XUnified: A Framework for Guiding Optimal Use of GPU Unified Memory

HAILU XU1, PEI-HUNG LIN2, MURALI EMANI3, LITING HU4, AND CHUNHUA LIAO2

1Department of Computer Engineering and Computer Science, California State University, Long Beach, CA 90802, USA
2Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
3Argonne National Laboratory, Lemont, IL 60439, USA
4Department of Computer Science and Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064, USA

Corresponding author: Hailu Xu (hailu.xu@csulb.edu)

This work was supported in part by the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and Contract LLNL-LDRD 18-ERD-006; in part by the National Science Foundation under Grant NSF-SPX-2202859, Grant NSF-CAREER-2205677, and Grant NSF-OAC-2212256; and in part by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357 and Contract LLNL-CONF-798297.

ABSTRACT Unified Memory is a single memory address space that is accessible by any processor (GPUs or CPUs) in a system. NVIDIA's unified memory creates a pool of managed memory on top of physically separated CPU and GPU memories. NVIDIA's unified memory automatically migrates page-level data on-demand, so programmers can quickly develop CUDA codes on heterogeneous machines. However, it is extremely difficult for programmers to decide when and how to efficiently use NVIDIA's unified memory because (1) users are usually unaware of which unified memory hint (e.g., ReadMostly, PreferredLocation, AccessedBy) should be used for a data object in the application, and (2) it is tedious and error-prone to do manual memory management (i.e., manual code modifications) for various applications with different data objects or inputs. We present XUnified, an advice controller which combines the offline training with the online adaptation to guide the optimal use of unified memory and discrete memory for various applications at runtime. The offline phase uses profiler-generated metrics to train a machine learning model, which is used to predict optimal memory advice choice and it then applies this advice to applications at runtime. We evaluate XUnified on NVIDIA Volta GPUs with a set of heterogeneous computing benchmarks. Results show that it achieves 94.0% prediction accuracy in correctly identifying the optimal memory advice choice with a maximal 34.3% reduction in kernel execution time.

INDEX TERMS GPU, unified memory, performance prediction.

I. INTRODUCTION

Unified Memory is a single memory address space that is transparently accessible by any processor in a system [3]. It means that the GPUs and CPUs are working over the same memory. Graphics resources, such as textures, images and geometry data, can be shared between the CPU and GPU efficiently, with no overhead, as there’s no need to copy data across a PCIe bus. NVIDIA's Unified Memory (UM) [28] was designed and implemented in current Nvidia GPUs after CUDA 6.0 [6]. It is widely used in supercomputers with heterogeneous accelerators such as GPUs. For example, six of the top ten fastest supercomputers employ NVIDIA GPUs according to the latest Top500 list [2].

NVIDIA's UM creates a pool of managed memory on top of physically separated CPU and GPU memories. It can automatically migrate page-level data on-demand, so programmers can quickly develop CUDA codes on heterogeneous machines in the HPC cluster. NVIDIA provides a cudaMemAdvise1 API to advise the UM driver about the usage pattern of data objects (e.g. dynamically allocated arrays [4]). Different hints (such as ReadMostly,

1NVIDIA CUDA Runtime API. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html
One application usually has multiple data objects involved in computations and users are usually unaware of which memory hint should be used for a certain data object. It typically requires tedious and error-prone manual efforts for memory optimization.

PreferredLocation, AccessedBy [28]) can be specified in this API by programmers to help improve the performance. For example, if some data is specified as ReadMostly, the UM driver may duplicate the data on both CPU and GPU memory to enable fast local memory access from either CPU or GPU. If some data is specified for a target device, it should use the PreferredLocation with the memory of the targeted device.

However, it is extremely difficult for users/programmers to decide when and how to efficiently use NVIDIA’s unified memory. As shown from Figure 1, (1) one application usually has multiple data objects involved in computations and users are usually unaware of which unified memory hint should be used for a certain data object. For example, CFD [10] has three arrays and four kernels. It is difficult to know which hint should be used for a particular array. If one array will be accessed by a specified CPU or GPU device, it is better to use the AccessedByCPU or AccessedByGPU option on the specific device. However, such information cannot be directly identified by the users. (2) It is tedious and error-prone to do manual memory management for various applications with difference data objects or inputs. Manually identifying and managing data for GPUs can be extremely tedious and error-prone since scientific applications often use dynamically allocated memory objects with nested pointers to many memory objects. Manual operations, such as inserting explicit memory copy calls and deep copying pointers in traditional discrete memories (non-unified memory) [11], [12], [15], [16], may incur long execution latency and low memory performance at runtime. This impact will be greatly aggravated for heterogeneous supercomputers that run applications at large scale.

Many previous studies explore the implementation of different hints to use unified memory. However, there is a lack of an automatic and accurate tool for deciding the use of hints for unified memory in various applications. Previous studies exhibit the following limitations: (1) they usually require extra manual operations on specific code or program [8], [17], [20], [30], such as using rule-based data access patterns to guide the memory selection for a Tesla GPU [17] or using a compiler-based approach to generate kernel variants for exploiting memory characteristics [32]; and (2) they lead to conjectural use of unified memory [19], [20], [23], which impacts the runtime performance and incurs longer execution times, resulting in sub-optimal resource utilization and degraded performance.

We propose XUnified, an advice controller, to automatically decide whether to use unified or discrete memory and how to select the proper unified memory advice in GPUs (e.g., ReadMostly, PreferredLocation, AccessedBy [28]). XUnified consists of two phases: an offline learning phase and an online inference phase. The offline learning phase involves building a classifier via supervised learning. The online inference involves determining the proper memory advice at runtime for a CUDA program and applying the advice through runtime code transformation. By combining the offline learning and online inference, XUnified can automatically guide the optimal use of unified memory and support runtime adaption of CUDA programs without tedious manual efforts.

We make the following contributions in this paper.

- First, we study the hybrid use of both discrete and unified memory APIs on GPUs and explore the impact of different memory advice hints in using unified memory CUDA APIs. To the best of our knowledge, our study is the first-of-its-kind work to study this unique problem.

- Second, we propose XUnified, an advice controller that automatically guides the optimal use of unified memory. Leveraging profiling metrics of both kernels and data objects, XUnified not only determines whether to use unified or discrete memory, but also predicts the optimal memory advice hint, rather than conjectural or speculative advice for that specific application.

- Third, we design code transformation to enable runtime adaptation of CUDA programs to exploit online inference decisions. We incorporate runtime code features to provide fine-grain control of data objects’ use of discrete and unified memory in GPUs.

- Fourth, we present the evaluation results of XUnified on two Nvidia GPU machines and demonstrate a high prediction accuracy of 94.0% with minuscule overhead. We also demonstrate the efficiency of the model used in XUnified with (i) data from diverse hardware and (ii) unseen application that is not included in the training set. The source code of XUnified has been made publicly available.²

²https://github.com/AndrewXu22/optimal_unified_memory
TABLE 1. Code variants in gaussian and CFD with different memory advice set for data objects in each variant.

| Variants | Description |
|----------|-------------|
| 1        | Baseline using GPU’s memory as discrete memory for all memory objects |
| 2        | Using default unified memory (UM) for all objects |
| 3        | UM is enabled with the ReadMostly advice |
| 4        | UM is enabled with the PreferredLocation option set to be on an GPU |
| 5        | UM is enabled with the AccessedBy option set to be on an GPU |
| 6        | UM is enabled with the PreferredLocation option set to be on a CPU |
| 7        | UM is enabled with the AccessedBy option set to be CPU |

While our technique is being discussed and evaluated within the context of heterogeneous CPU and GPU machines, it is also generally applicable to most accelerator systems in which programs running on the accelerators can access the host’s main memory either by direct-mapped PCIe memory space or a paging mechanism.

The rest of this paper is organized as follows: Section II details the motivation of the work and the background of unified memory and various advice choices. Section III describes the details of our design and methodology. Section IV shows the evaluation results. We discuss the related works in Section V and conclude this work in Section VI.

II. MOTIVATION AND BACKGROUND

In this section, we describe a motivating example and the background of unified memory and various advice choices.

A. A MOTIVATING EXAMPLE

Inappropriate use of unified memory may incur significant overhead for applications. The following example illustrates the variances of performances when using different hints of unified memory in the gaussian and CFD benchmark from Rodinia [10]. Table 1 lists the code variants for the two benchmarks. The major data objects in the kernel function refer to two matrices $a$, $m$ and one array $b$ in gaussian. We apply different memory usage choices to these objects and get multiple combinations of code variants.

Code variant 1 is the baseline version that uses default discrete memory for all the data objects in the benchmarks. Code variants 2 to 7 use the unified memory for the data objects and different memory advice options. We further specify CPU or GPU as the device for AccessedBy and PreferredLocation. The input data for gaussian is a $1024 \times 1024$ matrix. The input size of CFD is 97k elements of a 3-D grid cells.

Figure 2 displays the speedup of the gaussian and CFD code with different code variants using different memory allocations and advice choices. The red dotted line implies the kernel execution time using discrete memory (speedup as 1) which is the baseline kernel execution. Other bars show the range of speedup with different advice variants. We can observe that the gaussian benchmark achieves a speedup of $1.12 \times$ in the kernel execution (from 0.138s to 0.123s) when matrix $a$ is given the variant 5 with the input matrix is $1024 \times 1024$. On the contrary, a $117 \times$ performance degradation in the kernel execution (from 0.138s to 16.14s) is observed for the variant 3. For the CFD benchmark, the kernel can achieve a speedup of $1.51 \times$ in the execution (from 49.8ms to 32.9ms) when using the variant 6.

It shows that the memory choices for data objects have a significant impact on the kernel execution time. Appropriate use of memory choices in CFD, such as code variant 6, can greatly reduce the execution latency of specific application. And inappropriate use of memory in gaussian, such as ReadMostly (variant 3), can incur a huge latency at the runtime. This is because in the gaussian benchmark, ReadMostly should be beneficial for the data that is mostly for reading and only occasionally written to the unified memory. In this case, the matrix is not suitable for being read-only in unified memory.

Unfortunately, it is extremely challenging for programmers to decide when and how to efficiently use UM for various kinds of applications. For a given data object, there is a wide range of choices including managing it with the traditional discrete memory API, the unified memory API without advice, and the unified memory API combined with various memory hints. Such information cannot be directly obtained by users or programmers unless they manually examine the execution details of each data object. This typically requires huge manual efforts in the advice selection and easily leads to conjectural advice.

B. BACKGROUND

Programmers often encounter multiple choices to manage their data on GPU memory. Nvidia’s CUDA traditionally exposes GPU device memory as a discrete memory space from CPU memory space. Programmers are responsible for using a set of memory API functions to explicitly manage the entire life cycle of data objects stored in GPU memory, including allocation, de-allocation, data copying, etc. Since CUDA 6.0, Nvidia has introduced Unified Memory (UM) with a new set of API functions. The idea of UM is to
present developers with a single memory space unifying both CPU and GPU memories [3]. CUDA uses a UM driver to automatically migrate data between CPU and GPU memories at runtime. As a result, UM significantly improves the productivity of GPU programming. Both traditional memory APIs and unified memory APIs can be used in a single CUDA program.

To enable better performance of UM, CUDA allows developers to give the UM driver additional advice on managing a given GPU memory range via an API function named `cudaMemAdvise(const void *, size_t, enum cudaMemAdvise, int)` [28]. The first two parameters of this function accept a pointer to a memory range with a specified size. The memory range should be allocated via `cudaMallocManaged` or declared via __managed__ variables. When applicable, the third parameter sets the advice for the memory range. The last parameter indicates the associated device’s id, which can indicate either a CPU or GPU device when applicable. The details of these four kinds of advice are as follows:

- **Default** [5]: This represents the default on-demand page migration to the accessing processor by using the first-touch policy.
- **cudaMemAdviseSetReadMostly** [5]: This advice is used for the data which is mostly going to be read from and only occasionally written to. The UM driver may create read-only copies of the data in a processor’s memory when that processor accesses it. If this region encounters any write requests, then only the write occurred page will be valid and other copies will be invalid.
- **cudaMemAdviseSetPreferredLocation** [5]: Once a target device is specified, this device’s memory can be set as the preferred location for the allocated data. The host memory can also be specified as the preferred location. Setting the preferred location does not cause data to migrate to that location immediately. The policy only guides what will happen when a fault occurs on the specified memory region: if data is already in the preferred location, the faulting processor will try to directly establish a mapping to the region without causing page migration. Otherwise, the data will be migrated to the processor accessing it if the data is not in the preferred location or if a direct mapping cannot be established.
- **cudaMemAdviseSetAccessedBy** [5]: This advice implies that the data will be accessed by a specified CPU or GPU device. It has no impact on the data location and will not cause data migration. It only causes the data to be always mapped in the specified processor’s page tables, when applicable. The mapping will be accordingly updated if the data is migrated somehow. This advice is useful to indicate that avoiding faults is important for some data, especially when the data is accessed by a GPU within a system containing multiple GPUs with peer-to-peer access enabled.

The effect of `cudaMemAdvise` can be reverted with one of the following options: `UnsetReadMostly`, `UnsetPreferredLocation`, and `UnsetAccessedBy`. As a result, programmers can flexibly change the advice hints to the UM driver for a same memory object throughout the execution of a program.

### III. DESIGN

XUnified is an advice controller that automatically determines optimal memory advice for GPU programs, specifically in data allocation and manipulation using discrete and unified memory in GPUs. In this section, we introduce the workflow in building XUnified and present implementation details.

The workflow in XUnified consists of two phases: offline learning and online inference, as shown in Figure 3. The offline learning phase collects the runtime metrics from training applications using various memory choices, labels the data with best choices leading to optimal performance, and generates a classifier using the training dataset. In the online inference phase, runtime metrics from unseen applications are collected and feed the trained classifier model to find best memory usage choices.

![FIGURE 3. The workflow of XUnified. It consists of two phases: offline learning and online inference. The offline learning phase collects the runtime metrics from training applications using various memory choices, labels the data with best choices leading to optimal performance, and generates a classifier using the training dataset. In the online inference phase, runtime metrics from unseen applications are collected and feed the trained classifier model to find best memory usage choices.](image-url)

The offline learning phase consists of two phases: offline learning and online inference, as shown in Figure 4. We next describe the design of training configurations that help to capture metrics corresponding to diverse memory usage variants
The classes to choose the output from. These 7 choices object in it. There are 7 possible choices, which form with all possible memory usage choices for each data

Each kernel in the training samples should be implemented compose the training set is the foundation of the entire work.

Gathering sufficient runtime metrics from training samples to model via supervised learning.

- composed with extracted metrics and training a classifier from benchmarks, constructing training dataset from features composed with extracted metrics and training a classifier model via supervised learning.

1) TRAINING SAMPLES AND METRICS COLLECTION

Gathering sufficient runtime metrics from training samples to compose the training set is the foundation of the entire work. Each kernel in the training samples should be implemented with all possible memory usage choices for each data object in it. There are 7 possible choices, which form the classes to choose the output from. These 7 choices are `DiscreteMemMalloc`, `UnifiedMemoryMalloc` default, `ReadMostly`, `PreferredLocation` for GPU, `PreferredLocation` for CPU, `AcceptedBy` for GPU, and `AcceptedBy` for CPU. For a given kernel, the input data may have different sizes, ranging from small to large options. Ideally, at least three different input data input sizes should be used to generate enough training data. Another factor is that each kernel may reference multiple data objects. We select the top three largest data objects to collect training metrics since larger data objects usually have a bigger impact on the kernel’s performance.

Most existing CUDA programs do not adopt the latest unified memory advice at runtime and are unaware of whether to use the unified memory or not. Therefore, additional guidance needs to be provided to these programs. We first implement code variants of selected benchmark programs, using different GPU memory usage choices. The Rodinia benchmark [10] is selected to implement different memory usage choices for specific arrays or data structures. In order to use the latest unified memory, XUnified implements the `XUnified_mem alloc()` API to switch between discrete and unified memory versions of CUDA memory allocation. Figure 5 presents an example of using unified memory and different `cudaMemAdvise()` settings for two arrays (a and b) in the `gaussian` benchmark. We define different combinations of advice as `advOptionA` and `advOptionB` and use `cudaMemAdvise()` function to set the specific advice for the given data object. The combinations of advice create many versions that are used for the experiments. Essentially, the input program is staged to use a set of parameters to allow different choices of CUDA memory usages for different memory objects. Later, XUnified’s runtime inference phase can dynamically set the values of the parameters to use a given choice, based on the output of the trained model.

We also design source-to-source transformation, based on the ROSE compiler framework [1], to generate the variants and bypass the staged input code. Directives are given by users to identify the data object and specify the desired memory advice. The output variant will be generated with the memory allocation and memory advice APIs inserted.

All code variants are generated by using seven memory advice options for each kernel in the benchmark. Several different input data sets are used also to capture the impact of input sizes. For example, we specify memory advice to three arrays in the CFD benchmark and three input sizes. The total number of execution variants becomes 1029 (3 × 7 × 7 × 7). We examine the performance of modified benchmarks as described above and record the detailed execution time. The advice that yielded minimal execution latency for each data object is denoted as its specific `label` in the training set. For example, if `gaussian` benchmark achieves minimal execution latency when using `cudaMemAdviseSetAccessedBy`, this advice will be the default `label` for `gaussian` benchmark.

We next collect runtime metrics via `Nsight` and `Nvprof` profilers that read hardware performance counters on the GPU at regular intervals which also include unified memory-related metrics.

We run the profiler without any advice in this benchmark and normalize metrics into one `tuple` after removing the features that have the same value across all samples. Therefore, the training set consists of `tuples with labels` and can be used for creating the models.

2) FEATURE ENGINEERING

The initial 59 selected kernel and data object metrics result in 51 non-zero valued metrics generated by the profilers. Collecting all 51 metrics still adds significant overhead and long latency at runtime. Therefore, feature engineering is necessary to select useful metrics in XUnified. Useful metrics are selected by using a feature correlation algorithm. Feature correlation selects the most relevant features from the dataset by evaluating the worth of a feature by measuring the Pearson’s correlation between the feature and its class. This method results in a subset of 10 features that are crucial for the model. Details of selected features are listed in

---

3Nvidia Compute Command Line Interface https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html. To collect all metrics, we use the commands `nv-nsight-cu-cli -o profile <a.out>`


```c
// Modified to utilize unified memory in XUnifiedMalloc()

a = (float*) XUnifiedMalloc(Size + Size + sizeof(float), Managed);
b = (float*) XUnifiedMalloc(Size + sizeof(float), Managed);
...

// Assign unified memory advice for specific data object
#if advOptionA == 2
    cudaMemAdvise(a, Size + Size + sizeof(float), cudaMemAdviseSetReadMostly, 0);
    cudaMemAdviseSetPreferredLocation, 0);
#endif
#if advOptionB == 2
    cudaMemAdvise(b, Size + sizeof(float), cudaMemAdviseSetPreferredLocation, 0);
#endif
...
ForwardSub(); // Run the kernel function
```

Figure 5. Staging the input code of `gaussian` benchmark to allow multiple memory usage choices. Here we use the advice `advOptionA` and `advOptionB` and the `cudaMemAdvise()` function to set the specific advice for the given data object.

Table 2. List of selected features in the model.

| No. | Feature Name                  |
|-----|-------------------------------|
| 1   | Block Size                    |
| 2   | Registers per thread          |
| 3   | Number of Threads             |
| 4   | Waves Per SM                  |
| 5   | Block Limit Registers         |
| 6   | Block Limit Warps             |
| 7   | GPU Page Faults               |
| 8   | HtoD memory transfers         |
| 9   | DioH memory transfers         |

Table 2. It can be observed that the selected features capture metrics corresponding to GPU compute resources and unified memory accesses.

3) MODEL TRAINING

Offline processing creates a well-trained classifier for predicting memory advice in the online inference. We evaluated multiple machine learning classification algorithms with the normalized training dataset obtained from the offline processing.

The goal of the classifier is to learn a function `\( \Theta \)` that takes kernel and data object features as input and predicts the class of the memory advice choice `\( pm \)`.

\[
\Theta: < g_k, g_{d_j} > \rightarrow pm_{d_j} \quad \text{such that} \quad pm_{d_j} = m_{d_j} \quad \forall j
\]

These models that were evaluated with this training data set are Decision tree using J48 algorithm [27], Random Forest [9], Bagging algorithms [27], RandomTree [29] and REPTree [29]. To guarantee the robustness of our model and avoid overfitting, we use the 10-fold cross validation method to measure the prediction accuracy. Thus, we ensure the model’s performance is measured based on its predictions on unseen data. We rely on the model’s prediction accuracy as the metric of evaluation since the GPU memory choice determined by this model has a direct correlation to the program execution time.

B. ONLINE INFERENCE

The offline trained model is used to dynamically guide the choice of memory usage at runtime. Again, an input code needs to be transformed to enable multiple memory usage choices for important data objects (similar to Fig. 5). We use profiling to find largest data objects as candidates since they often have the biggest performance impact.

In the online inference phase, optimal memory advice is determined for important data objects in kernels of a target application. XUnified collects and normalizes the runtime metrics from the target application that is running in default mode using discrete memory. As shown in the first step in Figure 6, XUnified fetches the runtime metrics from the running applications via the `Nsight profiler`. Next, XUnified normalizes them into a reduced test feature vector as the input to the classifier model. After that, the classifier model is called at runtime to predict and set the best memory usage advice option value for each of the instances. In this paper, an instance represents a kernel’s execution using a specific input data, with memory advise given to data objects used in the given kernel. Finally, XUnified automatically executes the staged input source code to enable the execution using the selected memory advice choices.

C. DISCUSSION

1) ADVICE OBJECTIVES

We limit the scope of this work to be finding optimal memory usage choices for major data objects within each kernel level,
TABLE 3. Selected benchmarks for experiments. They are partitioned into two sets. CFD, BFS, Gaussian and HotSpot are used for training models. NW, LUD and PathFinders are used as unseen programs to test resulting models.

| Kernels   | Arrays | Variants | Inputs |
|-----------|--------|----------|--------|
| CFD       | 4      | 3        | (7×7×7) | 3      |
| BFS       | 2      | 4        | (7×7×7×7) | 3     |
| Gaussian  | 2      | 3        | (7×7×7) | 75     |
| HotSpot   | 1      | 2        | (7×7)   | 8      |
| NW        | 2      | 2        | (7×7)   | 64     |
| LUD       | 3      | 1        | (7)     | 64     |
| PathFinder| 1      | 1        | (7)     | 25     |

i.e., a specific memory usage choice is used for each of the selected data objects for a given kernel under all calling contexts. For example, if a program has two kernel functions and both refer to two data objects each during the execution, we can assign two different memory usage choices for the two data objects of the first kernel function. Before entering the second kernel, we can reassign two new memory usage choices for the two data objects. In applications with many kernels, we could apply this optimization to heavy kernels. Also, we focus on optimization on a single node in this work. Applying the model for applications running on a larger number of GPUs could have a similar effect and will be investigated as part of future work.

2) HARDWARE-AGNOSTIC

XUnified is designed to be hardware-agnostic where it omits the variances between different GPUs. Although the experiments are done using NVIDIA GPUs and the corresponding profilers, the selection of profilers and metrics can be changed as needed depending on a particular GPU used. The methodology is portable across different hardware configurations as long as they expose similar memory APIs and ships with supportive profilers. For drastically different architectures, the offline training phase needs to be redone just once.

3) DATA DEPENDENCY

Some applications have dependency among data objects across the kernels, which may affect the optimized performance with specific advice. For applications that have high data dependency in the runtime function, the effects of different advice for the specific dependency would need to be analyzed. Predicting proper advice that relies on data dependency is out of the scope of this paper, and it will be explored and examined in future work.

IV. EVALUATION

XUnified is evaluated by examining its prediction accuracy and overhead. We present the prediction accuracies of the trained models and a comparison of runtime execution latency with the predicted memory advise choice compared to default implementation. We also show the prediction with three new, unseen benchmarks to validate the robustness of the model.

A. EXPERIMENT SETTINGS

We conduct our experiments with multiple benchmarks running on two different GPU machines. The first one is an IBM machine with two IBM Power9 CPUs, four Tesla V100 GPUs, NVLink for GPU and CPU interconnect. It runs Red Hat Enterprise Linux 7.6 with CUDA v10.1.243 installed. Even though there are multiple GPUs on the IBM machine, the CUDA benchmarks we have selected only support a single GPU. The second machine is an AWS instance (p3.2xlarge) with an Intel Xeon processor, a single Tesla V100 GPU, and PCIe for GPU and CPU interconnect. It runs Ubuntu 18.04 LTS with CUDA version v10.2.89 installed. These two machines exhibit diverse behaviour with unified memory. A simple profiling with nvprof reveals higher CPU page faults in the Intel machine compared to IBM.

Several benchmarks from Rodinia [10] are selected for our evaluation, as shown in Table 3. Each benchmark has multiple versions ranging from their baseline versions using discrete memory APIs to versions using unified memory API combined with different CUDA memory advice.

B. PREDICTION ACCURACY EVALUATION

We now present the evaluation of the prediction accuracy of XUnified classifier as the number of correctly predicted values for each evaluated test sample. This gives a measure of how accurate is the model in predicting the ground truth. In particular, we show that for a given benchmark with specific input, XUnified predicts the optimal advice that achieves minimal execution time.

We applied different advice for data objects in the benchmarks and collected 2,688 instances from the IBM and Intel machine with CFD, BFS, Gaussian, and HotSpot. In the dataset, an instance represents a kernel’s execution with a specific input data, with memory advice assigned to the data objects used in the given kernel. After normalization and reformatting, the data is formatted into a single dataset.
different accuracies. This could be due to the features used
data from same applications but on different hardware, yields
interesting to observe that training a single classifier with
classifier performs best and achieves 94.4% accuracy. It is
accuracy up to 86.7%. For the Intel data, Random Forest
classifier on IBM data, it achieves the best performance with
that when implementing the model with a Random Forest classifier to build our learning model.
We found that models trained from data generated on
specific systems showed higher prediction accuracies than the
combined training data generated from both systems.

A snapshot of the decision tree model is shown in Figure 7.
It shows the flow of the features from the root through the leaf	node where it predicts the class of the memory advice, which
is underlined and enumerated in this figure. A partial view
of the decision tree is presented due to the page limits. It can
be observed from Figure 7 that block sizes and data-to-host
(DtoH) features are crucial.

C. VALIDATION ON UNSEEN KERNELS
To further evaluate the robustness of models with new appli-
cations which are unseen during the training, we measure
the predictions on three new kernels Needleman-Wunsch
(NW), LU Decomposition (LUD), and Pathfinder from the
Rodinia benchmarks, as listed in Table 3. NW is a dynamic
programming algorithm to find optimal global alignment of a
pair of DNA sequences. LU Decomposition is a fundamental
algorithm in linear algebra to factor a matrix as a product
of a lower triangular matrix and an upper triangular matrix.
Pathfinder exploits dynamic programming for grid traversal
to find a path on a 2-D grid with the smallest accumulated
weights.

Note that the data from these three benchmarks are
not included in the training set, to ensure the evaluation
is performed on totally new unseen data. We separately
extract 140 instances (from NW), 192 instances (from LUD),
and 225 instances (from Pathfinder) from IBM and Intel
machines, and normalize them as test cases for additional online
reference prediction.

We test trained Random Forest model with the test data
sets from these new benchmarks on Intel and IBM machines.
Table 5 shows the prediction accuracies of the trained models
of the memory advice choices. We can observe that the
selected models correctly predict the optimal memory advice
choices. Specially, the prediction values match to the ground
truth or labelled data for the Pathfinder benchmark for all
instances with 100% prediction accuracy on Intel machine
and 97.7% on IBM system and for NW benchmark on
both Intel and IBM machines, reaching prediction accuracies
above 97%. The optimal variant is the baseline version

| Benchmark | Acc on IBM (%) | Acc on Intel (%) |
|-----------|----------------|-----------------|
| NW        | 97.8           | 98.5            |
| Pathfinder| 97.7           | 100             |
| LUD       | 67.0           | 73.9            |

The measured accuracies of all classifiers across both data
sets are listed in Table 4. Note that the accuracy shows how
close a measurement is to actual advice that the application
should have. F-measure score illustrates the harmonic mean
of the fraction of correctly predicted advice in all predicted
results and the fraction of correctly identified advice in the
results. For the multi-class classification problem, we first
compute the F-measure score per class in a one-vs-rest
manner and then repeat across the classes. The average value
is then presented in the final score that is reported in this
paper.

The measured values are across all the evaluated bench-
marks with various input data set sizes. It can be observed
that when implementing the model with a Random Forest
classifier on IBM data, it achieves the best performance with
accuracy up to 86.7%. For the Intel data, Random Forest
classifier performs best and achieves 94.4% accuracy. It is
interesting to observe that training a single classifier with
data from same applications but on different hardware, yields
different accuracies. This could be due to the features used
to train the model vary with the hardware, that indirectly
affects the prediction accuracies. Other evaluated classifiers
report further lower values. Based on these results, we choose
a Random Forest classifier to build our learning model.

The measured values are across all the evaluated bench-
marks with various input data set sizes. It can be observed
that when implementing the model with a Random Forest
classifier on IBM data, it achieves the best performance with
accuracy up to 86.7%. For the Intel data, Random Forest
classifier performs best and achieves 94.4% accuracy. It is
interesting to observe that training a single classifier with
data from same applications but on different hardware, yields
different accuracies. This could be due to the features used
to train the model vary with the hardware, that indirectly
affects the prediction accuracies. Other evaluated classifiers
report further lower values. Based on these results, we choose
a Random Forest classifier to build our learning model.

The measured values are across all the evaluated bench-
marks with various input data set sizes. It can be observed
that when implementing the model with a Random Forest
classifier on IBM data, it achieves the best performance with
accuracy up to 86.7%. For the Intel data, Random Forest
classifier performs best and achieves 94.4% accuracy. It is
interesting to observe that training a single classifier with
data from same applications but on different hardware, yields
different accuracies. This could be due to the features used
to train the model vary with the hardware, that indirectly
affects the prediction accuracies. Other evaluated classifiers
report further lower values. Based on these results, we choose
a Random Forest classifier to build our learning model.
We found that models trained from data generated on
specific systems showed higher prediction accuracies than the
combined training data generated from both systems.

A snapshot of the decision tree model is shown in Figure 7.
It shows the flow of the features from the root through the leaf	node where it predicts the class of the memory advice, which
is underlined and enumerated in this figure. A partial view
of the decision tree is presented due to the page limits. It can
be observed from Figure 7 that block sizes and data-to-host
(DtoH) features are crucial.

C. VALIDATION ON UNSEEN KERNELS
To further evaluate the robustness of models with new appli-
cations which are unseen during the training, we measure
the predictions on three new kernels Needleman-Wunsch
(NW), LU Decomposition (LUD), and Pathfinder from the
Rodinia benchmarks, as listed in Table 3. NW is a dynamic
programming algorithm to find optimal global alignment of a
pair of DNA sequences. LU Decomposition is a fundamental
algorithm in linear algebra to factor a matrix as a product
of a lower triangular matrix and an upper triangular matrix.
Pathfinder exploits dynamic programming for grid traversal
to find a path on a 2-D grid with the smallest accumulated
weights.

Note that the data from these three benchmarks are
not included in the training set, to ensure the evaluation
is performed on totally new unseen data. We separately
extract 140 instances (from NW), 192 instances (from LUD),
and 225 instances (from Pathfinder) from IBM and Intel
machines, and normalize them as test cases for additional online
reference prediction.

We test trained Random Forest model with the test data
sets from these new benchmarks on Intel and IBM machines.
Table 5 shows the prediction accuracies of the trained models
of the memory advice choices. We can observe that the
selected models correctly predict the optimal memory advice
choices. Specially, the prediction values match to the ground
truth or labelled data for the Pathfinder benchmark for all
instances with 100% prediction accuracy on Intel machine
and 97.7% on IBM system and for NW benchmark on
both Intel and IBM machines, reaching prediction accuracies
above 97%. The optimal variant is the baseline version

### Table 4. Results of the performance of prediction in XUnified.

| Classifier      | IBM Data | Intel Data |
|-----------------|----------|------------|
|                 | Accuracy (%) | F-Measure | Accuracy (%) | F-Measure |
| Random Forest   | 86.7     | 0.87      | 94.4        | 0.934     |
| Decision Tree   | 85.23    | 0.86      | 93.9        | 0.93      |
| Random Tree     | 86.38    | 0.86      | 93.1        | 0.93      |
| Bagging         | 84.04    | 0.83      | 92.3        | 0.92      |
| REPTree         | 82.06    | 0.82      | 90.36       | 0.90      |

### Table 5. Results of the performance of prediction on new, unseen benchmarks. (Acc: accuracy).

| Benchmark | Acc on IBM (%) | Acc on Intel (%) |
|-----------|----------------|-----------------|
| NW        | 97.8           | 98.5            |
| Pathfinder| 97.7           | 100             |
| LUD       | 67.0           | 73.9            |
and our model correctly predicts that. The analysis of the performance results with LUD benchmark is part of future work. It demonstrates that for all input data sets, the memory advice variant predicted by the machine learning model matches with the best variant. This evaluation strengthens the robustness of the models, as they are evaluated with a different test data from a new application.

Results in Table 5 show that XUnified can effectively predict optimal choices for the new and unseen benchmarks. The model is generic and portable across different applications and input data set sizes and thus demonstrates the potential use in guiding the optimal memory choices.

### D. Optimized Execution

We next show the improvement in the execution time of given applications with predicted advice from XUnified. Optimal advice in using discrete memory and unified memory can lead to lower execution latency in the kernel execution. We evaluate the baseline execution latency of each kernel function of the applications and show the optimized execution latency after obtaining the advice from XUnified.

Figure 8 shows performance comparisons in the IBM machine between the baseline benchmarks (the versions using discrete memory APIs) and the versions with the

![Figure 8](image-url)

**FIGURE 8.** Performance comparison in the execution time of each kernel of the benchmarks’ baseline versions and the versions using XUnified’s predicted memory usage advice. Results are from the IBM machine. (NW and LUD are the optimized execution time with corrected predicted advices.) Note that the results only show the optimized kernel execution time.

![Figure 9](image-url)

**FIGURE 9.** The runtime latency of model training and test and the runtime overhead of metrics collection in XUnified. (a) shows the training and prediction latency of different classifiers in XUnified at runtime. (b) shows the latency of reformattin metrics in XUnified.

(a) Training vs. Prediction Time (RTree: Random Tree, RF: Random Forest).

(b) Latency of metrics collection and processing.
predicted memory advice in BFS, CFD, Gaussian, Hotspot, NW, and LUD. Note that for kernel execution time, the baseline versions do not include data copying overhead. However, data copy happens implicitly for unified memory versions so the overhead is included. We can observe that these benchmarks with specific sized inputs show a reduction in their execution latencies. Specifically, the BFS benchmark with an input of 1 million matrix data has 11.8% reduction in the execution latency. CFD benchmark achieves 21% reduction with input 97k. Gaussian benchmark with an input of 656 matrix gets 13.5% in the kernel 1’s execution latency. Hotspot benchmark’s baseline version happens to be the fastest version. As a result, there is no performance improvement for the predicted version compared to the baseline version. NW and LUD benchmarks achieve around 9% reduction with various inputs. Note that we do not display the execution time of PathFinder because the baseline version is the fastest version compared with other advices.

The reductions of execution latency for various benchmarks demonstrate the efficiency of XUnified with predicting optimal advice for a benchmark with specific input. Besides, performances of various benchmarks with larger size of inputs achieve a larger reduction in the latency. In contrast, the comparisons with small inputs in these benchmarks do not show large differences in the execution time. It proves that proper advice for a given large input leads to more benefits for a benchmark at runtime. XUnified can be very useful when guiding benchmarks with large memory footprints to efficiently use unified memory on GPUs.

The performance differences observed on the Intel machine are different compared to what we have seen on the IBM machine. For example, the CFD benchmark achieves up to 34.3% reduction in the execution latency. However, for BFS and Gaussian, the execution latency achieves around 1% to 3.5% reduction with the optimized advice. The results indicate that different benchmarks may have various optimized performances with different hardware or system. Therefore, for large-scale applications in heterogeneous platforms, XUnified should be deliberately implemented and evaluated with various benchmarks.

### E. OVERHEAD

XUnified is designed to be light-weight and fast without incurring huge overhead at runtime. In this subsection, we evaluate the overhead associated with offline training and online inference, using both the IBM and Intel machines. The results are quite similar so we only show one machine’s results.

Figure 9 shows the offline training and online prediction time of XUnified using different classifier algorithms on the IBM machine. The results show that the entire latency of model processing with Random Tree, REPTree, J48, and Bagging is less than 1 second. The latency of online prediction is very small. Figure 9b displays the distribution of time in metrics reformatting to the dataset with various benchmarks. Results show that runtime metrics from GPUs can be quickly reformatted into a proper dataset with around 30 to 50 milliseconds, which proves that XUnified can instantly process programs with low latency.

Besides, we compare the online inference latency with the entire execution of various benchmarks. Overall, the latency of online inference is trivial compared with the long-running various benchmarks, especially benchmarks will run for hundreds of cycles. The experimental results show that the online inference takes additional 2% – 7% latency (around 35 ms) in the BFS benchmark which finishes one execution cycle. The online inference phase takes around 10% to 15% of one execution cycle in the Hotspot, CFD, and gaussian benchmarks. Note that the online inference only takes one time and the predicted advice can be used for the long execution cycles. Therefore, the online inference phase is lightweight and fast and can be efficiently implemented at runtime.

### V. RELATED WORK

Many applications have been implemented with the unified memory in HPC to reduce the complexities of memory management [23]. An early investigation of unified memory [19] shows that applications did not perform well in most cases due to high overhead caused by CUDA 6. Sakharyukh [28] introduces a comprehensive overview of unified memory on three generations of GPUs (Kepler, Pascal, and Volta), with a few application studies using advanced UM features. Chien et al. [13] evaluated memory advises and prefetch of unified memory on Intel and IBM-based GPU systems.

Awan et al. [8] exploit advanced unified memory features in CUDA 9 and Volta GPUs for out-of-core DNN training. They observed minor performance degradation in OC-Caffe with the help of memory prefetching and advice operations. Unified memory is also studied under the context of OpenACC and OpenMP. OpenARC [20] is an OpenACC compiler with extensions to support unified memory. They found that unified memory is beneficial if only a small random portion of data is accessed for a set of benchmarks. Wolfe et al. [30] study how the data model is supported in several OpenACC implementations. They mentioned some implementations were able to use unified memory. Mishra et al. [24] evaluate unified memory for OpenMP GPU offloading. They show that UM performance is competitive for benchmarks with little data reuse while it incurred significant overhead for large amount of data reuse with memory over-subscription. Li et al. [21], [22] propose a compiler-runtime collaborative approach to optimize GPU data under unified memory within an OpenMP implementation. Static and runtime analysis are used to collect data access properties in order to guide adaptive implicit and explicit data transfer as well as prevent data thrashing. HUM [18] introduces the use of unified memory advice to avoid page migration if the pages are read by both the CPU and the GPU without any write (i.e., read-only access). HALO [14] investigates the performance of subsequent traversals on large real world graphs in BFS.
where it uses the read-only memory advise hint with GPU unified memory. Pearson et al. [26] use the ReadMostly unified memory hint to solve performance scaling challenges when scaling in-GPU-memory triangle counting to multiple GPUs. But they do not investigate other advice in both CPU and GPU memory.

Our work differs from the previous studies mentioned above in that we study the hybrid use of both discrete and unified memory APIs, combined with different memory advice options. Our solution uses a unique design including machine learning, code transformation, and runtime support to achieve optimal use of GPU unified memory.

VI. CONCLUSION

In this paper, we propose XUnified, an advice controller which can guide the optimal use of unified memory of GPUs for various applications at runtime. Unlike existing approaches in heterogeneous systems, which rely on the tedious and error-prone manual efforts in using the unified memory, XUnified automatically and accurately guides the optimal use of unified memory rather than conjectural or speculative advice. XUnified consists of two phases: offline learning and online inference. After collecting and filtering the offline metrics from multiple benchmarks, XUnified trains a machine learning-based classifier using supervised learning. It then guides the memory usage at runtime by predicting the optimal memory choices for data objects in kernels. The experimental results show that XUnified alleviates the burden on application developers by automating the complex decision making process which otherwise would require extensive, time-consuming experiments and manual efforts.

In future, XUnified will be applied to optimize more applications (e.g., machine learning libraries including PyTorch [25] and Tensorflow [7]) on more heterogeneous hardware platforms (such as those using Optane memory or Nvidia GPU Direct Storage). An interesting question is that the data dependencies across multiple kernels may affect the optimized performance, we will explore the role of dependencies and propose efficient approach to address this challenge.

The XUnified source is open source, together with all code and data used to produce the results in this paper.

ACKNOWLEDGMENT

An earlier version of this paper was presented at the 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing [DOI: 10.1109/MCHPC49590.2019.00016].

REFERENCES

[1] ROSE Compiler. Accessed: 2022. [Online]. Available: http://rose compiler.org/

[2] (Jun. 2021). Top500 List. [Online]. Available: https://www.top500. org/lists/top500/2021/06/

[3] (Jun. 2017). Unified Memory for CUDA Beginners. [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/

[4] (2015). Gpu Pro Tip: Fast Dynamic Indexing of Private Arrays in CUDA. [Online]. Available: https://devblogs.nvidia.com/fast-dynamic-indexing-private-arrays-cuda/

[5] (2021). CUDA Runtime Api: 5.9. Memory Management. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__MEMORY.html

[6] (2021). CUDA Toolkit 6.0. [Online]. Available: https://developer.nvidia.com/cuda-toolkit-60

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and M. Kudlur, “TensorFlow: A system for large-scale machine learning,” in Proc. 12th USENIX Symp. Operating Syst. Design Implement. (OSDI), 2016, pp. 265–283.

[8] A. A. Awan, C.-H. Chu, H. Subramoni, X. Li, and D. K. Panda, “DC-DNN: Exploiting advanced unified memory capabilities in CUDA 9 and Volta GPUs for out-of-core DNN training,” in Proc. IEEE 25th Int. Conf. High Perform. Comput. (HiPC), Dec. 2018, pp. 143–152.

[9] L. Breiman, “Random forests,” Mach. Learn., vol. 45, no. 1, pp. 5–32, 2001.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in Proc. IEEE Int. Symp. Workload Characterization (IISW), Oct. 2009, pp. 44–54.

[11] G. Chen, X. Shen, B. Wu, and D. Li, “A performance evaluation of unified memory hint to solve performance scaling challenges in CUDA.,” in Proc. IEEE/ACM Workshop Memory Centric High Perform. Comput. (MCHPC), Nov. 2019, pp. 50–57.

[12] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, “Traversing large graphs on GPUs with unified memory,” Proc. VLDB Endowment, vol. 13, no. 7, pp. 1119–1133, Mar. 2020.

[13] B. Ha and C. J. Rossbach, “ALTIS: Modernizing GPGPU benchmarking,” 2019, arXiv:1906.16457.

[14] Y. Huang and D. Li, “Performance modeling for optimal data placement on GPU with heterogeneous memory systems,” in Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER), Sep. 2017.

[15] B. Jiang, D. Schaa, P. Mistry, and D. Kaeli, “Exploiting memory access patterns to improve memory performance in data-parallel architectures,” IEEE Trans. Parallel Distrib. Syst., vol. 22, no. 1, pp. 105–118, Jan. 2011.

[16] J. Jung, D. Park, Y. Do, J. Park, and J. Lee, “Overlapping host-to-device copy and computation using hidden unified memory,” in Proc. 25th ACM SIGPLAN Symp. Princ. Proct. Parallel Program., Feb. 2020, pp. 321–335.

[17] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, “An investigation of unified memory access performance in CUDA,” in Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC), Sep. 2014, pp. 1–6.

[18] S. Lee and J. S. Vetter, “OpenARC: Extensible OpenACC compiler framework for directive-based accelerator programming study,” in Proc. 1st Workshop Accel. Program. Using Directives, Nov. 2014, pp. 1–11.

[19] L. Li and B. Chapman, “Compiler assisted hybrid implicit and explicit GPU memory management under unified address space,” in Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal., Nov. 2019, pp. 1–16.

[20] L. Li, H. Finkel, M. Kong, and B. Chapman, “Manage OpenMP GPU data environment under unified address space,” in Proc. Int. Workshop OpenMP, Cham, Switzerland: Springer, 2018, pp. 69–81.

[21] W. Li, G. Jin, X. Cui, and S. See, “An evaluation of unified memory technology on NVIDIA GPUs,” in Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput., May 2015, pp. 1092–1098.

[22] M. Almasri, O. Anjum, V. S. Mailthody, Z. Qureshi, R. Nagi, J. Xiong, and W.-M. Hwu, “Update on triangle counting on GPU,” in Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC), Sep. 2019, pp. 1–7.

[23] J. R. Quinlan, “Bagging, boosting, and C4.5,” in Proc. AAAI/IAAI, vol. 1, Aug. 1996, pp. 725–730.
H. Xu et al.: XUnified: A Framework for Guiding Optimal Use of GPU Unified Memory

[28] N. Sakharnykh, “Everything you need to know about unified memory,” NVIDIA, Santa Clara, CA, USA, Tech. Rep., 2018. [Online]. Available: https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[29] H. I. Witten, E. Frank, and A. M. Hall, Data Mining: Practical Machine Learning Tools and Techniques, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann, 2011.

[30] M. Wolfe, S. Lee, J. Kim, X. Tian, R. Xu, S. Chandrasekaran, and B. Chapman, “Implementing the OpenACC data model,” in Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW), May 2017, pp. 662–672.

[31] H. Xu, M. Emani, P.-H. Lin, L. Hu, and C. Liao, “Machine learning guided optimal use of GPU unified memory,” in Proc. IEEE/ACM Workshop Memory Centric High Perform. Comput. (MCHPC), Nov. 2019, pp. 64–70.

[32] Y. Yang, P. Xiang, J. Kong, and H. Zhou, “A GPGPU compiler for memory optimization and parallelism management,” in Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI), New York, NY, USA, 2010.

HAILU XU received the B.S. degree in computer science from North China Electric Power University, China, in 2014, the M.S. degree in computer science from The University of Toledo, USA, in 2016, and the Ph.D. degree in computer science from Florida International University, in summer 2020. He is currently an Assistant Professor with the Department of Computer Engineering and Computer Science, California State University, Long Beach, CA, USA. His current research interests include cloud computing, big data systems, and operating systems.

PEI-HUNG LIN received the B.S. degree from the National Tsing Hua University, Taiwan, the M.S. degree from the University of Minnesota, and the Ph.D. degree in computer science from the University of Minnesota, in June 2013. He is currently a Computer Scientist with the Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory. His research interests include compiler optimization, parallel programming model, and domain-specific optimization.

MURALI EMANI received the Ph.D. degree. He worked as a Research Associate at the School of Informatics, Institute for Computing Systems Architecture, The University of Edinburgh, U.K., under the guidance of Prof. Michael O’Boyle. He was a Postdoctoral Research Staff Member at the Lawrence Livermore National Laboratory, USA. He is currently a Computer Scientist with the Data Science Group, Argonne Leadership Computing Facility (ALCF), Argonne National Laboratory. His research interests include parallel programming models, high-performance computing, scalable machine learning, runtime systems, emerging HPC architectures, and online adaptation.

LITING HU received the bachelor’s degree in computer science from the Huazhong University of Science and Technology, China, in 2007, and the Ph.D. degree in computer science from the Georgia Institute of Technology, USA, in 2016. She is currently an Assistant Professor with the Department of Computer Science and Engineering, University of California at Santa Cruz, Santa Cruz, CA, USA. Her research interests include the general area of distributed systems and its intersection with big data analytics, resource management, power management, and system virtualization. She has served as an Editorial Board Member for Future Generation Computer Systems (FGCS) and the Program Co-Chair for the IEEE International Conference on Distributed Computing Systems (ICDCS).

CHUNHUA LIAO received the B.E. and M.E. degrees in computer science from Sichuan University, China, and the Ph.D. degree in computer science from the University of Houston, in August 2007. He has been the Co-Principal Investigator, a Critical Contributor, and a Leading Maintainer of the widely used ROSE Compiler Infrastructure, since 2007. He is currently a Senior Computer Scientist with the Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory. His research interest includes software techniques to improve the performance and correctness of parallel programs.