MLCD: Machine Learning-based Code Version and Device Selection for Heterogeneous Systems

Kaiwen Cao, Hanchen Ye, Yihan Pang, *Student Member, IEEE* and Deming Chen, *Fellow, IEEE*

***Abstract*—Heterogeneous systems with hardware accelerators are increasingly common, and various optimized implementations/algorithms exist for computation kernels. However, no single best combination of *code version and device* (C&D) can outperform others across all input cases, demanding a method to select the best C&D pair based on input. We present machine learning-based code version and device selection method, named *MLCD*, that uses input data characteristics to select the best C&D pair dynamically. We also apply active learning to reduce the number of samples needed to construct the model. Demonstrated on two different CPU-GPU systems, MLCD achieves near-optimal speed-up regardless of which systems tested. Concretely, reporting results from system one with mid-end hardwares, it achieves 99.9%, 95.6%, 99.9%, and 98.6% of the optimal acceleration attainable through the ideal choice of C&D pairs in General Matrix Multiply, PageRank, N-body Simulation, and K-Motif Counting, respectively. MLCD achieves a speed-up of 2.57×, 1.58×, 2.68×, and 1.09× compared to baselines without MLCD. Additionally, MLCD handles end-to-end applications, achieving up to 10% and 46% speed-up over GPU-only and CPU-only solutions with Graph Neural Networks. Furthermore, it achieves $7.28\times$ average speed-up in execution latency over the state-of-the-art approach and determines suitable code versions for unseen input $10^8 - 10^{10}\times$ faster.**

***Index Terms*—Machine Learning, Heterogeneous Systems, Input Data-aware, Performance Optimizations, Active Learning**

I. Introduction

The end of Dennard scaling has driven the adoption of accelerators, such as GPUs, in data centers and edge devices. For example, Amazon's P4 or G4 cloud machines come with high-performance discrete GPUs and multicore CPUs, while Nvidia Jetson Orin Nano is a low-power platform with a mobile GPU and six ARM CPUs. The increase in the heterogeneity of hardware is accompanied by a growing spectrum of algorithms and optimizations for common computational kernels. For instance, convolution kernels, such as those found in Deep Neural Networks (DNN), can be realized by the Fast Fourier Transform (FFT) [1], Winograd [2], and General Matrix Multiply (GEMM) [3]. Thus, to achieve the best performance, we must select both the software implementation and hardware device, which often use different techniques to optimize the performance. In this work, we refer to these combinations of software implementations and hardware devices as *code version and device* (C&D) pairs.

C&D pair selection problem. In this heterogeneous landscape, we observe: (1) there are multiple C&D pairs for a given kernel, (2) the performance of a C&D pair may vary based on the input data's characteristics, and (3) the input data's characteristics vary based on the domain of the application and when the kernel is invoked [4]. For a given application, a single kernel may be used multiple times, with drastically different data in each call. Thus, we find that there is no single best choice for all cases. For example, GPUs excel at processing GEMM, when matrix size and shape enable efficient tiling and distribution across GPU. However, extremely sparse matrices with irregular memory access patterns can offset the benefits of the GPU's parallelism [5]. Thus, a multicore CPU may outperform GPUs depending on the size, shape, and sparsity of input data. Similarly, different algorithms on the same device demonstrate varying characteristics. For instance, for GPU, nvGRAPH's [6] PageRank implementation is generally faster for larger graphs, but Gunrock's [7] implementation can perform better for graphs with a larger diameter. In N-body Simulation, as we find in our evaluation later, the CPU implementations generally perform better with a smaller number of bodies, but the number of timesteps also influences which CPU implementation is the best. In K-Motif Counting, the FlexMiner [8] implementation outperforms Sandslash [9] implementation for lower graph density (the ratio between the number of edges over vertices); meanwhile, when the size of the graph becomes extremely large the G^2Miner [10] implementation on GPU shows advantages over the previous implementations. In Graph Neural Networks, as shown later, with different characteristics of the graph, layers running on the CPU or GPU may outperform each other. Based on the observations over such a diverse set of workloads, intelligent C&D pair selection is critical for attaining high performance.

Existing approaches. Accurately predicting the best C&D pair is a challenging problem. Previous attempts fall into two categories, online and offline, and both have their limitations. Online profiling [11], [12] operates during application runtime without needing prior knowledge. Because the overhead of transferring data from the CPU to the GPU on the same die is negligible, the latency of online profiling of different devices' performance and workloads re-scheduling will not impact end-to-end performance too much. Therefore, it is often constrained to integrated CPU-GPU systems with shared memory, exhibiting limited applicability to systems with discrete accelerators due to high data transfer overhead. In contrast, offline

K. Cao, H. Ye and D. Chen are with the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, Urbana, IL, 61801, USA (e-mail: {kaiwenc2, hanchen8, dchen}@illinois.edu).

Y. Pang is with the Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, IL, 61801, USA (e-mail: yihanp2@illinois.edu).

profiling [13]–[15] is applicable to a wider range of hardware. However, it fails to consider runtime data characteristics like graph diameters, leading to decisions based on static input features or predefined thresholds. Recent works [16], [17] combine online and offline approaches. They utilize the offline method when input characteristics are previously profiled. However, they fall back to the online method for unseen input characteristics, which often results in suboptimal performance due to the lack of prior knowledge.

Our proposal. In this work, we introduce the concept of *dynamic data-aware optimization* and provide an effective methodology for it. We propose a machine learning-based method, named *MLCD*, to select the best C&D pair during runtime, demonstrated on a CPU-GPU system. It is aware of both the static and runtime input data characteristics, such as the sparsity of matrices or the diameter of graphs, allowing it to make more accurate predictions on a wider range of workloads. We also extend MLCD with a dynamic programming-based algorithm to handle complicated end-to-end applications like neural networks.

Our contributions are as follows:

- Analyze the relationship between the input characteristics and performance of C&D pairs using a diverse number of different workload cases, demonstrating the performance gain of data-aware C&D pairs selection.
- Propose MLCD, an efficient decision tree-based machine learning algorithm, to guide data-aware C&D selection, achieving performances that are 99.9%, 95.6%, 99.9%, and 98.6% of the optimal ideal speed-up for GEMM, PageRank, N-body Simulation and K-Motif Counting kernels. MLCD achieves a speed-up of 2.57×, 1.58×, 2.68×, and 1.09× for these kernels, respectively, compared to the baselines without MLCD.
- Enhance MLCD with active learning that significantly reduces the number of samples needed to construct the decision tree model without compromising the selection quality. The active learning-augmented method requires at least $4.86\times$, $7.38\times$, $9.11\times$ and $1.06\times$ fewer samples for training in GEMM, PageRank, N-body Simulation, and K-motif Counting while maintaining near-optimal decision-making capabilities that achieve 99.9%, 95.1%, 99.9%, and 94.9% of the ideal speed-up, respectively.
- Extend MLCD with a dynamic programming-based algorithm to handle end-to-end applications in $O(N)$ time complexity, where N is the number of kernels in the application. MLCD offers up to 10% and 46% speed-up over GPU-only and CPU-only solutions (running the entire application on the GPU or CPU) on Graph Neural Networks (GNN).
- Achieve $7.28\times$ average reduction in GEMM execution latency compared to TVM [16], [18] across diverse input shapes. More importantly, MLCD is several orders of magnitude faster in determining the most suitable code version compared to TVM: $10^4\times$ faster for seen data inputs and $10^8 - 10^{10}\times$ faster for unseen ones.

II. Motivation

GEMM is one of the most important computation kernels in a wide range of application domains, including scientific computing [4], machine learning [19], and graph processing [20]. In this section, we use a GEMM between matrix A (size of $M \times K$) and B (size of $K \times N$) as a driving example to illustrate the difficulty of the C&D pair selection problem from two perspectives: the quality of results and long search time.

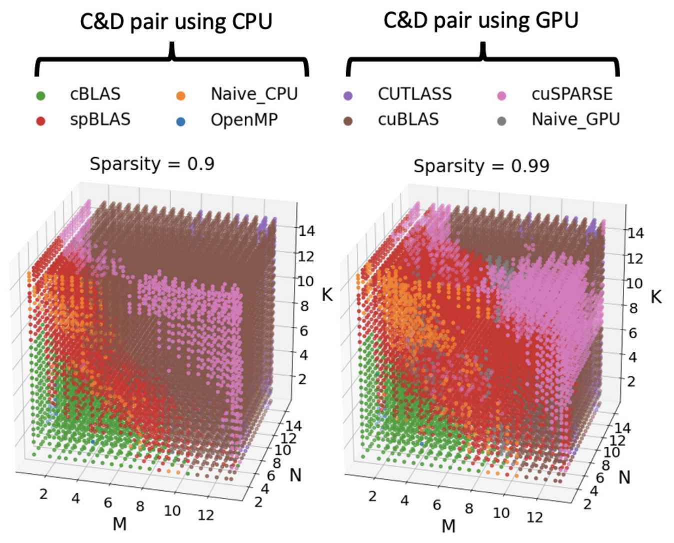


Fig. 1. Best C&D pair for GEMM in the input feature space. All axes are in log2 scale, with axis labels representing the corresponding powers of two.

A. Challenges in the quality of results

We first construct a set of synthetic GEMM samples with a wide range of input shapes and sparsity. For each sample, we measure the latency of all available C&D pairs and record the best one, as detailed in Section IV. Figure 1 shows the distribution of the best C&D pair in the input feature space. The three axes represent the M, N, and K dimensions in the log scale, and the color of a point represents the best C&D pair in terms of latency for the specific input. As the legend in Figure 1 shows, the orange, blue, green, and red points are for the four CPU code versions as detailed in Section IV-A. The gray, brown, pink, and purple colors are for the four GPU versions as detailed in Section IV-A.

The decision boundaries between the colored regions are extremely complex and non-linear. While the GPU implementations are faster for larger input in general, the boundary between CPU and GPU versions is not a flat hyper-plane but a complex curved surface, reflecting the intricate interplay of code optimizations and system characteristics. More importantly, the comparison of sub-figures with varying sparsity levels shows a shift and transformation in the boundary, underscoring the importance of implicit input characteristics in determining the C&D pair.

To further demonstrate the benefits of data-aware C&D pair selection, we measure the average performance gain from our method compared to the baseline without MLCD. The baseline without MLCD is where only a single C&D pair is available and is used across all input characteristics. We choose the best-performing baseline for comparison with our MLCD method. For example, for the GEMM benchmark, there

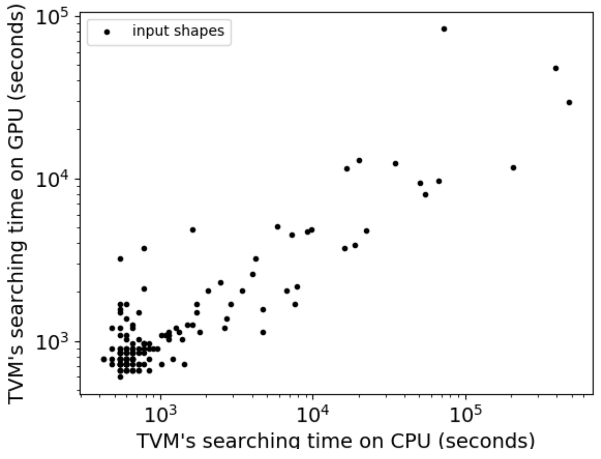


Fig. 2. Scatter plot of TVM online search time targeting CPU and GPU as the backend. All axes are in log scale.

are eight baselines in total (four on CPU and four on GPU as listed in Table II). We choose *Nvidia cuBLAS* to compare with our MLCD method as it achieves the best performance among all eight baselines. In the experimental results, we demonstrate the advantages of MLCD: for GEMM, PageRank, N-body simulation, and K-Motif Counting the speed-up is 2.57×, 1.58×, 2.68×, and 1.09×, respectively, compared to the baseline without MLCD.

B. Challenges in long search time

The state-of-the-art method, Ansor-integrated TVM [16], [17], uses offline profiling to match the existing searching logs with different input shapes, providing the best code version for a targeted device. If it has not seen the input shape before, it falls back to the time-consuming online searching algorithm, tuning the code with various optimizations for that specific input. In the remainder of this paper, we use *TVM* to refer to Ansor-integrated TVM.

In Figure 2, we profile and visualize online search time of TVM for the GEMM benchmark, with each dot representing a distinct input shape defined by a combination of M, N, and K. For clarity, the input shapes legend indicates that each dot corresponds to one input, with detailed inputs provided later in Table V. Covering 143 different shapes, the figure shows the search times for CPU and GPU code versions on the x-axis and y-axis, respectively, as TVM can only target either CPU or GPU for GEMM execution. The x-axis and y-axis are in log scale and the time unit is in seconds. In the beginning, TVM doesn't see any input shapes, relying on its online search algorithm to determine the best code version. Most input shapes in TVM require a search time in the order of 10^3 seconds, but some take an order of 10^4-10^5 seconds (up to 5×10^5 seconds which is $\sim$6 days). Considering the diversity of input data in real-world scenarios, searching for the best C&D pair for every possible combination becomes infeasible due to the long search time. On the other hand, MLCD is able to handle unseen input by running model inferences quickly within μs, as demonstrated later.

III. C&D Pair Selection Algorithm

A. Decision tree algorithm

1) Motivation and design: Decision tree (DT) is a popular machine learning algorithm that uses a tree structure to represent the learned model. There are two main reasons for using DT for C&D pair selection.

- **Learning capability.** A major advantage of DT is its ability to learn complicated non-linear relationships using a relatively small dataset. As shown earlier, the true decision boundary between different colored regions in Figure 1 for the best C&D pairs is extremely complex and cannot be captured by linear models without complicated mathematical transformations.
- **Inference complexity and scalability.** The inference of DT is simpler than deep learning algorithms. The time complexity of the DT inference is $O(log(S))$, where S is the number of training samples. Therefore, the latency of inference increases moderately with training samples, thus supporting fast inferences with larger training datasets for more complex kernels.

In training, samples are recursively split into subgroups based on input feature values until a stopping condition is met, forming a decision tree's internal and leaf nodes. In inference, a test sample navigates the tree based on these splits and predicts the sample at the leaf node according to its label. The output of DT is the predicted C&D pair. The splits aim to minimize label variation within each subgroup and are chosen to maximize the purity gain of the sample groups. Several DT-based ensemble algorithms, such as random forest and gradient-boosting trees, which have higher inference latency and larger data footprint, being undesirable for our purpose.

2) Data-aware decision tree feature engineering: The proposed DT model does not solely rely on the function parameters or problem sizes. Instead, we extract more implicit features of the input and use them for more accurate predictions. We use GEMM, PageRank, N-body Simulation, and K-Motif Counting as examples to illustrate how feature engineering works for various computing patterns.

- **GEMM.** We use the M, N, K dimensions of two input matrices, and sparsity as the input features.
- **PageRank.** It is an algorithm that quantifies the importance of nodes within a network, leveraging the principle that links (edges) from significant neighbor nodes contribute more to a node's score. It takes a graph of nodes and edges as input and outputs a numerical value representing each node's relative importance or ranking within the network. So, we construct a more complex set of features suitable for graph inputs. The features include the number of nodes, number of edges, total size (nodes + edges), density, effective diameter, and distribution of input and output degrees. The effective diameter of the graph is measured by randomly sampling 10 paths and picking the 90 percentile. The distribution of input and output degrees are represented using the 25th, 50th, and 75th percentile of the histogram of the in/out degree. For relatively small sample dataset sizes, we regularize the model by limiting the minimum number of samples in a leaf node as 12 based on empirical testing. This regularization

Algorithm 1: Active learning algorithm

```
Input:
Initial training set size: $S_{init}$
Target training set size: $S_{target}$
Number of samples added in each iteration: $S_{batch}$
Threshold of margin: $U_{th}$
Random sample generator: $G$(# of samples to generate)
Output:
Trained decision tree model: $M_{final}$
Training set: $T$
1  $T = G(S_{init})$
2  Train the guiding model $M_{guide}$ with dataset T
3  while size(T) $\leq S_{target}$ do
4      $T_{batch} = \emptyset$  // $T_{batch}$ is a batch of samples to be added
         into the training set in this iteration
5      while size($T_{batch}$) $< S_{batch}$ do
6          Sample $P = G(1)$
7          if $P \notin T$ and Margin($M_{guide}$, $P$) $< U_{th}$ then
8              $T_{batch} = T_{batch} \cup P$
9          end
10     end
11     $T = T \cup T_{batch}$
12     Train the guiding model $M_{guide}$ with $T$
13 end
14 Train the final model $M_{final}$ with $T$
```

strategy prevents the splitting of leaf nodes, thereby limiting the depth and node count in DT.

- **N-body Simulation.** An N-body simulation models the interactions and dynamics of a system of N bodies, typically under gravitational forces, where the input includes initial conditions such as positions, velocities, and masses of all bodies, and the output consists of their evolving trajectories, providing insights into the system's behavior over time. So, for input data features, besides different numbers of bodies, we include other features such as the different distribution of initial velocity, position, and mass of the body. We also include the time stepping and the total simulation iterations as input features.
- **K-Motif Counting.** It identifies and counts all unique subgraphs (motifs) of size K within a larger graph, taking the graph and the motifs as inputs, and outputting a list of motifs along with their respective counts. The value K in K-Motif Counting is one of the input features because it directly decides the matching patterns (sub-graphs) to count within the graphs. As the algorithm takes the graph as input, we also consider input features from the graph. For undirected graphs, they include the number of nodes, number of edges, effective diameter, and distribution of degree. The diameter is measured in the same manner as described in the PageRank section. The distribution of degrees is represented using the 25th, 50th, and 75th percentile of the degree histogram.

3) Active learning for efficient training: A major limitation of current methods is the high cost of profiling times for different kernel implementations, especially when large input feature spaces are involved. To address this problem, we propose an active learning strategy to reduce the number of training samples required.

Why active learning can help. Active learning [21] enhances sampling efficiency by allowing the model to determine the importance of a sample and only label important ones selectively. So, it reduces training costs by limiting the number of samples labeled. Meanwhile, it also acts as a regularization method for DT, preventing overfitting by minimizing noise sensitivity. In our case, DT partitions the input feature space where each internal node's value defines the boundaries of the partitions. These boundaries should ideally lie on true divisions between differently labeled regions, like the boundary between any two colored areas in Figure 1. Consequently, the decision boundary primarily depends on samples near the true boundary. However, as illustrated in Figure 1, many samples are distant from any boundary in the feature space, allowing them to be excluded from the training dataset by the active learning algorithm.

How we use active learning. Our active learning technique uses the uncertainty sampling strategy [21] to construct the training set, where uncertainty indicates whether a sample is close to any boundary. In our case, uncertainty is measured by the probability difference between the top two possible labels for a given sample, termed as *margin*. For example, for a sample, if the probability of guiding model's prediction for C&D pair 'X' 'Y', and 'Z' are 0.5, 0.4, and 0.1, respectively, the margin is $0.5 - 0.4 = 0.1$. We use separate models for constructing the training dataset (guiding model) and for runtime prediction (final model). For the guiding model, we employ a random forest model that consists of multiple DTs because such a sophisticated model enables a better exploration quality. Given a sample, the random forest gives the probability of every possible label (C&D pair) based on predictions made by all DTs, and the margin can be computed accordingly to guide the sampling process. Once the training dataset is constructed, we train a single DT (final model) based on the constructed dataset and use that DT for runtime prediction due to reasons discussed in Section III-A1. This enables both effective explorations during model learning and efficient prediction for inference.

Proposed algorithm. Algorithm 1 outlines our active learning algorithm. Initially, a small random set of labeled samples forms the base training set (lines 1-2). In each sampling iteration (lines 5-10), we generate unlabeled random samples and pass them into the guiding model. Samples with margins below a set threshold are selected for the current re-training batch. This process repeats until the desired number of samples is reached, and samples are then labeled based on C&D pair profiling. Next, they are added to the training set, and the model is retrained (lines 11-12). This cycle repeats until the stopping condition is met (lines 3). Finally, the refined dataset is used to train the final model for inference (line 14).

B. Dynamic programming algorithm for complicated workloads

To generalize MLCD to complicated multi-kernel workloads, we come up with two methods to solve the problem. The first is based on the shortest path algorithm and the other is based on the Dynamic Programming (DP) algorithm shown in Algorithm 2. As we will discuss later, although the first one can find the optimal solution, the DP-based approach can solve the problem optimally as well with a lower time complexity.

Algorithm 2: Optimal C&D selection algorithm for applications with multiple kernels

```
Input:
Total layers of the neural network: $N$
Optimal C&D latency list on CPU of all layers: $T_{cpu}[N]$
Optimal C&D latency list on GPU of all layers: $T_{gpu}[N]$
Data transfer latency list between layers: $T_{xfer}[N+1]$
// Assume the initial data and final results are on the CPU.
// $T_{xfer}[0]$ is the initial data transfer time from CPU to GPU.
// $T_{xfer}[N]$ is the final data transfer time from GPU to CPU.
Output:
Optimal device list (size of $N$) of all layers: $D[N]$
1  // $D_{cpu}$ and $D_{gpu}$ are optimal device lists assuming the current layer is on CPU/GPU.
2  $D_{cpu}[0] = cpu$; $D_{gpu}[0] = gpu$
3  // Optimal accumulated latency assuming the current layer is on CPU/GPU.
4  $T_{cpu}^{min} = T_{cpu}[0]$; $T_{gpu}^{min} = T_{gpu}[0] + T_{xfer}[0]$
5  for $i = 1 \to N-1$ do
6      // $D_{cpu}^{temp}$ and $D_{gpu}^{temp}$ are temporary arrays to hold values for updating.
7      // Select the optimal device list from layer 0 to layer $i-1$ assuming layer $i$ is on CPU. $[0:i-1]$ denotes values from array index 0 to index $i-1$ (inclusive).
8      $D_{cpu}^{temp}[0:i-1] = (T_{cpu}^{min} > T_{gpu}^{min} + T_{xfer}[i])\ ?\ D_{gpu}[0:i-1] : D_{cpu}[0:i-1]$
9      // Select the optimal device list from layer 0 to layer $i-1$ assuming layer $i$ is on GPU.
10     $D_{gpu}^{temp}[0:i-1] = (T_{gpu}^{min} > T_{cpu}^{min} + T_{xfer}[i])\ ?\ D_{cpu}[0:i-1] : D_{gpu}[0:i-1]$
11     // Update the device lists.
12     $D_{cpu}[0:i-1] = D_{cpu}^{temp}[0:i-1]$; $D_{cpu}[i] = cpu$
13     $D_{gpu}[0:i-1] = D_{gpu}^{temp}[0:i-1]$; $D_{gpu}[i] = gpu$
14     // Add computation and data transfer latency of layer $i$ to the optimal latency.
15     $T_{cpu}^{min} = T_{cpu}[i] + min(T_{cpu}^{min}, T_{gpu}^{min} + T_{xfer}[i])$
16     $T_{gpu}^{min} = T_{gpu}[i] + min(T_{gpu}^{min}, T_{cpu}^{min} + T_{xfer}[i])$
17 end
18 // Return the optimal device list based on the final optimal latency.
19 $D[0:N-1] = (T_{cpu}^{min} > T_{gpu}^{min} + T_{xfer}[N])\ ?\ D_{gpu}[0:N-1] : D_{cpu}[0:N-1]$
```

Two algorithms can select the best C&D pair for each kernel in end-to-end applications like Graph Neural Networks (GNN), where each layer functions as a separate kernel [22]. When kernel fusion is possible, such as fusing GEMM with ReLU, we treat two kernels as a single fused kernel in our algorithm, because the purpose of fusion is to improve the performance as a single kernel instead of executing two kernels one by one. To handle the most common scenario, without loss of generality, our algorithm assumes the initial data is on the CPU, and the result is transferred back to the CPU at the end.

1) Algorithm input and output: We consider two possible cases for whether we have seen the input data to the workloads: (1) the workload executed on the input before, and we have that input characteristics information. (2) the input is unseen, and we do not know the input characteristics information. For the seen input characteristics, MLCD will call the DT model inference to obtain the C&D pair decision and latency for each individual kernel in an offline manner before running the actual workload. So, the time spent on this can be ignored. As we show later in Section V, even for unseen input characteristics, DT inferences occur during runtime in a negligible latency compared to the kernel execution latency (less than 0.1% of one layer's execution latency). Meanwhile, the data transfer latency between CPU and GPU is analytically determined based on data size and hardware setup; we verify latency analysis correctness with micro-benchmarking. Based on these initial inputs, we design two algorithms that can identify the optimal devices of each kernel in the workload.

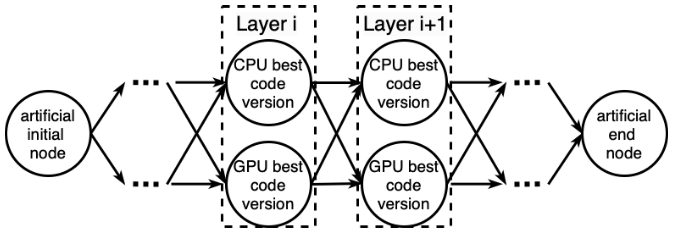


Fig. 3. Graph constructed for shortest path algorithm.

2) Shortest path-based algorithm: We first describe how to construct the graph for the shortest path algorithm based on the input mentioned above. As shown in Figure 3, the graph needs to model each layer i of the workload with two nodes to represent the CPU and GPU best code version at each layer. Nodes in layer i connect to two nodes that represent the CPU and GPU best code version in the next layer $i+1$ with directed edges, totaling two outgoing edges per vertex. Edge weights represent the computation latency of the CPU or GPU best code version indicated by the source node, and may include data transfer latency if the source and destination nodes represent different devices. The shortest path algorithm can have better time complexity when it becomes a single-source shortest path algorithm because it only needs to run the shortest path algorithm once for a single node instead of running two times for two nodes representing the first layer in the constructed graph [23]. To make the graph satisfy the requirement of such an algorithm, we add one artificial initial node before the first layer and connect it to the two nodes that represent the first layer. Similarly, we add another artificial end node after the last layer and connect the last layer nodes to this artificial node. This reduces the problem to the single-source shortest path problem. The graph is a non-negative weighted directed acyclic graph, and the algorithm's complexity is $O(E+V)$ [23], where E and V are the number of edges and vertices in the constructed graph, respectively. Given that N is the number of kernels in the workload, for the single source shortest path algorithm in a directed acyclic graph, the latency is proportional to $E+V$, where $V = 2N$ and $E = 2V$ (i.e. $4N$). So, the latency of the shortest path algorithm is proportional to $6N$. Next, we further reduce the time complexity by Dynamic Programming algorithm.

3) Dynamic programming algorithm: The algorithm is based on Dynamic Programming and uses internal memoization data structures for fast recursion. Algorithm 2 describes the details. Defining N as the number of kernels in the workload, we introduce internal memoization data structures as two 1D arrays of size N, D_{cpu} and D_{gpu}, to record the optimal devices in the DP running process. Correspondingly, there are also two values, T_{cpu}^{min} and T_{gpu}^{min}, which record the optimal latencies in the DP process, assuming the current layer is being handled on CPU and GPU, respectively. The DP algorithm is based on the two recurrence relationships below:

$$Find\_T_{cpu}^{min}(i) = T_{cpu}[i] + \min \begin{cases} Find\_T_{gpu}^{min}(i-1) + T_{Xfer}[i], \\ Find\_T_{cpu}^{min}(i-1) \end{cases}$$

$$Find\_T_{gpu}^{min}(i) = T_{gpu}[i] + \min \begin{cases} Find\_T_{cpu}^{min}(i-1) + T_{Xfer}[i], \\ Find\_T_{gpu}^{min}(i-1) \end{cases}$$

The $Find\_T_{cpu}^{min}(i)$ and $Find\_T_{gpu}^{min}(i)$ return the minimum latency from layer 0 to layer i, assuming we finish the computation of layer i on CPU and GPU, respectively. The layer's latency is added in at each recursion call, implicitly containing all the latencies of previous layers. The base case is when layer i is the first layer, i.e. $i = 0$, where $Find\_T_{cpu}^{min}(i) = T_{cpu}[0]$ and $Find\_T_{gpu}^{min}(i) = T_{gpu}[0] + T_{xfer}[0]$.

Based on the recurrence relationship above, in Algorithm 2, lines 1-4 set up initial conditions and a memoization data structure. Subsequent lines determine the optimal devices for each layer by iterating through this structure. Concretely, line 8 and line 12 handle the case when layer i will be computed on the CPU. It first checks whichever it is optimal between: (1) layer $i-1$ is computed on CPU; (2) layer $i-1$ is computed on GPU and then transferred to CPU. Then, it can update D_{cpu} accordingly. Symmetrically, line 10 and line 13 handle the case where the layer i will be computed on the GPU and update the D_{gpu} accordingly. Note that the device list updates in lines 8, 10, and 12-13 are $O(1)$ operations because they can be implemented in pointer-swapping operations in C++. Lines 15-16 update the optimal latencies, T_{cpu}^{min} and T_{gpu}^{min}, assuming the current layer i is computed on CPU and GPU, respectively. Lines 18-19 handle the transfer time back to the CPU after the final layer is computed and we can find the optimal devices list D based on the comparison result. The whole algorithm runs with a "for" loop with N iterations, and each iteration only has $O(1)$ time complexity. So, the DP-based algorithm time complexity is $O(N)$. While detailed proof of correctness is omitted due to space, the algorithm, illustrated here with neural networks, is applicable to various applications.

4) Algorithm runtime comparison: Given that N is the number of kernels in the workload, for the single source shortest path algorithm, the latency is proportional to $6N$; for the DP-based algorithm, the latency is proportional to N, as discussed above. Meanwhile, we implement the single source shortest path algorithm and DP-based algorithm in C++. DP-based one is 9.6× faster than the shortest path algorithm. Given that runtime speed is very important, we choose the DP-based algorithm as our solution. Both algorithms optimize the end-to-end latency by selecting the best C&D pair for each kernel systematically, avoiding local optimizations that might miss the global optimum. They determine whether data transfers are needed and how they should occur (CPU to GPU or vice versa), with each kernel's optimal C&D pair decided and applied before execution. For example, if frequent transfers will result in overhead instead of benefit, the algorithm will identify this issue and automatically pick the best solution.

TABLE I
SYSTEM HARDWARE SPECIFICATIONS

System configuration	System 1	System 2
CPU Model	Intel Xeon W-2125	Intel Xeon Gold 5320
CPU Memory	16GB DDR4-2666	128GB DDR4-3200
GPU Model	NVIDIA TITAN Xp	NVIDIA H100 PCIe
GPU Global Memory	12 GB GDDR5X	80 GB HBM2e

5) Extendability to more complicated hardware configurations: When three or more devices are available, our method will include more candidate C&D pairs, and the end-to-end DP algorithm has more recursive relationships and associated memorization data structures. For instance, with two GPUs and a CPU, there would be three devices to consider. Additionally, when heterogeneous interconnects like PCIe and NVLink co-exist for a GPU, the data transfer time can be handled properly by deciding the optimal interconnect based on micro-benchmarked latencies between two interconnects. Data transfers between the CPU and GPU would still rely on PCIe, allowing us to follow our existing method. For transfers between two GPUs, the data transfer time in our DP algorithm can be determined by the lower latency between NVLink-based and PCIe-based P2P, verified using the micro-benchmark for data transfer.

IV. TRAINING AND EVALUATION METHODOLOGY

A. Data collection

Our data is collected using a system with a 4-core Xeon CPU and a Nvidia GPU connected via PCIe, detailed in "System 1" in Table I. To show the generalizability of our method, we also test on the second system described as "System 2" in Table I with a recent server-grade 26-core Xeon CPU and H100 GPU, and the data collection and evaluation methodology are the same. While demonstrated on a CPU-GPU setup, this method is adaptable to other heterogeneous systems as it does not depend on specific hardware. The only required system-specific information is the latency of the C&D pairs, which can be systematically collected.

Table II shows the CPU and GPU code versions for all benchmarks. To ensure consistency among all the benchmarks, we use single-precision floating-point representation whenever the floating-point operation is involved. For GEMM fused with ReLU, the Nvidia cuSPARSE library requires a 50% floating-point sparsity structure for executing the fused kernel, which necessitates pruning the matrix to meet this requirement. As a result, we exclude this C&D pair to avoid inconsistencies with other inputs. Note that Intel's fused kernel implementation is provided under oneDNN, as detailed in Table II. For GEMM, we collect the latency of four code versions on both CPU and GPU. Similarly, for PageRank, N-body Simulation, and K-Motif Counting, we collect the latency for three and two code versions on CPU and GPU, respectively. For all benchmarks, we stabilize CPU execution latencies by averaging over 10 iterations, excluding warm-up runs, and GPU execution latencies over 30 iterations due to observed inconsistency in samples with shorter latencies. We also measure the latency of data transfers to and from the GPU for a fair comparison.

TABLE II
CPU AND GPU CODE VERSION FOR GEMM, PAGERANK, N-BODY SIMULATION, AND K-MOTIF COUNTING

	CPU code version	GPU code version
GEMM	Naive implementation	Naive implementation
	OpenMP implementation	Nvidia cuBLAS [24]
	Intel MKL cBLAS [25]	Nvidia cuSPARSE [26]
	Intel MKL spBLAS [27]	Nvidia CUTLASS [28]
GEMM fused w/ ReLU	Naive implementation fused w/ReLU	Naive implementation fused w/ReLU
	OpenMP implementation fused w/ReLU	Nvidia cuBLAS [24]
	Intel oneDNN [29]	Nvidia CUTLASS [28]
PageRank	Ligra [30]	Gunrock [7]
	Ligra delta [30]	nvGRAPH [6]
	Galois [31]	-
N-body Simulation	Intel N-body [32]	Barnes-hut tree algorithm [33]
	Cache tiling w/o OpenMP	Nvidia N-body [34]
	Cache tiling w/ OpenMP	-
K-Motif Counting	Pangolin [35]	G^{2}Miner [10]
	FlexMiner [8]	G^{2}Miner [10] w/ counting-only pruning
	Sandslash [9]	-

The same set of inputs is used to collect the latency for all code versions of each kernel.

B. Dataset structure

We describe the structure of the datasets that we used for evaluation. For GEMM, PageRank, N-body Simulation, and K-Motif Counting, we randomly split the dataset constructed for each benchmark with 90%/10% splitting, meaning 90% and 10% of the samples used for training and testing, respectively.

1) GEMM: Input matrices are generated with the given M, N, K dimension, and the sparsity. Sparse matrices, converted from dense matrices, are stored in Compressed Sparse Row (CSR) format. To cover diverse matrix characteristics, we sweep the N, K values in the range of (2 - 31,298), with 27 points chosen approximately from the geometric series 1.5^n and M of (2 - 9,152) with 22 points. We sweep the sparsity across the following ten values (0.60, 0.70, 0.80, 0.90, 0.91, 0.93, 0.95, 0.97, 0.99, 0.995), as the study [4] shows most matrice's sparsity values in DNN or scientific computing are higher than 0.6. All these parameters give us 160,380 data points in total. As discovered in [36], given two different input sizes that have a small variation, the performance difference between the two inputs may be significant. To showcase our method's effectiveness, we follow a similar way in [36] to construct the additional dataset as follows: the M, N, and K are equal and their value varies from 50 to 31,350 in increments of 100, sweeping across the same ten different sparsity values we used before. This additional dataset has 3,140 data points. For GEMM fused with ReLU, we use the same data collection methodology to produce the additional dataset.

2) PageRank: For the PageRank kernel, we use the Kronecker random graph generator from the SNAP package [37] to generate directed, unweighted graphs, similar to those in the Graph500 challenge [38]. We generate 8,800 graphs with the number of nodes ranging from 2^{10} to 2^{17}, and limit the edge-to-node ratio to $1.5-20$ to mirror real-world graphs from the SNAP network dataset repository [39]. We also pre-process the generated graphs to ensure that they satisfy the requirements of all 5 implementations.

3) N-body Simulation: In N-body Simulation, we generate bodies with varied characteristics. The number of bodies ranges from 2 to 191,751, selected at 29 points approximately derived from the geometric series 1.5^n. Initial velocities of bodies are randomly assigned using a normal distribution with a mean of 1 and three distinct standard deviations (10^{-5}, 1, and 10^5) for each different input [40], [41]. Body positions and masses are generated following the same methodology as the generation of initial velocities. Additionally, we explore various time steps (in microseconds [34]) for the simulation (10^{-7}, 10^{-4}, and 10^{-1}), and different numbers of simulation iterations (10, 20, 40, 80, and 160). All these parameters give us 11,745 unique inputs in total.

4) K-Motif Counting: For the number of vertices in K-Motif (subgraph pattern) counting, we select the value K as either 3 or 4, because these two values are widely used in previous implementations [8]–[10], [35]. For the undirected graph, the number of nodes ranges from 2^9 to 2^{22}, increasing by the power of 2. The number of edges is determined by the edge-to-node ratio that changes from 2 to 20, with increments of 1, to mirror real-world graphs from the SNAP network dataset repository [39]. When fixing the input feature with the same number of nodes and edges, we generate 16 different graph connection topologies, giving various diameters and distribution of degree percentiles. Considering the different values of K, we have $8,512$ different input cases in total.

C. Active learning

This algorithm includes several adjustable parameters: the initial number of training samples (S_{init}), the number of samples labeled per iteration (S_{batch}), and the uncertainty threshold value of margin (U_{th}), which can be tuned to optimize the final model's accuracy. For GEMM, we set $S_{init}=1000$, $S_{batch}=1000$, and $U_{th}=0.5$. For PageRank which has fewer samples than GEMM, we set $S_{init}=200$, $S_{batch}=100$, but maintain $U_{th}=0.5$. For N-body Simulation, we apply the same hyperparameter settings as for GEMM. For K-Motif Counting, we set $S_{init}=6100$, $S_{batch}=1000$, and $U_{th}=0.5$.

D. Metrics for evaluation

1) Percentage Performance Penalty (PPP): For classification tasks, accuracy is a common metric, but it does not always reflect the performance impact in C&D pair selection. Specifically, mispredictions near the true decision boundary, where the performance difference between the correct and incorrect C&D pairs is minimal, may not significantly degrade accuracy. To better evaluate the C&D pair selection, we introduce the Percentage Performance Penalty (PPP) metric. PPP calculates the percentage difference between the performance of the predicted best C&D pair and the actual best-performing pair. It is computed by the following formula:

$$PPP = \frac{L_{i,j_{pred}} - \min_j L_{i,j}}{\min_j L_{i,j}} \times 100\%$$

where $L_{i,j}$ represents the latency of the j-th C&D pair for the i-th input sample and j_{pred} is the predicted best choice. j is ranging from one to the total number of C&D pairs for each benchmark. We further define average PPP as averaging across every sample's PPP. In our studies, *average PPP* is a key metric, more practical than accuracy, as it directly reflects performance degradation due to mispredictions. We use this metric to evaluate selections for GEMM, PageRank, N-body Simulation, and K-Motif Counting benchmarks. However, for complicated multi-kernel applications, as mentioned in Section III-B, since each single selection at each kernel alone is not enough for end-to-end optimality, we will not use average PPP for evaluation.

2) Percentage of Ideal Speed (PoIS): PoIS is another metric we create for evaluating the quality of C&D pair selection. The PoIS directly reflects how close the resulting speed is to the ideal speed (i.e. latency), which demonstrates the model's effectiveness from another angle. We define the PoIS as the ratio of the sum of the latencies by always selecting the fastest C&D pair over the sum of the latencies by using the C&D pairs selected by our model. It is computed as follows:

$$PoIS = \frac{\Sigma_{i=1}^{Y} \min_j L_{i,j}}{\Sigma_{i=1}^{Y} L_{i,j_{pred}}} \times 100\%$$

where Y is the number of samples and the rest of the notations are the same as defined above in PPP. PoIS represents the achieved speed relative to the theoretical maximum speed, with an upper limit being 100%. A higher PoIS indicates that the performance is nearing the optimal speed. Since the sum of latency is mostly contributed by the samples with larger latency, the PoIS is more sensitive to these samples, reflecting that the optimization over larger latency kernels or workloads is more critical in the real world. In contrast, PPP gives equal attention to all the samples. For this reason, we use both PPP and PoIS to show the effectiveness of the C&D pair selection method from different angles.

For multi-kernel applications, we expand the definition of PoIS to capture the potential data transfer latency. We first define *minimum latency* as the optimal latency achieved by the exhaustive search. We then calculate the PoIS as the ratio of the *minimum latency* over the latency achieved using the dynamic programming algorithm along with the decision tree model to run a workload. Note that we show the optimality of the DP algorithm and the performance gap solely comes from the mis-selection of the DT model.

Based on our experimental data, lower PPP and higher PoIS typically lead to better speed-up over the baseline. Meanwhile, DT inference latency is independent of PPP, PoIS, and accuracy changes, as computations remain the same regardless of model weight values.

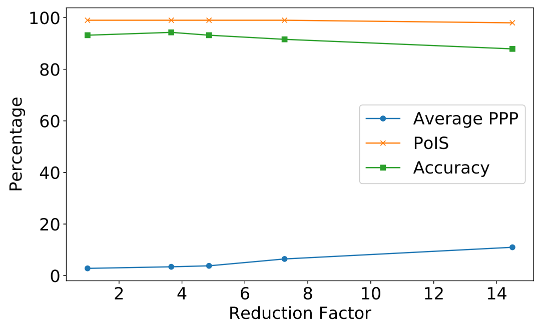


Fig. 4. Model performance comparison (Average PPP, PoIS, and Accuracy in percentages) between models built with active learning and full training dataset for GEMM. The x-axis shows the training-samples reduction factor with active learning.

V. Experimental Results

A. GEMM case study

The GEMM represents the most common and fundamental computation pattern in many workloads. In this section, we show the GEMM benchmark results, the performance with active learning applied, and the comparison with TVM [17].

1) Effectiveness of data-aware selection: We first test our data-aware method without active learning, summarized in Table III, showing the method's average PPP, PoIS, and model prediction accuracy. The ML model, which includes data transfer times between CPU and GPU, successfully predicts the best version to select in most cases with PoIS close to 100%. Robustness is further demonstrated using 10-fold cross-validation, showing low standard deviations in PPP (0.44%), PoIS (0.02%), and accuracy (0.22%). Furthermore, the latency of the model inference is 0.9 μs. Despite some mispredictions, their impact on speed-up is negligible due to their proximity to the decision boundary, where ground truth and predicted C&D pairs have similar latencies. Additionally, the average PPP is 3.81%, confirming that high PoIS scores are not due to biased sample selection. Moreover, MLCD can obtain a speed-up of 2.57 $\times$ across all input characteristics compared with Nvidia cuBLAS, a competitive baseline without MLCD. The speed-up is calculated as an average of speed-up values based on each different input characteristic. Section II-A discusses how the baseline is defined.

2) Effectiveness of active learning: Table III demonstrates the effectiveness of active learning to the GEMM kernel, achieving a 4.86× reduction in sample size with minimal loss in PoIS and a reasonable performance degradation on average PPP. The reduction factor of a value like 4.86 indicates that, compared to a reduction of 1 where active learning is not applied, we have decreased the number of required samples by a factor of 4.86. Figure 4 further depicts the performance of the active learning method for the GEMM kernel in detail, showing the reduction factor on the x-axis and the average PPP, PoIS, and prediction accuracy on the y-axis. Although a higher reduction factor like 14.5× still allows for a perfect PoIS, the average PPP is notably higher due to its sensitivity to samples with smaller latencies. As PPP is inversely proportional to ideal latency, thus the mispredicted samples with smaller ideal latency will have increased PPP sensitivity. Meanwhile, since

TABLE III
EXPERIMENTAL RESULT OF MLCD IN SYSTEM 1 FOR GEMM, PAGERANK, N-BODY SIMULATION, AND K-MOTIF COUNTING

Benchmark	Dataset Size	Active Learning	Avg. PPP (%)	PoIS (%)	Pred. Acc. (%)	Samples Reduction Factor	Speed-up over Baseline	Speed-up over TVM	Selection Time Reduction over TVM
GEMM	160,380	×	2.81	99.9	93.2	1×	2.57×	7.28×	$\sim 10^4\times$ (seen) $\sim 10^8 - 10^{10}\times$ (unseen)
		✓	3.81	99	93.2	4.86×	2.54×	7.27×	$\sim 10^4\times$ (seen) $\sim 10^8 - 10^{10}\times$ (unseen)
PageRank	8,800	×	5.65	95.6	88.1	1×	1.58×	N.A.	N.A.
		✓	6.18	95.1	86.9	7.38×	1.57×	N.A.	N.A.
N-body Simulation	11,745	×	2.06	99.9	98.13	1×	2.68×	N.A.	N.A.
		✓	0.66	99.9	97.7	9.11×	2.68×	N.A.	N.A.
K-Motif Counting	8,512	×	3.31	98.6	93.1	1×	1.09×	N.A.	N.A.
		✓	4.54	94.9	89.6	1.06×	1.05×	N.A.	N.A.

TABLE IV
EXPERIMENTAL RESULT OF MLCD IN SYSTEM 2 FOR GEMM, PAGERANK, N-BODY SIMULATION, AND K-MOTIF COUNTING

Benchmark	Dataset Size	Active Learning	Avg. PPP (%)	PoIS (%)	Pred. Acc. (%)	Samples Reduction Factor	Speed-up over Baseline	Speed-up over TVM	Selection Time Reduction over TVM
GEMM	160,380	×	3.12	99.85	91.97	1×	2.84×	6.70×	$\sim 10^4\times$ (seen) $\sim 10^8 - 10^{10}\times$ (unseen)
		✓	2.46	99.76	92.43	4.41 ×	2.78×	6.69×	$\sim 10^4\times$ (seen) $\sim 10^8 - 10^{10}\times$ (unseen)
GEMM*	163,520	×	3.14	99	95.62	1×	2.83×	6.70×	$\sim 10^4\times$ (seen) $\sim 10^8 - 10^{10}\times$ (unseen)
		✓	3.04	99.94	92.74	4.2 ×	2.78×	6.69×	$\sim 10^4\times$ (seen) $\sim 10^8 - 10^{10}\times$ (unseen)
PageRank	8,800	×	6.04	97.37	93.17	1×	2.58×	N.A.	N.A.
		✓	6.35	97.22	92.87	6.85×	2.58×	N.A.	N.A.
N-body Simulation	11,745	×	3.30	99.9	97.85	1×	2.04×	N.A.	N.A.
		✓	0.52	99.7	97.33	8.98×	2.03×	N.A.	N.A.
K-Motif Counting	8,512	×	3.26	98.71	93.27	1×	1.12×	N.A.	N.A.
		✓	4.23	95.12	90.33	1.05×	1.07×	N.A.	N.A.

* means we include additional data points to test our methodology's capability to handle small input variations for GEMM

TABLE V
INPUT DATA SHAPE COLLECTED FOR TVM'S PERFORMANCE

Input shape	Input shape space to sweep across
M	26, 130, 986, 2217
N	26, 130, 986, 2217, 9152, 26477
K	26, 130, 986, 2217, 9152, 26477

the aggregated speed-up combines the latency of all samples, as long as the model can make correct predictions on samples with larger latency, the PoIS is still close to perfect. Therefore, with the reduction of 4.86× our active learning method can maintain excellent model quality while having a significant reduction in the training sample size. Moreover, MLCD with active learning can obtain a speed-up of 2.54 × compared with Nvidia cuBLAS, a competitive baseline without MLCD.

3) Comparison with TVM: We use the most recent TVM (commit aa47018) with Ansor [17] integration as the comparison target, running on the same hardware as other experiments (Table I). Following the TVM guide [42], we set the number of trials for all TVM searches to 1,000 for a fair comparison with our method. Due to TVM's long searching time, we do not search every combination from the input characteristics space used to show our method's effectiveness in Section IV-B1. Therefore, as shown in Table V, we conduct a uniform sampling to obtain representative input shapes in a balanced way. We exclude the M=2,217, N=26,477, and K=26,477 input shape because the searching time is more than two weeks for TVM. In total, we have 143 different input shape combinations of M, N, and K.

TVM targets a single device during its search process and does not use the concept of C&D pairs. To ensure a fair comparison, we define TVM's best C&D pair as the one with the shortest latency between CPU and GPU code versions. Because TVM is agnostic to sparsity, we compare the latency of the best C&D pair that has the same M, N, and K reported by TVM against our model under all sparsity settings. Each M, N, and K combination has 10 different sparsity settings in our methodology, so we have 143 × 10=1,430 different input characteristics. All comparison results over TVM are reported based on these 1,430 points. For each input data characteristic, we calculate the speed-up ratio as the execution latency provided by the TVM solution over that provided by our MLCD method. If the ratio is above one, our ML model's selection performs better than TVM's best C&D pair, and vice versa. We count the data transfer time between CPU and GPU for fair comparison.

Performance comparison. We first compare the performance of the best C&D pair selected by MLCD (without active learning applied) and by TVM. MLCD performs better for 1,354 out of 1,430 different input cases (94.6%), demonstrating the superiority of our method. We also observe

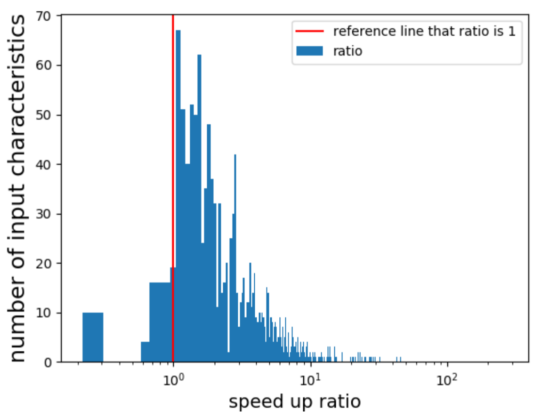


Fig. 5. Speed-up ratio distribution histogram for comparing between MLCD and TVM.

that higher sparsity matrices ($\geq$ 0.95) tend to achieve better speed-up (13.97×). As we mentioned before, TVM does not consider sparsity information, so it cannot achieve a good C&D pair for a relatively sparse matrix. Considering all 1,430 input shapes, as shown in Table III, the average speed-up over all samples is 7.28×. To further visualize the result, in Figure 5, we plot out all 1,430 different data points in a histogram. The x-axis is the speed-up ratio and the y-axis is the number of input characteristics that fall into the speed-up ratio. Applying active learning to MLCD yields similar results, MLCD performs better for 1,349 out of 1,430 different input cases (94.3%). Considering all 1,430 input shapes, as shown in Table III, the average speed-up is 7.27×. Again, we observe that higher sparsity matrices ($\geq$ 0.95) achieve better speed-up than average (13.97×). So, active learning largely maintains the quality of our ML model.

When TVM outperforms our method, the average speed-up is 1.64×. These cases fall into two categories: (1) All dimensions (M, N, K) of the GEMM operation are small (e.g., all equal to 26), where TVM performs well across all sparsity levels. (2) When the N dimension is relatively large (9152 or 26477) such that it is 10-25× larger than the M dimension and 4-25× larger than the K dimension. In case (2), the TVM shows an advantage in lower sparsity matrices ($<$ 0.97), while our method outperforms the TVM in all the other matrices. TVM can effectively find the optimal solution for GEMM kernels when matrix dimensions are small, resulting in better performance on these cases.

Searching time comparison. TVM needs a long searching time if it does not see the input shape before and needs to store all the searching logs. As mentioned in Section II, the TVM search time usually takes around 10^3s to obtain a selection solution and sometimes takes an order of $10^4 - 10^5$s. In contrast, our method only takes 0.9μs for an inference. The selection time over TVM of Table III summarizes the advantage of obtaining the best C&D pair. If we are given an unseen input shape, TVM has to first search for and then report the best schedule for this particular new input shape. On the other hand, due to MLCD's generalizability, we only need to make a single inference with the ML model to obtain the best C&D pair. For an input shape previously searched in TVM, utilizing the TVM `apply_best` API to retrieve the schedule from past searches is still 10^4× slower than MLCD. TVM trains different cost models for each GEMM input case in minutes, while MLCD only requires a single model for all input cases, which is a one-time training done in minutes.

4) Generalizability to different systems: As shown in Table IV, we test our methodology with the same set of benchmarks on a different system to demonstrate the generalizability. The PPP and PoIS differences between the two systems are minimal. With active learning, they show a similar reduction in training samples and a comparable speed-up over the baseline. These results confirm that our method is effective across different systems. We further envision that our method can be generalized to include more input characteristics such as denormalized numbers, a representation for extremely small floating-point numbers that leads to longer GEMM execution latency than the normal floating-point representation.

5) Capability of handling input variation: As noted in [36], even small variations in input sizes can lead to a significant difference in performance. To show our method's capability, we include the additional 3,140 data points described in IV-A in our experiment, of which the results are shown in row "GEMM*" in Table IV. One can observe that the model's quality and performance are similar to the result in the row "GEMM". Our method considers more internal characteristics like sparsity and focuses on relative performance when identifying the optimal C&D pair. This can help partition the input space into four or more dimensions space, which enables our model to address this challenge effectively.

6) Generality to handle kernel fusion: ReLU is a classic activation used after the GEMM execution in many modern kernel libraries. They often provide fused GEMM-ReLU kernels to reduce kernel launch latency and repeated memory access. We use GEMM fused with ReLU to demonstrate the extensibility of our method to kernel fusion scenarios. Our testing result shows PPP, PoIS, and accuracy of 1.5%, 99%, and 94.08%, respectively. The speed-up over the baseline is 1.13×. With active learning, we are able to have a reduction factor of 8.12×, and the PPP, PoIS, and accuracy values are 1.49%, 99%, and 94.11%, respectively. The speed-up over the baseline is 1.11×. These results demonstrate the generality of our method for kernel fusion.

B. PageRank case study

We demonstrate our method's generalizability to other application domains and input data with the PageRank kernel in this subsection. The input graphs of PageRank have substantially different characteristics from the input matrices of GEMM. Meanwhile, PageRank extracts information from graphs, exhibiting different computation patterns from GEMM. For fairness, all GPU versions of PageRank include CPU-GPU data transfer times in their comparisons.

We train a regularized decision tree on the PageRank dataset. As shown in Table III, it attains an average PPP of 5.65% and PoIS of 95.6%, indicating that the speed-up achieved through MLCD is near ideal. TVM, designed for machine learning [43], cannot handle the PageRank, so comparisons to TVM in Table III are marked as Not Applicable

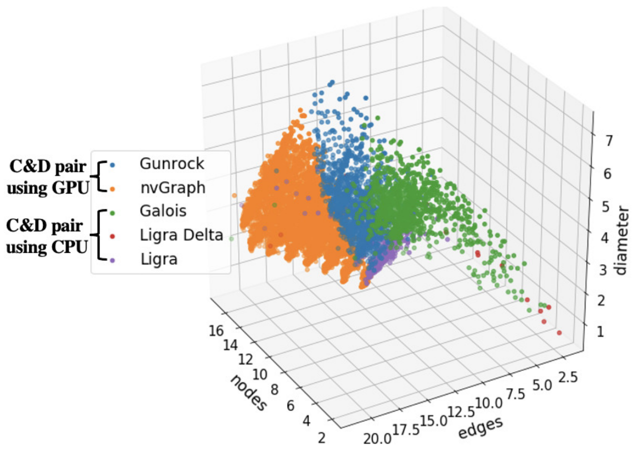


Fig. 6. Best PageRank C&D pairs in the input feature space. The values of nodes, edges, and diameter are plotted on a log2 scale.

(N.A.). We test the active learning on the PageRank kernel. As shown in Table III, we reduce the sample size by 7.38× while achieving the average PPP of 6.18% and PoIS of 95.1%, being very close to the performances without active learning. As shown in Table III and IV, in most evaluation metrics, our method performs similarly on both tested systems, with system 2 showing a marginally larger speed-up over the baseline.

Figure 6 shows the distribution of the best C&D pair of the samples in the input feature space with three feature dimensions: number of nodes, number of edges, and effective diameter. The three axes are all in the log2 scale (the actual values are 2^{exp} where exp numbers are shown for the axes) and the color of a point represents the best C&D pair in terms of latency for that sample. For graphs with approximately 2^{10} to 2^{15} edges, the CPU versions (green and purple points) can be faster depending on the diameter of the graph. Moreover, among the two GPU versions, the best C&D pair also depends on the diameter and other characteristics of the graphs. Our experiment shows that, in cases where nvGRAPH is the fastest C&D pair, the Gunrock library is around 2.51× slower on average. On the other hand, when Gunrock is the fastest, nvGRAPH is also around 2× slower on average. Additionally, as shown in Figure 6, the nvGRAPH library baseline exhibits a minimum performance gap from our C&D pair selection and we can achieve a 1.58× speed-up over nvGRAPH. These observations indicate the need for a data-aware C&D pair selection method that considers various input characteristics to obtain the best C&D pair choices.

C. N-body Simulation case study

We use the N-body Simulation as an application from the scientific computing domain to demonstrate the generalizability of our approach. Also, the input is different from matrices or graphs, which further shows the robustness of our method. For a fair comparison, we include the CPU-GPU data transfer time for all GPU versions. As shown in Table III, it achieves an average PPP of 2.06% and PoIS of 99.9%. Note that TVM is incapable of handling the N-body Simulation because it is constrained to generating code versions for machine learning [43]. So, we mark entries comparing to TVM as Not Applicable (N.A.) in Table III. We also test the active learning on the N-body Simulation. As shown in Table III, we can reduce the sample size by 9.11× while achieving the average PPP of 0.66% and PoIS of 99.9%, being very close to the performances without active learning. Notably, we achieve a 2.68× speed-up over the performance of solely selecting the Nvidia N-body library baseline. This shows that input characteristics are important to obtain the best C&D pair. In Table III and IV, the model demonstrates similar performances across the two tested systems.

D. K-Motif Counting case study

K-Motif Counting is a representative application of graph mining. Moreover, the input data consists of two parts. The first parts are undirected graphs and the second parts are sub-graph patterns (motifs) to identify. These expand the input data's variety to show our methodology's robustness and generality. For a fair comparison, we include the CPU-GPU data transfer time for all GPU versions. As shown in Table III, the PPP is 3.31% and PoIS is 98.6%. Note that we mark entries comparing to TVM as Not Applicable (N.A.) in Table III because TVM is incapable of handling the K-Motif Counting [43]. With the MLCD, we achieve a speed-up of 1.09× over the baseline that solely selects the state-of-the-art Sandslash [9] implementation across all input cases. In our experiment, we find that the GPU implementations often have a relatively large overhead in data structure transferring between devices due to the extra initializations and kernel launches to transfer sub-graph patterns (motifs) and matching results. This makes the GPU implementations less desirable to select and reduces the advantage of MLCD. We also apply the active learning technique and reduce the sample size required by 1.06× while achieving the average PPP of 4.54% and PoIS of 94.9%. Through analyzing the input feature space, we find that the boundaries are extremely complicated around GPU code versions, making it difficult for active learning to reduce the sample size effectively. In Table III and IV, the model achieves similar performances on the second tested system.

E. Graph Neural Networks end-to-end case study

Many neural networks, such as Graph Neural Networks (GNN) and Convolution Neural Networks, have diverse input characteristics, making the input shape alone not enough to guide the kernel selection process [4]. Optimizing GNN based on input data is challenging because different input graphs show vastly different characteristics (number of nodes, edges, density, etc.) from each other. To further demonstrate our method's generalizability with the Dynamic Programming (DP) algorithm on these complicated end-to-end applications, we evaluate the performance of MLCD on multiple representative GNN models. We use the popular Graph Convolution Network (GCN) [44] and GraphSAGE [19] model as the testing benchmarks that primarily rely on GEMM computation internally [22]. So, instead of developing an ad-hoc model for different GNN workloads or re-training, MLCD directly applies the DT model based on GEMM for these benchmarks with the integration of the DP algorithm.

TABLE VI
MLCD RESULT ON GRAPH NEURAL NETWORK FOR SYSTEM 1

Benchmark	Speed-up over GPU-only	Speed-up over CPU-only	PoIS
GCN	1.08×	1.46×	94.6%
GraphSAGE	1.1×	1.36×	96.2%

TABLE VII
MLCD RESULT ON GRAPH NEURAL NETWORK FOR SYSTEM 2

Benchmark	Speed-up over GPU-only	Speed-up over CPU-only	PoIS
GCN	1.08×	1.51×	94.8%
GraphSAGE	1.11×	1.39×	96.5%

1) Experiment setup: To apply our methodology to the GNN, we use DP-based Algorithm 2, which can find the optimal end-to-end solution. Inference and dynamic programming are usually done before actual workload execution, but if inputs are unknown beforehand, these processes occur at runtime. The inference latency of our C&D selection ML model is less than 1 μs, while the latency of the dynamic programming algorithm is 0.58 μs, which is trivial when compared to the overall execution latency of the workload.

We evaluate three real-world datasets (Pubmed, Cora, Citeseer [45]) and various synthetic graphs with different input characteristics. For synthetic graphs, the node numbers range from 3,000 to 24,000 (in 3,000-node increments) and the edge-to-node ratio varies from 1 to 8, which leads to different numbers of edges as well. To better match common GNN structures, the number of layers varies from 4 to 9 layers with an increment of one and hidden dimensions from 2^2 to 2^7, increasing in powers of two. Additionally, node feature dimensions range from 1,000 to 5,000, in increments of 1,000.

2) Result: When we measure the latency, we include data transfer and kernel execution. As shown in Table VI, considering the decision tree model inference latency and search algorithm latency at the beginning, on GCN, we achieve the average speed-up of 1.08× and 1.47× over GPU-only and CPU-only (running the entire application on GPU or CPU) solutions, respectively. On GraphSAGE, we achieve the average speed-up of 1.1× and 1.36× over GPU-only and CPU-only solutions, respectively. In terms of PoIS, on average, we are only 5.4% and 3.8% away from the ideal speed-up for GCN and GraphSAGE, respectively. In Table VI and VII, the model shows similar performances across the two systems tested.

To demonstrate the variation in the selected transfer points, we plot a histogram showing the distribution of the layer where selected transfer occurs in GCN. For a clear representation of graph diversity, we fix the node count at 3,000 and vary the number of edges by sweeping different edge-to-node ratios. In terms of network structure, we maintain a constant hidden dimension of 4 and a total of 9 layers, while altering the node feature dimension. Figure 7 presents a histogram of the distribution. The x-axis denotes the layer number where the selected transfer occurs, while the y-axis reflects the percentage of GNN networks corresponding to each transfer point on the x-axis. All transfers happen when it runs several initial layers on the GPU and then transfers the rest of the

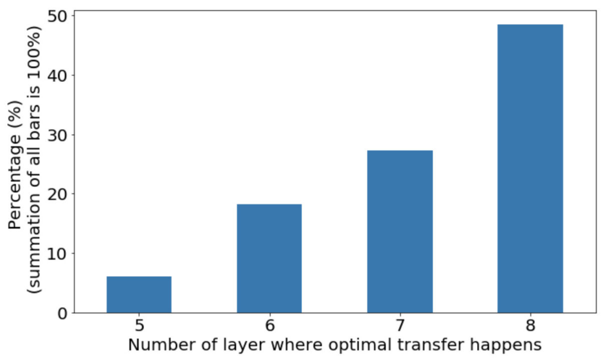


Fig. 7. Histogram of the selected transfer of GCN.

computation to the CPU. While the overall trend suggests that more selected transfer happens when placing relatively fewer layers on the CPU, no single and straightforward rule for decision-making.

VI. Related Works

A. Online approaches

Previous research, such as Farooqui et al. [12] and Kaleem et al. [11], has explored dynamic device and algorithm selection primarily for integrated CPU-GPU systems. Farooqui et al. developed an online, instrumentation-based method that selects the code version by running an instrumented kernel. However, this method omits critical features like graph size and diameter. Kaleem et al. focused on workload partitioning between CPU and GPU without considering multiple algorithms and versions. In contrast, our method accommodates a wider range of input characteristics and is applicable to both integrated and non-integrated systems, addressing the impracticality of online profiling on systems with high data transfer overhead between discrete devices.

B. Offline approaches

Phothilimthana et al. [13] presented an offline method that uses empirical data to decide on algorithmic and device choices during compilation. However, it only considers input size and simple cutoffs, making it inadequate for varied input characteristics. Thus, it cannot adapt to inputs with varying characteristics. Muralidharan et al. [14] used an SVM model for tuning code variants only based on the input size, and it only targets Nvidia GPU's irregular memory access kernel such as SpMV, limiting its applicability. Luk et al. [15] developed an offline method to profile different workload partitioning schemes for runtime decision-making but did not account for multiple code versions. In contrast, our method makes decisions based on implicit input characteristics, and hence, can adapt to a wider range of input cases while supporting multiple code versions and algorithms.

C. Combined approaches

TVM [16] provides a state-of-the-art method to searching for the best code version (schedule) for a targeted device.

For unseen input characteristics, it uses the schedule explorer and the ML-based cost model together, iterating until the TVM reaches the number of iterations set by users, logging the best code version from each iteration, and ultimately selecting the top-performing version as the final output. For seen input characteristics, it reports the best code version by directly query searching logs. However, users must provide templates to guide the search process, requiring detailed hardware knowledge (e.g., tile size, loop reordering). Ansor [17] enhances TVM by automating the search algorithm, improving efficiency, and eliminating the need for manual template guidance. Compounding the issue is that TVM with Ansor integration's offline approach is limited to previously encountered inputs; for new, diverse input characteristics, it falls back to more time-consuming online searching for good code versions.

VII. Conclusion

In this work, we presented a dynamic data-aware **m**achine **l**earning-based **c**ode version and **d**evice selection method (MLCD) for heterogeneous systems. We evaluated MLCD on a diverse set of benchmarking workloads and demonstrated its effectiveness and generalizability across different domains. When applied to GEMM, PageRank, N-body Simulation, and K-Motif Counting, MLCD achieved near-optimal decision-making capabilities, being 99.9%, 95.6%, 99.9%, and 98.6% of the ideal speed-up. Compared to the baseline without MLCD, MLCD had a speed-up of 2.57×, 1.58×, 2.68×, and 1.09 × respectively, for these workloads. With active learning, the sample size was reduced by 4.86×, 7.38×, 9.91× and 1.06 ×, while maintaining near-optimal decision-making capabilities that achieve 99%, 95.1%, 99.9%, and 94.9% of the ideal speed-up for these workloads, respectively. Additionally, we extended MLCD to end-to-end applications with a series of kernels, showing up to 10% and 46% speed improvements over GPU-only and CPU-only solutions, respectively, in Graph Neural Network (GNN) benchmarks. Our method, when compared against TVM, a state-of-the-art approach, achieved a 7.28× speed-up in execution latency for representative GEMM input shapes, and was $10^8 - 10^{10}\times$ faster in finding suitable code versions for unseen inputs. For previously seen inputs, it was still $10^4\times$ quicker. We further demonstrate the effectiveness of MLCD on the second hardware platform and achieve similar model accuracy and performance numbers to show our model's generalization capability. Although demonstrated in a CPU-GPU system, this methodology is envisioned as a universal approach, potentially applicable to other domains and heterogeneous systems, and adaptable for optimizing metrics like power or energy consumption.

VIII. Appendix

Here we present the pseudo code for naive GEMM for CPU in Algorithm 3 and GPU in Algorithm 4.

Algorithm 3: CPU Naive Matrix Multiplication

```
Input: Matrices $A$, $B$; Dimensions $M$, $N$, $K$
Output: Matrix $C$ after computation
1 for $i \leftarrow 0$ to $M-1$ do
2     for $j \leftarrow 0$ to $N-1$ do
3         $sum \leftarrow 0$;
4         for $k \leftarrow 0$ to $K-1$ do
5             $sum \leftarrow sum + A[i \cdot K + k] \cdot B[k \cdot N + j]$;
6         $C[i \cdot N + j] \leftarrow sum$;
```

Algorithm 4: GPU Naive Matrix Multiplication

```
Input: Matrices $A$, $B$; Dimensions $M$, $N$, $K$
Output: Matrix $C$ after computation
1 Kernel Function:
  NaiveGEMMGPUKernel$(A, B, C, M, N, K)$;
2     $row \leftarrow$ blockIdx.y · blockDim.y + threadIdx.y;
3     $col \leftarrow$ blockIdx.x · blockDim.x + threadIdx.x;
4     $sum \leftarrow 0$;
5     if $col < N$ and $row < M$ then
6         for $i \leftarrow 0$ to $K-1$ do
7             $sum \leftarrow sum + A[i \cdot M + row] \cdot B[col \cdot K + i]$;
8         $C[col \cdot M + row] \leftarrow sum$;
9 Host Function: NaiveGEMMGPU$(A, B, C, M, N, K)$;
10    $block \leftarrow 32$;
11    $dimGrid \leftarrow (\lceil N/block \rceil, \lceil M/block \rceil, 1)$;
12    $dimBlock \leftarrow (block, block, 1)$;
13    NaiveGEMMGPUKernel
        ⟨⟨⟨dimGrid, dimBlock⟩⟩⟩$(A, B, C, M, N, K)$;
```

References

[1] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *arXiv preprint arXiv:1312.5851*, 2013.

[2] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4013–4021.

[3] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 19–24.

[4] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[5] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "Adaptive sparse matrix-matrix multiplication on the gpu," in *Proceedings of the 24th symposium on principles and practice of parallel programming*, 2019, pp. 68–81.

[6] "nvGraph," https://docs.nvidia.com/cuda/nvgraph/index.html.

[7] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: Gpu graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017.

[8] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 581–594.

[9] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: a two-level framework for efficient graph pattern mining," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 378–391.

[10] X. Chen *et al.*, "Efficient and scalable graph pattern mining on {GPUs}," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 857–877.

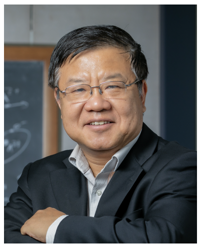
[11] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated gpus," in *Proceedings*

of the 23rd international conference on Parallel architectures and compilation, 2014, pp. 151–162.
[12] N. Farooqui, I. Roy, Y. Chen, V. Talwar, and K. Schwan, "Accelerating graph applications on integrated gpu platforms via instrumentation-driven optimizations," in *CF '16*, 2016, pp. 19–28.
[13] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable performance on heterogeneous architectures," in *ASPLOS '13*, 2013, pp. 431–444.
[14] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 501–512.
[15] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 45–55.
[16] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.
[17] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.
[18] "Introducing tvm auto-scheduler (a.k.a. ansor)," https://tvm.apache.org/2021/03/03/intro-auto-scheduler.
[19] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.
[20] C. W. Coley, W. Jin, L. Rogers, T. F. Jamison, T. S. Jaakkola, W. H. Green, R. Barzilay, and K. F. Jensen, "A graph-convolutional neural network model for the prediction of chemical reactivity," *Chemical science*, vol. 10, no. 2, pp. 370–377, 2019.
[21] B. Settles, "Active learning literature survey," Tech. Rep., 2010.
[22] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–12.
[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.
[24] "cublas," https://docs.nvidia.com/cuda/cublas/index.html.
[25] "Intel BLAS Level 3 Routines," https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-2/blas-level-3-routines.htmlGUID-73761EEF-B0DC-4F41-9B7E-C8D6202866BF.
[26] "cusparse," https://docs.nvidia.com/cuda/cusparse/index.html.
[27] "Intel Sparse BLAS Level 2 and Level 3 Routines," https://www.intel.com/content/www/us/en/docs/onemkl /developer-reference-c/2024-1/sparse-blas-level-2-and-level-3-routines-001.html.
[28] "Cutlass," https://nvidia.github.io/cutlass/.
[29] "Intel oneAPI Deep Neural Network Library," https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn .html.
[30] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013.
[31] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, 2011.
[32] "Intel oneAPI Base Toolkit," https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html.
[33] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," in *GPU computing Gems Emerald edition*. Elsevier, 2011, pp. 75–92.
[34] "Fast N-Body Simulation with CUDA," https://developer.nvidia.com/gpugems/gpugems3/ part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda.
[35] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An efficient and flexible graph mining system on cpu and gpu," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1190–1205, 2020.
[36] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2176–2190, 2018.
[37] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.
[38] "Benchmark specification — graph 500," https://graph500.org/?page_id=12.
[39] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data.
[40] N. Perraudin, A. Srivastava, A. Lucchi, T. Kacprzak, T. Hofmann, and A. Réfrégier, "Cosmological n-body simulations: a challenge for scalable generative models," *Computational Astrophysics and Cosmology*, vol. 6, pp. 1–17, 2019.
[41] A. Jenkins, "A new way of setting the phases for cosmological multiscale gaussian initial conditions," *Monthly Notices of the Royal Astronomical Society*, vol. 434, no. 3, pp. 2094–2120, 2013.
[42] "Optimizing operators with auto-scheduling," https://tvm.apache.org/docs/tutorial/auto_scheduler_matmul_x86.html.
[43] "Apache TVM," https://tvm.apache.org.
[44] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
[45] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.

Kaiwen Cao is a PhD candidate in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign (UIUC), Urbana, IL, 61801, USA. His research interests include heterogeneous computing. He received his dual Bachelor's degree from Zhejiang University and the University of Illinois at Urbana-Champaign.

Hanchen Ye is a PhD candidate in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign (UIUC). His research focuses on hardware compilers (ScaleHLS/XLS/CIRCT), AI compilers (StreamTensor/HIDA), and AI accelerators (CHARM/HybridDNN/DNNExplorer). He received his Bachelor's and Master's degrees from Fudan University, respectively.

Yihan Pang is a Ph.D. candidate in the Department of Computer Science at the University of Illinois at Urbana-Champaign (UIUC). His research focuses on the design and development of efficient Extended Reality (XR) systems. He holds both Bachelor's and Master's degrees in Computer Engineering from Virginia Tech.


Deming Chen is the Abel Bliss Professor in the Grainger College of Engineering at the University of Illinois Urbana-Champaign. His research interests include hybrid cloud systems, machine learning and AI, security and confidential computing, reconfigurable and heterogeneous computing, and system-level design methodologies. He has published over 290 research papers, received 10 Best Paper Awards and an ACM/SIGDA TCFPGA Hall-of-Fame Paper Award, and delivered more than 160 invited talks. His work has had a significant impact, with open-source solutions adopted by industry, such as FCUDA, DNNBuilder, CSRNet, SkyNet, ScaleHLS, and Medusa. He is an IEEE Fellow, an ACM Distinguished Speaker, and the former Editor-in-Chief of ACM Transactions on Reconfigurable Technology and Systems (TRETS). He serves as the Illinois Director of the IBM-Illinois Discovery Accelerator Institute and the Director of the AMD-Xilinx Center of Excellence. He received his Ph.D. in Computer Science from UCLA in 2005.