Machine Learning Guided Optimal Use of GPU Unified Memory

Hailu Xu
Florida International University
Miami, FL, USA
hxu017@fiu.edu

Murali Emani
Argonne National Laboratory
Lemont, IL, USA
memani@anl.gov

Pei-Hung Lin
Lawrence Livermore National Laboratory
Livermore, CA, USA
lin32@llnl.gov

Liting Hu
Florida International University
Miami, FL, USA
lhu@cs.fiu.edu

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, CA, USA
liao6@llnl.gov

ABSTRACT

NVIDIA's unified memory (UM) creates a pool of managed memory on top of physically separated CPU and GPU memories. UM automatically migrates page-level data on-demand so programmers can quickly write CUDA codes on heterogeneous machines without tedious and error-prone manual memory management. To improve performance, NVIDIA allows advanced programmers to pass additional memory use hints to its UM driver. However, it is extremely difficult for programmers to decide when and how to efficiently use unified memory, given the complex interactions between applications and hardware. In this paper, we present a machine learning-based approach to choosing between discrete memory and unified memory, with additional consideration of different memory hints. Our approach utilizes profiler-generated metrics of CUDA programs to train a model offline, which is later used to guide optimal use of UM for multiple applications at runtime. We evaluate our approach on NVIDIA Volta GPU with a set of benchmarks. Results show that the proposed model achieves 96% prediction accuracy in correctly identifying the optimal memory advice choice.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies → Machine learning**.

KEYWORDS

Unified memory, GPU, Data allocation, Machine learning

1 INTRODUCTION

Graphic Processing Units (GPUs) have been the fundamental hardware components for supporting high performance computing, artificial intelligence, and data analysis in datacenters and supercomputers. As of June 2019, 133 of Top 500 supercomputers are GPU-accelerated, and 128 systems debuting on the TOP500 are accelerated with NVIDIA GPUs [16]. Efficient memory management across CPUs and GPUs has been a challenging problem, while it is critical to performance and energy efficiency. Before CUDA 6.0, data shared by CPUs and GPUs is allocated in discrete memories, which require explicit memory copy calls to transfer data between CPUs and GPUs. Since CUDA 6.0, NVIDIA has introduced unified Memory (UM) with a single unified programmable memory place within a heterogeneous CPU-GPU architecture consisting of separated physical memory spaces. UM relieves programmers from manual management of data migration between CPUs and GPUs such as inserting memory copy calls and deep copying pointers. It tremendously improves productivity and also enables oversubscribing GPU memory.

NVIDIA continuously improves UM throughout different generations of GPUs. The latest UM implementation has accumulated a rich set of features including GPU page fault, on-demand migration, over-subscription of GPU memory, concurrent access and atomics, access counters, and so on. Moreover, NVIDIA provides the `cudaMemAdvice`[1] API to advise the UM driver about the usage pattern of memory objects (e.g. dynamically allocated arrays). Different hints (such as `ReadMostly`, `PreferredLocation`, `AccessedBy`) can be specified in this API by programmers to improve the performance of UM. However, it is extremely challenging for programmers to decide when and how to efficiently use UM for various kinds of applications. For a given memory 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.

In this paper, we present a novel approach to choosing between discrete memory and unified memory on GPUs, with additional consideration of different memory usage hints. Our approach consists of two phases: an offline learning phase and an online inference phase. 1) The offline learning phase involves building a classifier via

[1]Nvidia CUDA Runtime API (May 2019) https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

978-1-7281-6007-8/19/$31.00 ©2019 IEEE
DOI 10.1109/MCHPC49590.2019.00016

supervised learning. It first collects the runtime GPU kernel features from selected benchmarks and labels the best advice based upon the performance caused by various GPU memory usage choices. After that, it constructs a classifier that can predict the best advice for new applications. 2) The online inference relates to determining the proper advice at runtime for a running CUDA program. By combining offline learning and online inference, our method can effectively and accurately obtain optimal use of GPU memory for different kinds of CUDA applications and presents fine-grain control over managed memory allocations.

This paper makes the following contributions:

- We study the hybrid use of both discrete and unified memory APIs on GPUs, with additional consideration for selecting different memory advice choices.
- A machine learning-based approach is proposed to guide optimal use of GPU unified memory.
- We design code transformation to enable runtime adaptation of CUDA programs leveraging online inference decisions.
- We incorporate kernel features at runtime to provide fine-grain control over GPU memory.
- Our experiments show that our approach is effective to predict the optimal memory advice choices for the selected benchmarks.

The remainder of this paper is organized as follows: Section 2 presents the background of GPU memory and our motivation. Section 3 describes the details of our design and methodology. Section 4 presents the experimental results of evaluation in GPU. We discuss the related works in Section 5 and conclude this work in Section 6.

2 BACKGROUND AND MOTIVATION

2.1 Choices for Using GPU Memory

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 a single memory space unifying both CPU and GPU memories. CUDA uses a unified memory 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 together within 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 cudaMemoryAdvise, int)`. 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`. 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. The details and differences of these four kinds of advice are presented as follows:

- `Default`: This represents the default on-demand page migration to accessing processor, using the first-touch policy.
- `cudaMemAdviseSetReadMostly`: 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`: Once a target device is specified, this device 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`: 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 the following options: `UnsetReadMostly`, `UnsetPreferredLocation`, and `UnsetAccessedBy`.

2.2 Impact of Different Usage of GPU Memory

Various applications have diverse data access patterns throughout their executions. Different choices of memory APIs and their parameter values often result in a wide variation in performance. To explore the impact of various memory usage choices, we modify several benchmarks from Rodinia [3] to use different memory allocations and advice choices and subsequently examine their execution times. We focus on large dynamically allocated data objects since they usually have major impact on execution time.

Table 1 lists a subset of various code variants for the `gaussian` benchmark in Rodinia. There are two matrices `a`, `m` and one array `b` which are the major data objects in `gaussian`. We apply different memory usage choices to these objects and get multiple combinations. Code variant 1 is the baseline version using the default discrete memory for the three data objects. Variant 2 to 7 use unified memory for matrix `a` and different memory advise of `AccessedBy`, `ReadMostly`, and `PreferredLocation`. We can further specify CPU or GPU as the device for `AccessedBy` and `PreferredLocation`.

We evaluate the performance of all the code variants and present results in Figure 2. The input data is a 1024×1024 matrix and the measurement includes all the memory transferring between

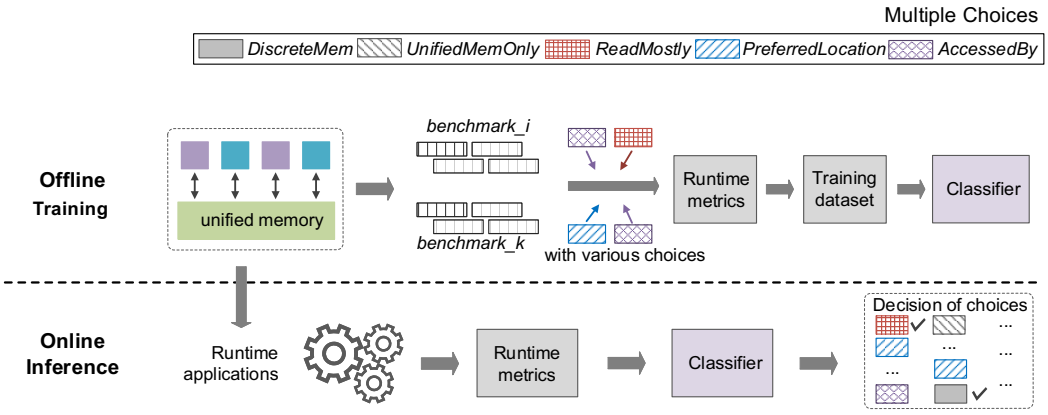


Figure 1: The workflow of the proposed approach in guiding the GPU unified memory advice.

Variants	Description
1	baseline using discrete memory for all objects
2	modified to use unified memory for all
3	set array a with the `ReadMostly` advice
4	set array a with the `PreferredLocation` on GPU
5	set array a with the `AccessedBy` for GPU
6	set array a with the `PreferredLocation` on CPU
7	set array a with the `AccessedBy` for CPU

Table 1: Code variants in the `gaussian` benchmark

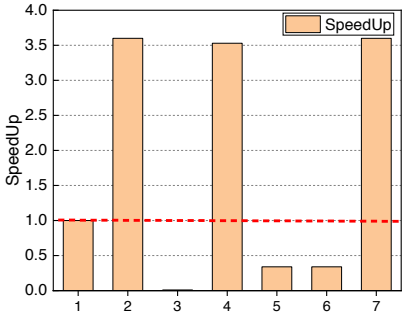


Figure 2: Speedup of different code variants in `gaussian`

the CPU memory and the GPU device memory. It is shown that the code achieves a speedup of 3.5× when matrix `a` is given the `PreferredLocation` to GPU (variant 4). A 200× performance degradation is observed when the `ReadMostly` (variant 3) advice is wrongfully given to matrix `a`. This experiment demonstrates the significant performance impact of choosing the right memory usage of GPU memory.

3 DESIGN

CUDA allows programmers to use either discrete memory or unified memory APIs, potentially combined with different kinds of advice, to manage memory objects in one application or benchmark. Applications and benchmarks can be deployed with various combinations of these choices at different granularity, such as program-level, kernel-level, or object-level. The coarse-grain memory usage optimization is easy to implement but may not deliver the best performance gain. On the other hand, fine-grain memory usage optimization involves carefully deciding a choice for each object of each kernel, even with the consideration of the kernel's calling context. This is challenging to implement but may result the best performance improvements.

We limit the scope of this work to be finding optimal memory usage choices at the object level, i.e., different memory usage choices are used for different data objects for a given kernel function under all calling context. For example, if a program has two kernel functions and both refer to two data objects during the execution, we can assign the default unified memory choice for the first data object for the first kernel function, we then assign the UM combined with the `ReadMostly` advice for the other data object. We can re-assign different choices for the data objects for the next kernel.

3.1 Approach

We are developing a machine learning framework to automatically decide the optimal choice of GPU memory usage for CUDA applications. The framework has a two-phase workflow: offline learning and online inference. As shown in Figure 1, the offline learning phase uses code variants using different GPU memory usage choices, to collect a set of runtime profiling metrics via Nsight CUDA Profiler[2]. The best performing versions are identified and labeled. We then use the collected data as a training data set to construct a classifier model via supervised learning. We explore different kinds of machine learning classifiers such as Random Forest, Random Tree, LogitBoost and select the one that yields highest accuracy and F1 measures. The online inference phase uses Nsight to collect runtime metrics of running applications and passes the input feature vector to the learned model to guide the runtime GPU

[2]Nvidia Compute Command Line Interface https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

memory usage choices for various applications. We elaborate the two phases in the following subsections.

3.2 Offline Learning

In this offline learning phase, we design training configurations that help to capture diverse memory usage variants. Running these experiments will yield the raw training data to train the classifiers.

Training Benchmarks. We manually prepare several variants of selected benchmarks and execute them to find the best performing variant, which is then labelled for supporting training later. Rodinia benchmark [3] is selected to implement different memory usage choices for selected arrays or data structures. Figure 3 presents an example using unified memory and different cudaMemAdvise() settings for two arrays (`a` and `b`) in `gaussian` benchmark. `xplacer_malloc()` is a wrapper function we introduce to switch between discrete or unified memory version of CUDA memory allocation.

```
...
//Modified to utilize unified memory in xplacer_malloc()
a = (float*) xplacer_malloc(Size*Size*sizeof(float), Managed);
b = (float*) xplacer_malloc(Size*sizeof(float), Managed);
...
//Assign unified memory advice for specific data object
#if advOptionA == 1
cudaMemAdvise(a, Size*Size*sizeof(float), cudaMemAdviseSetReadMostly, 0);
#elif advOptionA == 2
cudaMemAdvise(a, Size*Size*sizeof(float),
        cudaMemAdviseSetPreferredLocation, 0);
...
#endif
#if advOptionB == 1
cudaMemAdvise(b, Size*sizeof(float), cudaMemAdviseSetAccessedBy, 0);
#elif advOptionB == 2
cudaMemAdvise(b, Size*sizeof(float), cudaMemAdviseSetPreferredLocation, 0);
#endif
...
ForwardSub(); //Run the kernel function
```

Figure 3: Code showing `gaussian` benchmark using unified memory with memory advice.

Feature Engineering. We utilize the `Nsight Compute` command line profiler to fetch detailed runtime performance metrics of the benchmarks. We implement the data collection on machines using Tesla V100 GPUs. `Nsight Compute` provides metrics organized within different sections. Each section focuses on a specific part of the kernel analysis. The default profiling phase contains 8 sections, including GPU Speed Of Light, Compute Workload Analysis, Memory Workload Analysis, Scheduler Statistics, Warp State Statistics, Instruction Statistics, Launch Statistics and Occupancy. We collect a total of 49 non-zero-valued metrics that correspond to these sections. We then utilize feature correlation and information gain techniques to remove the redundant features. The remaining 9 useful features are listed in Table 2.

Model Training. We evaluate multiple classical machine learning classification algorithms with the collected data. These models include classifiers such as Random Forest, Random Tree and Decision Tree. To guarantee the robustness of our model, we use 10-fold cross validation to verify the model's performance and also ensure

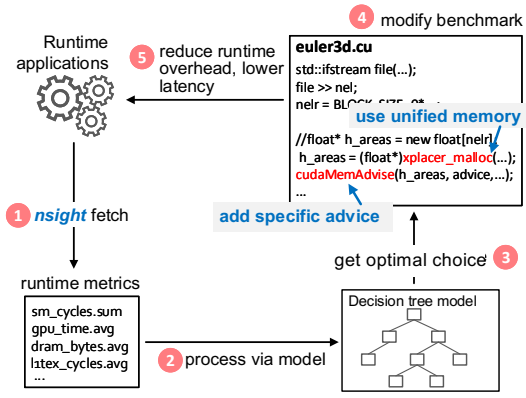


Figure 4: Workflow of the online inference.

No.	Feature Name
1	Elapsed Cycles
2	Duration
3	SM Active Cycles
4	Memory Throughput
5	Max Bandwidth
6	Avg. Execute Instructions Per Scheduler
7	Grid Size
8	Number of Threads
9	Achieved Active Warps Per SM

Table 2: List of selected features in the model.

the model performs evaluation on unseen data. We rely on model's prediction accuracy as the metric of evaluation since the GPU memory choice determined by this model has direct correlation to the program execution time.

3.3 Online Inference

The online inference consists of five major steps, as shown in Figure 4. First, we fetch the runtime metrics from the running applications with the Nsight profiler. Next, a feature vector is composed after normalizing these metrics, which is then passed as input to the offline trained model. The model will then output its predicted memory advice for each of the kernel instance. Once the new predicted choice is given, as shown in the fourth step, we then modify the original application code to implement the optimal choice of memory usage. Modifications to the source code can be automated in the future by a source-to-source tool if available or by a library support to switch among the memory advises. Finally, the optimized code is run with the corresponding memory advice.

4 EVALUATION

4.1 Experiment Settings

We evaluate our approach with multiple benchmarks running on the Lassen supercomputer at Livermore Computing [8]. Each compute node of Lassen has two IBM Power9 CPUs and four Tesla V100 GPUs.

We selected four benchmarks from Rodinia [3] for our evaluation. They are *Computational Fluid Dynamics Simulation* (CFD), *Breadth-first Search* (BFS), *Gaussian Elimination* (Gaussian), and *HotSpot* as shown in Table 3. All the variants are generated by two options of flags (cudaMemAttachGlobal or cudaMemAttachHost) given to data allocation by cudaMallocManaged API, and six memory advise options (no advise, `ReadMostly`, `PreferredLocation` for GPU, `PreferredLocation` for CPU, `AccessedBy` GPU, and `AccessedBy` CPU) for each kernel in the benchmark. For example, we specify memory advises to three arrays in CFD benchmark. The overall number of variants become 432 (2×6×6×6). There are six arrays used in one GPU kernel for the BFS benchmark with a total of 93312 (2×6^6) variants. We reduce the variant number down to only 84 (2×6×6) by only specifying advise to one array in a variant. Note that we use large number of variants to extract the different runtime metrics. When using in the training and prediction, we category these variants by program-level advice based on the most common advice among them with minimal execution time. We use the default input data provided by the Rodinia benchmark suite and generate additional input data sets for HotSpot and Gaussian following the instructions given by the Rodinia suite.

	Kernels	Arrays	Variants	Input data set
CFD	4	3	(2×6×6×6)	3
BFS	2	6	(2×6×6)	3
Gaussian	2	3	(2×6×6×6)	67
HotSpot	1	2	(2×6×6)	8

Table 3: Benchmarks for experiments

4.2 Preliminary Results

We collect total 2,753 instances for training data. After normalization and reformatting, the data is made into one single dataset. We split them into ten subsets, train the evaluated models with many algorithms based on nine of ten, and the final subset is used for examining the predictions of advice from the trained models.

We evaluate the collected data with various classifiers and display the F-measure scores in Figure 5 compared against ground truth which are the memory advice that yield best performance. Note that 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 original results.

The measured values are across all the evaluated benchmarks with various input data set sizes. It can be observed that when implementing the model with a Random Forest classifier, it achieves the best performance with F-measure up to 96.3%. These results establish that our approach can effectively predict optimal choices for the 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.

Fig. 6 shows performance comparisons between the execution time from the original benchmarks which is the baseline for this evaluation and from the codes with the predicted memory advice. All the selected benchmarks with the predicted memory advice achieve equivalent or better performance compared to the original benchmarks. The model can thus effectively assist to achieve better performance for the selected benchmarks.

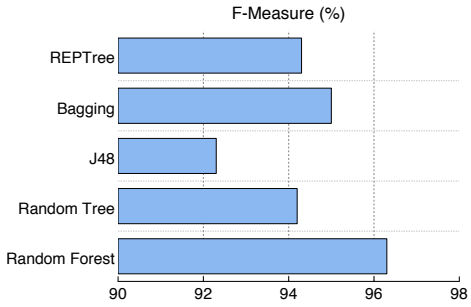


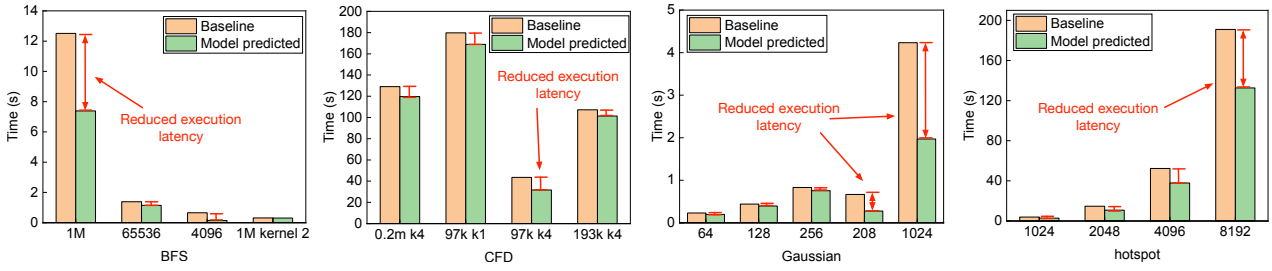
Figure 5: Evaluation of the model prediction accuracies of different classifiers.

5 RELATED WORK

Numerous studies have explored optimization strategies to place data within the various types of memories and caches of GPUs, without considering unified memory. For example, PORPLE [4, 5] is a portable approach using a lightweight performance model to guide run-time selection of optimal data placement policies. Huang and Li [6] have analyzed correlations among different data placements and used a sample data placement to predict performance for other data placements. Jang et al. [7] have presented several rules based on data access patterns to guide the memory selection for a Tesla GPU. Yang et al. [18] proposed compiler-based approach to generate kernel variants for exploiting memory characteristics. More recently, Stoltzfus et al. [15] designed a machine learning approach for guiding data placement using offline profiling and online inference on Volta GPUs. Bari et al. [2] studied the impact of data placement on newer generations of GPUs such as Volta.

Many applications had been implemented with the unified memory in the high performance computing areas to reduce the complexities of memory management [12]. An investigation of early implementation of unified memory [9] showed that applications did not perform well in most cases due to high overhead caused by CUDA 6. Sakharnykh [14] presented a comprehensive overview of unified memory on three generations of GPUs (Kepler, Pascal and Volta), with a few application studies using advanced UM features. Awan et al. [1] exploited 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 advise operations.

Unified memory is also studied under the context of OpenACC and OpenMP. OpenARC [10] is an OpeACC compiler with extensions to support unified memory. They found that unified memory is beneficial if only a small random portion of data is accessed. Wolfe et al. [17] studied how the data model is supported in several OpenACC implementations. They mentioned some implementations were able to use unified memory. Mishra et al [13] evaluated unified memory for OpenMP GPU offloading. They reported that the UM performance was competitive for benchmarks with little data reuse while it incurred significant overhead for large amount data reuse with memory over-subscription. Li et al. [11] proposed 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 to



(a) Execution time of BFS. **(b)** Execution time of CFD. **(c)** Execution time of gaussian. **(d)** Execution time of hotspot.

Figure 6: The comparison of execution times between baseline benchmarks and model predicted performances.

guide if data should be placed on CPU or GPU memory, and how to transfer the data (explicitly through traditional memory copy operations vs. implicitly through UM) if mapped to GPU.

6 CONCLUSION & FUTURE WORK

In this paper, we present a novel machine learning-based approach which can guide the optimal use of unified memory of GPUs for various applications at runtime. It consists of two phases: offline learning and online inference. After collecting and filtering the offline metrics from multiple benchmarks, we train a machine learning model based on remaining useful metrics. We then use the trained model to guide the online execution of applications, by predicting the optimal memory choices for each kernel based on its runtime metrics. The experimental results show that given a set of CUDA benchmarks, the proposed approach is able to accurately determine what kind of memory choices are optimal: either in discrete memory or unified memory space (and combined with various memory advice hints). It alleviates the burden on application developers by automating the complex decision making process which otherwise would require extensive, time-consuming experiments.

In the future, we will extend this work to evaluate the advice choices at a finer granularity considering calling context. Second, using collaborative compiler and runtime support, we will employ runtime code generation and/or adaptation techniques to automatically generate codes using suggested optimal memory choices. Third, we will evaluate the overhead for collecting training data and investigate how to reduce the overhead. Last but not least, the model will be applied to more hardware platforms.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by LLNL-LDRD 18-ERD-006. This research was funded in part by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. LLNL-CONF-793704.

REFERENCES

[1] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K Panda. 2018. OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 143–152.

[2] M Bari, Larisa Stoltzfus, P Lin, Chunhua Liao, Murali Emani, and Barbara Chapman. 2018. *Is data placement optimization still relevant on newer gpus?* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[3] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[4] Guoyang Chen, Xipeng Shen, Bo Wu, and Dong Li. 2017. Optimizing data placement on GPU memory: A portable approach. *IEEE Trans. Comput.* 66, 3 (2017), 473–487.

[5] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 88–100.

[6] Yingchao Huang and Dong Li. 2017. Performance modeling for optimal data placement on GPU with heterogeneous memory systems. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 166–177.

[7] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2010. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel & Distributed Systems* 1 (2010), 105–118.

[8] Lawrence Livermore National Laboratory. 2019. Lassen supercomputer. (2019). https://computing.llnl.gov/computers/lassen

[9] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. 2014. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[10] Seyong Lee and Jeffrey S Vetter. 2014. OpenARC: extensible OpenACC compiler framework for directive-based accelerator programming study. In *Proceedings of the First Workshop on Accelerator Programming using Directives*. IEEE Press, 1–11.

[11] Lingda Li, Hal Finkel, Martin Kong, and Barbara Chapman. 2018. Manage OpenMP GPU Data Environment Under Unified Address Space. In *International Workshop on OpenMP*. Springer, 69–81.

[12] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An evaluation of unified memory technology on nvidia gpus. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1092–1098.

[13] Alok Mishra, Lingda Li, Martin Kong, Hal Finkel, and Barbara Chapman. 2017. Benchmarking and evaluating unified memory for OpenMP GPU offloading. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 6.

[14] Nikolay Sakharnykh. 2018. Everything You Need to Know About Unified Memory. (2018). http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[15] Larisa Stoltzfus, Murali Emani, Pei-Hung Lin, and Chunhua Liao. 2018. Data Placement Optimization in GPU Memory Hierarchy using Predictive Modeling. In *Proceedings of the Workshop on Memory Centric High Performance Computing*. ACM, 45–49.

[16] Tiffany Trader. 2019. Top500 Purely Petaflops; US Maintains Performance Lead. (June 2019). https://www.hpcwire.com/2019/06/17/us-maintains-performance-lead-petaflops-top500-list/

[17] Michael Wolfe, Seyong Lee, Jungwon Kim, Xiaonan Tian, Rengan Xu, Sunita Chandrasekaran, and Barbara Chapman. 2017. Implementing the OpenACC data model. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 662–672.

[18] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 86–97. https://doi.org/10.1145/1806596.1806606

SUMMARY OF THE EXPERIMENTS REPORTED

ARTIFACT AVAILABILITY

Software Artifact Availability:

List of URLs and/or DOIs where artifacts are available:

- Evaluation artifact: https://gitlab.com/AndrewXu22/optimal_unified_memory.git
- Original Rodinia Benchmark: http://lava.cs.virginia.edu/Rodinia/download.htm

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Nvidia GPU Tesla V-100

Operating systems and versions: Debian GNU/Linux 16.04

Compilers and versions: NVCC

Applications and versions: Rodinia Benchmark 3.1

Libraries and versions: CUDA 10.1 with Nsight command line tool

Key algorithms: Random Forest, J48

Input datasets and versions: provided in evaluation artifact

Paper Modifications:

ARTIFACT EVALUATION

The workflow can be performed with the following commands:

```
git clone https://gitlab.com/AndrewXu22/optimal_unified_memory.git
cd optimal_unified_memory
./script/driver.sh
```