced by 69.83% and 52.47% respectively. Such reductions come from the fact that the surrogate model on GPU, as an NN model, is highly optimized by the GPU vendor and able to run highly efficiently on GPU. For other applications, the number of FP operations and L2 level cache-miss rate are reduced by an average of 42.7% and 35.1% respectively. The main reasons for this observation are the reduction of model size and excellent data locality of matrix multiplication in neural network inference.

Quality. Fig. 5 also reports $HitRate$. We observe that Auto-HPCnet leads to high $HitRate$: $HitRate$ for MG, Canneal, streamcluster and AMG is 93%, 93%, 98% and 94% respectively; for the other seven applications, $HitRate$ is 100%. Note that for an application where $HitRate$ is not 100%, when running a specific input problem using the surrogate model leads to the final output failing to meet the quality requirement, the application has to restart and use the original code.

7.2 Comparison with Existing Work

In essence, Auto-HPCnet applies NN-based approximation to accelerate applications. We compare Auto-HPCnet with the following works for approximate computing.

- **ACCEPT** [56] is a tool to apply NN-based approximation to applications. ACCEPT relies on the user to manually identify the replaced code region and generate NN models without considering the impact of NN models on the final computation outcome quality.
- **Loop perforation** is a technique that selectively skips loop iterations to accelerate applications without causing significant quality degradation. Recently, loop perforation has been applied to HPC applications successfully [62]. We apply loop perforation to the 11 applications according to the recent work HPAC [62]. In particular, we use HPAC to decide how frequently the loop iterations can be skipped without causing significant quality degradation.
- **Autokeras** [26] is a tool that automatically generates NN models given training datasets. It has been reported [63] that Autokeras shows similar performance as other commercial AutoML frameworks such as Google's AutoML, H2O-AutoML, and Auto-sklearn. Autokeras cannot be used for NN-based approximation. We compare Auto-HPCnet with Autokeras in terms of the NN model effectiveness of accelerating applications.

Note that we only apply ACCEPT to Type-II applications, but not Type-I and Type-III applications, because ACCEPT heavily relies on the user to specify the NN model topology. For those applications in Type-II, ACCEPT defines their NN model topology, but not for other types of applications. To enable fair comparison, ACCEPT, loop perforation, Autokeras, and Auto-HPCnet are used to accelerate the same code regions depicted in Table 2. During the evaluation, we ensure that the final computation quality meets the pre-determined requirement (i.e., 10%)

Fig. 6 shows the application performance speedup after applying the above work and Auto-HPCnet. The speedup is calculated with respect to the execution time of the exact execution (i.e., the original execution), using Equation 2. Fig. 6 shows that Auto-HPCnet consistently performs better in all applications than the other work. Auto-HPCnet is able to find simple but effective NN architectures for small applications (Type-II) and also find more complicated NN architectures for larger applications (Type-III).

ACCEPT and the loop perforation method perform well on a few applications (i.e., Blackschole with ACCEPT, and fluidanimation and X264 with the loop perforation) with more than 2x speedup. ACCEPT and the loop perforation perform poorly on other applications with less than 2x speedup, because of the following reason: (1) ACCEPT heavily relies on the user to specify NN models, which limits its feasibility to explore a wide range of NN models. (2) The loop perforation limits its performance improvement because its approximation granularity is only at the loop's iteration level.

Autokeras achieves 12.8x and 10.89x speedup on Blackschole and fluidsimulation respectively, which is impressive. However, Autokeras cannot lead to better performance than Auto-HPCnet because of the following reasons. (1) Autokeras does not use feature reduction and does not consider model inference time, hence the NN model produced by Autokeras can have long inference time; and (2) Autokeras has problems handling sparse matrices with many zero elements, because those zero elements will cause a gradient overflow problem during the NN model training. In fact, Fig. 6 shows that using models generated by Autokeras, there is even dramatically slowdown in those applications whose inputs are high-dimensional sparse matrices (e.g., CG, FFT, MG, miniQMC and AMG). Auto-HPCnet does not have the above problems.

Effectiveness of Bayesian Optimization. Auto-HPCnet uses the Bayesian optimization to choose an NN model to replace the original code. The Bayesian optimization in Auto-HPCnet has three steps, i.e., update, generation, and evaluation. We compare the Bayesian Optimization with a traditional approach, grid search [26], which simply makes a complete search over a given subset of the topologies

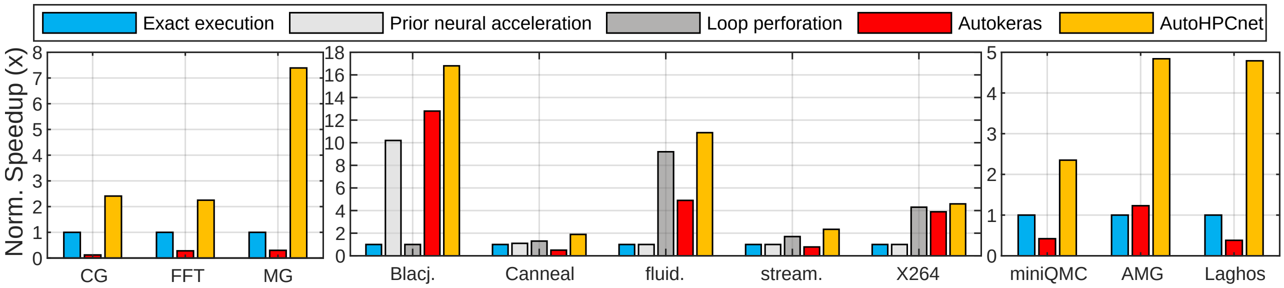


Figure 6: Performance comparison of other representative methods

space of neural network search. We use the default setting of grid search in AutoKeras.

We count the number of search steps per time unit (i.e., one hour) to reach the same model quality, as an indicator of search efficiency. For Type-I, II, and III applications, the average number of search steps per hour using the Bayesian optimization is 3.3, 6.5, and 2.1 respectively, while using the grid search, it is 1.6, 3.2, and 1.9 respectively. The Bayesian optimization has higher search efficiency, especially for Type-I and Type-II applications, because the quality-aware algorithm adopted by Bayesian optimization can effectively guide the search in the right direction compared with the grid search.

7.3 Overhead Analysis

Auto-HPCnet includes offline and online phases. We quantify the time spent on the two phases to analyze the feasibility.

Offline time. The offline phase of Auto-HPCnet includes the trace generation, the Bayesian optimization, and Autoencoder training. The execution time of the offline phase differs from one application to another. In our evaluation, the trace generation, Bayesian optimization, and Autoencoder training take 24-59 minutes, 6-13 hours, and 1.4-2.2 hours respectively. Note that the overhead of the offline phase in Auto-HPCnet, like other NN-based approximations, can be amortized, because the offline phase happens only once, and the NN-based approximation is expected to happen many times with performance benefit.

Online time. The online time includes (1) fetching input data to GPU memory, (2) encoding input data to low-dimensional features, (3) loading a pre-trained surrogate model from a file, and (4) running the surrogate model and retrieving the model output for the application. (1), (2), (3) and (4) take 21.2%, 10.1%, 1.6% and 67.1% of the whole online time on average. The online time is reported in Fig. 5 and Fig. 6.

8 OTHER RELATED WORK

Scientific Machine Learning. Scientific machine learning [64–73] aims at using machine learning methods to solve scientific and engineering problems. There are many successful cases in scientific machine learning, such as using machine learning to reproduce molecular energy surfaces [74] and simulate infrared spectra for molecular dynamics [75–77]. In [74], researchers use a DNN to approximate Discrete Fourier Transform (DFT) for Quantum chemistry (QC) simulation acceleration. Different from the existing efforts, our work does not assume any prior knowledge on application domains.

Approximate Computing. Approximate computing can be leveraged to shorten execution or save energy by trading computation accuracy. The computation approximation usually happens at a coarse granularity (e.g., the whole application or multiple functions). Approximate computing has been explored in many fields, including hardware accelerators [78–81], compiler optimization [30, 37, 82, 83], programming language designs [84–87], and runtime system designs [88–93]. Approximate computing has also been applied to many applications, such as streaming applications [29, 94, 95]. The use of surrogate models for computation, like existing efforts in approximate computing [56, 57, 60, 96, 97], does not guarantee that the application outcome is valid for all input problems. If the application outcome is not valid, the application may restart using the original code region [22, 96]. Auto-HPCnet makes best efforts to improve the accuracy of the surrogate model while guaranteeing valid application final output. Therefore, the human effort to ensure the validness of application outcomes during the practice of NN-based surrogates is reduced.

9 CONCLUSIONS

Using the surrogate models to replace computation in HPC applications is promising, but is difficult to be applied in practice, because of a series of challenges on feature acquisition, feature reduction, and NN model construction. Relying on the domain scientist to manually use those steps to apply the surrogate models is time-consuming, and fundamentally prevents the popularity of using this promising method to accelerate the performance of HPC applications. This paper aims to address the above problem and introduces an end-to-end framework (named Auto-HPCnet) that democratizes the usage of NN-based approximation in HPC applications. The design of Auto-HPCnet is driven by the observations on the major challenges of applying the surrogate models in practice. Built upon a novel hierarchical Bayesian optimization, customized autoencoder for sparse matrix and NN model construction, and compiler-assisted feature acquisition, Auto-HPCnet can effectively ease and accelerate the exploring process of applying the surrogate models to HPC applications.

ACKNOWLEDGEMENT

The research described in this paper was supported through DOE Office of Advanced Scientific Computing Research as a part of the Center for Artificial Intelligence focused Architectures and Algorithms

(ARIAA) under project 74756 Co-design of Reconfigurable Accelerators for Sparse, Irregular Computations Underlying Machine Learning and Graph Analysis. Pacific Northwest National Laboratory is a multiprogram national laboratory operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830. This work was supported in part by Oracle Cloud credits and related resources provided by the Oracle for Research program. This work was also partially supported by U.S. National Science Foundation (OAC 2104116) and the Chameleon Cloud.

REFERENCES

[1] Markus Reichstein, Gustau Camps-Valls, Bjorn Stevens, Martin Jung, Joachim Denzler, Nuno Carvalhais, et al. Deep learning and process understanding for data-driven earth system science. *Nature*, 566(7743):195–204, 2019.

[2] Vladimir M Krasnopolsky and Michael S Fox-Rabinovitz. Complex hybrid models combining deterministic and machine learning components for numerical climate modeling and weather prediction. *Neural Networks*, 19(2):122–134, 2006.

[3] Paul A O'Gorman and John G Dwyer. Using machine learning to parameterize moist convection: Potential for modeling of climate, climate change, and extreme events. *Journal of Advances in Modeling Earth Systems*, 10(10):2548–2563, 2018.

[4] Arvind T Mohan and Datta V Gaitonde. A deep learning based approach to reduced order modeling for turbulent flow control using lstm neural networks. *arXiv preprint arXiv:1804.09269*, 2018.

[5] Dunhui Xiao, CE Heaney, L Mottet, F Fang, W Lin, IM Navon, Y Guo, OK Matar, AG Robins, and CC Pain. A reduced order model for turbulent flows in the urban environment using machine learning. *Building and Environment*, 148:323–337, 2019.

[6] Mathis Bode, Michael Gauding, Zeyu Lian, Dominik Denker, Marco Davidovic, Konstantin Kleinheinz, Jenia Jitsev, and Heinz Pitsch. Using physics-informed enhanced super-resolution generative adversarial networks for subfilter modeling in turbulent reactive flows. *Proceedings of the Combustion Institute*, 38(2):2617–2625, 2021.

[7] Akinori Tanaka, Akio Tomiya, and Kōji Hashimoto. *Deep Learning and Physics*. Springer, 2021.

[8] Rahul Rai and Chandan K Sahu. Driven by data or derived through physics? a review of hybrid physics guided machine learning techniques with cyber-physical system (cps) focus. *IEEE Access*, 8:71050–71073, 2020.

[9] Paul Raccuglia, Katherine C Elbert, Philip DF Adler, Casey Falk, Malia B Wenny, Aurelio Mollo, Matthias Zeller, Sorelle A Friedler, Joshua Schrier, and Alexander J Norquist. Machine-learning-assisted materials discovery using failed experiments. *Nature*, 533(7601):73–76, 2016.

[10] Gabriel R Schleder, Antonio CM Padilha, Carlos Mera Acosta, Marcio Costa, and Adalberto Fazzio. From dft to machine learning: recent approaches to materials science–a review. *Journal of Physics: Materials*, 2(3):032001, 2019.

[11] Ruijin Cang, Hechao Li, Hope Yao, Yang Jiao, and Yi Ren. Improving direct physical properties prediction of heterogeneous materials from imaging data via convolutional neural network and a morphology-aware generative model. *Computational Materials Science*, 150:212–221, 2018.

[12] Peter Sadowski, David Fooshee, Niranjan Subrahmanya, and Pierre Baldi. Synergies between quantum mechanics and machine learning in reaction prediction. *Journal of chemical information and modeling*, 56(11):2125–2128, 2016.

[13] Kristof Schütt, Pieter-Jan Kindermans, Huziel Enoc Sauceda Felix, Stefan Chmiela, Alexandre Tkatchenko, and Klaus-Robert Müller. Schnet: A continuous-filter convolutional neural network for modeling quantum interactions. *Advances in neural information processing systems*, 30, 2017.

[14] Suraj Pawar, Omer San, Burak Aksoylu, Adil Rasheed, and Trond Kvamsdal. Physics guided machine learning using simplified theories. *Physics of Fluids*, 33(1):011701, 2021.

[15] Alireza Yazdani, Lu Lu, Maziar Raissi, and George Em Karniadakis. Systems biology informed deep learning for inferring parameters and hidden dynamics. *PLoS computational biology*, 16(11):e1007575, 2020.

[16] Mark Alber, Adrian Buganza Tepole, William R Cannon, Suvranu De, Salvador Dura-Bernal, Krishna Garikipati, George Karniadakis, William W Lytton, Paris Perdikaris, Linda Petzold, et al. Integrating machine learning and multiscale modeling—perspectives, challenges, and opportunities in the biological, biomedical, and behavioral sciences. *NPJ digital medicine*, 2(1):1–11, 2019.

[17] Charuleka Varadharajan, Vipin Kumar, Jared Willard, Jacob Zwart, Jeff Sadler, Helen Weierbach, Talita Perciano, Juliane Mueller, Valerie Hendrix, and Danielle Christianson. Using machine learning to develop a predictive understanding of the impacts of extreme water cycle perturbations on river water quality. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States); Univ . . . , 2021.

[18] Tianfang Xu and Albert J Valocchi. Data-driven methods to improve baseflow prediction of a regional groundwater model. *Computers & Geosciences*, 85:124–136, 2015.

[19] Wenqian Dong, Jie Liu, Zhen Xie, and Dong Li. Adaptive neural network-based approximation to accelerate eulerian fluid simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–22, 2019.

[20] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[21] Denghui Lu, Han Wang, Mohan Chen, Lin Lin, Roberto Car, E Weinan, Weile Jia, and Linfeng Zhang. 86 pflops deep potential molecular dynamics simulation of 100 million atoms with ab initio accuracy. *Computer Physics Communications*, 259:107624, 2021.

[22] Wenqian Dong, Zhen Xie, Gokcen Kestor, and Dong Li. Smart-pgsim: Using neural network to accelerate ac-opf power grid simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[23] Dalvan Griebler, Junior Loff, Gabriele Mencagli, Marco Danelutto, and Luiz Gustavo Fernandes. Efficient nas benchmark kernels with c++ parallel programming. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 733–740. IEEE, 2018.

[24] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019.

[25] Ekaba Bisong. Google automl: Cloud vision. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 581–598. Springer, 2019.

[26] Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. Keras tuner. *Retrieved May*, 21:2020, 2019.

[27] Rudi Helfenstein and Jonas Koko. Parallel preconditioned conjugate gradient algorithm on gpu. *Journal of Computational and Applied Mathematics*, 236(15):3584–3590, 2012.

[28] Aiichiro Nakano. Parallel multilevel preconditioned conjugate-gradient approach to variable-charge molecular dynamics. *Computer Physics Communications*, 104(1-3):59–69, 1997.

[29] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-Based Approximation for Data Parallel Applications. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[30] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[31] Marc De Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. *ACM SIGARCH Computer Architecture News*, 38(3):497–508, 2010.

[32] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, 2012.

[33] Stefano Cherubina Giovanni Agostaa Imane Lasrib, Erven Rohoub, and Olivier Sentieysb. Implications of reduced-precision computations in hpc: Performance, energy and error. *Parallel Computing is Everywhere*, 32:297, 2018.

[34] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3424–3433. JMLR. org, 2017.

[35] Kurt Binder, Jürgen Horbach, Walter Kob, Wolfgang Paul, and Fathollah Varnik. Molecular dynamics simulations. *Journal of Physics: Condensed Matter*, 16(5):S429, 2004.

[36] Andrew C Lorenc. Analysis methods for numerical weather prediction. *Quarterly Journal of the Royal Meteorological Society*, 112(474):1177–1194, 1986.

[37] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134, 2011.

[38] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. 2009.

[39] Branko Grünbaum and Geoffrey Colin Shephard. *Tilings and patterns*. Courier Dover Publications, 1987.

[40] Philip J Davis. *Interpolation and approximation*. Courier Corporation, 1975.

[41] Frank Noé, Gianni De Fabritiis, and Cecilia Clementi. Machine learning for protein folding and dynamics. *Current opinion in structural biology*, 60:77–84, 2020.

[42] Chiyu Max Jiang, Soheil Esmaeilzadeh, Kamyar Azizzadenesheli, Karthik Kashinath, Mustafa Mustafa, Hamdi A Tchelepi, Philip Marcus, Anima Anandkumar, et al. Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework. *arXiv preprint arXiv:2005.01463*, 2020.

[43] Simone Campanoni, Giovanni Agosta, Stefano Crespi Reghizzi, and Andrea Di Biagio. A highly flexible, parallel virtual machine: design and experience of ildjit. *Software: Practice and Experience*, 40(2):177–207, 2010.

[44] L. Guo, D. Li, I. Laguna, and M. Schulz. FlipTracker: Understanding Natural Error Resilience in HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018.

[45] Mark Probst, Andreas Krall, and Bernhard Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 35–44. IEEE, 2002.

[46] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(2):175–204, 1994.

[47] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[48] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[49] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. Boa: The bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*, volume 1, pages 525–532. Citeseer, 1999.

[50] Ian Dewancker, Michael McCourt, and Scott Clark. Bayesian optimization for machine learning: A practical guidebook. *arXiv preprint arXiv:1612.04858*, 2016.

[51] Sam Partee, Matthew Ellis, Alessandro Rigazzi, Scott Bachman, Gustavo Marques, Andrew Shao, and Benjamin Robbins. Using machine learning at scale in hpc simulations with smartsim: An application to ocean climate modeling. *arXiv preprint arXiv:2104.09355*, 2021.

[52] Raj Patel. Data+ education. redis is a cache or more? Technical report, EasyChair, 2021.

[53] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[54] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[55] Christian Bienia. *Benchmarking modern multiprocessors*. Princeton University, 2011.

[56] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01*, 1(2), 2015.

[57] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE, 2012.

[58] Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. *ACM SIGARCH Computer Architecture News*, 44(3):66–77, 2016.

[59] Paul Messina. The exascale computing project. *Computing in Science & Engineering*, 19(3):63–67, 2017.

[60] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. Snnap: Approximate computing on programmable socs via neural acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–614. IEEE, 2015.

[61] SIAM Journal on Scientific Computing. Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *Notices of the AMS*, 37.5:S602–S626, 2015.

[62] Konstantinos Parasyris, Giorgis Georgakoudis, Harshitha Menon, James Diffenderfer, Ignacio Laguna, Daniel Osei-Kuffuor, and Markus Schordan. Hpac: evaluating approximate computing techniques on hpc openmp applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[63] Anh Truong, Austin Walters, Jeremy Goodsitt, Keegan Hines, C Bayan Bruss, and Reza Farivar. Towards automated machine learning: Evaluation and comparison of automl approaches and tools. In *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI)*, pages 1471–1479. IEEE, 2019.

[64] Uri Shaham, Alexander Cloninger, and Ronald R Coifman. Provable approximation properties for deep neural networks. *Applied and Computational Harmonic Analysis*, 44(3):537–557, 2018.

[65] Sungmoon Jung and Jamshid Ghaboussi. Neural network constitutive model for rate-dependent materials. *Computers & Structures*, 84(15-16):955–963, 2006.

[66] Baris Sen and Suresh Menon. Representation of chemical kinetics by artificial neural networks for large eddy simulations. In *43rd Aiaa/Asme/Sae/Asee Joint Propulsion Conference & Exhibit*, page 5635, 2007.

[67] Erik Marchi, Fabio Vesperini, Florian Eyben, Stefano Squartini, and Björn Schuller. A novel approach for automatic acoustic novelty detection using a denoising autoencoder with bidirectional lstm neural networks. In *Proceedings 40th IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2015*, pages 5–pages, 2015.

[68] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807:155–166, 2016.

[69] Giuseppe Carleo and Matthias Troyer. Solving the quantum many-body problem with artificial neural networks. *Science*, 355(6325):602–606, 2017.

[70] Yunjie Liu, Evan Racah, Joaquin Correa, Amir Khosrowshahi, David Lavers, Kenneth Kunkel, Michael Wehner, William Collins, et al. Application of deep convolutional neural networks for detecting extreme weather in climate datasets. *arXiv preprint arXiv:1605.01156*, 2016.

[71] Guanghui Liang and K Chandrashekhara. Neural network based constitutive model for elastomeric foams. *Engineering structures*, 30(7):2002–2011, 2008.

[72] Yuelin Shen, K Chandrashekhara, WF Breig, and LR Oliver. Finite element analysis of v-ribbed belts using neural network based hyperelastic material model. *International Journal of Non-Linear Mechanics*, 40(6):875–890, 2005.

[73] Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on gpu. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 426–440, 2021.

[74] J. S. Smith, O. Isayev, and A. E. Roitberg. Ani-1: an extensible neural network potential with dft accuracy at force field computational cost. 2017.

[75] Michael Gastegger, Jörg Behlerb, and Philipp Marquetand. Machine Learning Molecular Dynamics for the Simulation of Infrared Spectra . *Chemical Science*, pages 6695–7270, 2017.

[76] Wenqian Dong, Letian Kang, Zhe Quan, Kenli Li, Keqin Li, Ziyu Hao, and Xiang-Hui Xie. Implementing molecular dynamics simulation on sunway taihulight system. In *2016 IEEE 18th international conference on high performance computing and communications; IEEE 14th international conference on smart city; IEEE 2nd international conference on data science and systems (HPCC/SmartCity/DSS)*, pages 443–450. IEEE, 2016.

[77] Wenqian Dong, Kenli Li, Letian Kang, Zhe Quan, and Keqin Li. Implementing molecular dynamics simulation on the sunway taihulight system with heterogeneous many-core processors. *Concurrency and Computation: Practice and Experience*, 30(16):e4468, 2018.

[78] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[79] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *ETS*, 2013.

[80] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *ACM TOCS*, 2014.

[81] Zeshi Liu, Zhen Xie, Wenqian Dong, Mengting Yuan, Haihang You, and Dong Li. A heterogeneous processing-in-memory approach to accelerate quantum chemistry simulation. *Parallel Computing*, 116:103017, 2023.

[82] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *ICSE*, 2010.

[83] Adrian Sampson, Andre Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01*, 2015.

[84] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Design Automation Conference*, 2010.

[85] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *PInternational Conference on Supercomputing*, 2006.

[86] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, 2013.

[87] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[88] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. Helix-up: Relaxing program semantics to unleash parallelization. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[89] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS*, 2015.

[90] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling large-scale gnn training with batch-level graph partitioning. 2023.

[91] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[92] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. Fauce: Fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proceedings of the VLDB Endowment*, 14(11):1950–1963, 2021.

[93] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. Md-hm: memoization-based molecular dynamics simulations on big memory system. In

Proceedings of the ACM International Conference on Supercomputing, pages 215–226, 2021.

[94] Daya S Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. Rumba: An online quality management system for approximate computing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, 2015.

[95] Dhanya R. Krishnan, Do Le Quoc, Pramod Bhatotia, Christof Fetzer, and Rodrigo Rodrigues. Incapprox: A data analytics system for incremental approximate computing. In *WWW*, 2016.

[96] Xin Sui, Andrew Lenharth, Donald S Fussell, and Keshav Pingali. Proactive control of approximate programs. *ACM SIGPLAN Notices*, 51(4):607–621, 2016.

[97] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 161–176, 2016.