Cost-sensitive learning for guided GPU memory decisions using estimated normalized costs: a case study using the Xplacer framework

Brendan Badia
Department of Electrical and
Computer Engineering
Northwestern University
Evanston, Illinois 60201
Email: brendanbadia2021@u.northwestern.edu

Chunhua Liao
Lawrence Livermore National Laboratory
7000 East Ave,
Livermore, CA 94550
Email: liao6@llnl.gov

***Abstract*—More and more research attention is being focused on guiding memory decisions in various applications like GPU acceleration in order to improve performance (computation time, code size e.t.c.). One API that allows for this is Nvidia's CUDA, which gives programmers the ability to pass memory advises to their GPU code in order to improve certain objectives. While these advices can be manually chosen, much research is focused on how to utilize ML to guide these decisions to abstract this work from programmers. However, while certain advises are optimal for a given fragment of code, often there are multiple advises that perform similarly while others may perform much more poorly in terms of a user's objective. Traditional ML algorithms seek only to choose the best option in each case even though these other advices considered as misclassifications carry different costs. Looking to simply classify correctly therefore ignores information about the underlying problem. We build upon previous work which introduced Xplacer (Xu et al, 2019), a ML guided memory advice controller for Nvidia GPUs. In the previous work, advises were chosen by solving a classification problem. We extend the work by utilizing cost-sensitive learning. With no ex ante given costs for each memory advise, we examine three ways of estimating them in order to construct a cost matrix, which is then used to implement cost-sensitive learning. While the simplest method leads to a reduction in performance, we show the other two methods can lead to both improvements in the average computation time (in ms) over all predicted cases as well as reduce the computation time in the worst case misclassification. For example, we are able to get within 0.02% of the optimal total computation time on a dataset with 7566 examples. Our work both serves to demonstrate the value of cost sensitive learning in this specific application as well as introduce simple ways to extend cost-sensitive learning to other work.**

***Index Terms*—High performance computing, CUDA, GPU, Machine Learning, Cost-sensitive learning**

I. Introduction

Utilizing machine learning to guide memory decisions in variety of settings is an emerging field of study. In the high-performance computing realm, the use of GPUs specifically merits methods that can help guide memory decisions with these processors. Many of these are Nvidia GPUs, which offer programmers the ability to use CUDA to offer memory hints [1]. For example, by utilizing the Unified Memory (UM) option, one can have both the CPU and GPU store data in their local memory. In some cases, programmers can use domain knowledge to choose suitable options for their code. However, a more scalable option is to view choosing these options as a classification problem and utilize machine learning in order to pick the correct option to meet some objective (reduce computation time, reduce memory utilization, e.t.c.). Xplacer, introduced in [2], is an ML guided advice controller that learns the best CUDA memory options to reduce computation time and automatically applies it at runtime.

A significant issue with machine learning guided memory like Xplacer is the large variance in performance if sub-optimal options are chosen. Xplacer attempts to choose between 7 memory advices (representing the use of either discrete memory or unified memory with different additional guidance). While one option will perform the best and is chosen as the optimal label, the traditional classification problem that machine learning algorithms attempt to solve will view each other option as equally bad (i.e. as a misclassification). In reality, certain misclassifications can perform much better than others as can be seen in Figure 1. Incorporating this information in these settings therefore incorporates additional information about the underlying goal. If applied properly, this should improve the effectiveness of our solution: this paper aims to demonstrate this.

Utilizing this information in machine learning applications has been studied under the umbrella of cost-sensitive learning[1]. It is first necessary to define costs for classifying data with a true label i as label j. These costs are used to construct a weight vector over all labels, with data that is on average more costly when misclassified having higher weights. This weight vector is passed into into existing machine learning algorithms. For example, the weight vector can be passed into a decision tree algorithm which causes more heavily weighted classes to impact the purity of a split more so than more lowly

[1]Cost-sensitive learning is also used to help with unbalanced datasets, but we do not examine that issue in this work.

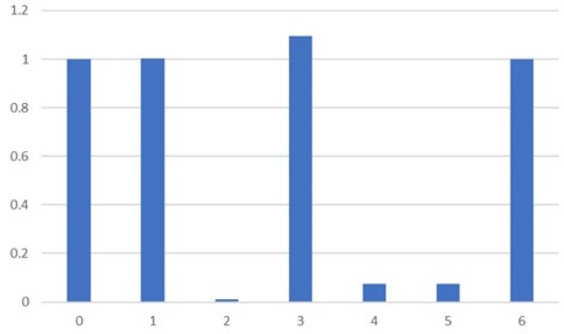

Fig. 1. Speedup with different code variants in an example from the *gaussian* benchmark. The traditional classification problem would label this as 3, and treat all other labels as incorrect. In reality, labels 0,1 and 6 offer similar performance while labels 2,4 and 5 significantly reduce performance. The cost-sensitive approach attempts to incorporate these subtleties that a traditional classification problem would miss.

weighted cases.

However, unless one has domain knowledge about these costs it may it is unclear how to utilize cost-sensitive learning. In this work we look at the specific case of choosing CUDA memory advice, and there is no reasonable way to generate a data set where each type of misclassification has the same cost or where we have ex-ante knowledge about the distribution of costs. In order to use cost-sensitive learning, a measure of average cost has to be estimated. In this work we focus on doing this, and show how different pre-processing of the data can have a large impact on the efficacy of using cost sensitive learning versus a traditional classification algorithm.

We use the Xplacer framework introduced in [2]. Previously, the choice of what memory option to pass forward was treated as a classification problem. In this work we attempt to quantify the cost of misclassification in three ways and analyze the performance of the same learning algorithm with and without cost-sensitive learning.

II. XPLACER FRAMEWORK

Before we set up the contribution of this paper, we briefly discuss Xplacer and mention our additions to extend it to the cost-sensitive framework[2]. Xplacer is an advice controller to guide whether to use discrete or unified memory (UM) in GPU applications as well as what additional advice to set if using UM through CUDA. It consists of two components, First, a supervized model is trained offline using GPU kernel and data object features to predict which advice to pass. Secondly, an online inference step with determines the memory advice and applies it at runtime for the CUDA program. Table I lists the seven chosen data advice that Xplacer can set: this means the offline model has a choice of 7 labels to choose (each one representing a different memory advice set).

[2]More specific details can be found in [2]

TABLE I
MEMORY ADVICE SET FOR DATA OBJECTS IN EACH VARIANT.

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

TABLE II
LIST OF SELECTED FEATURES IN THE MODEL

No.	Feature Name
1	Executed IPC elapsed
2	Issued Warp per Scheduler
3	Avg Executed Instructions Per Scheduler
4	Block Size
5	Registers per thread
6	Number of Threads
7	Waves Per SM
8	Block Limit Registers
9	Block Limit Warps
10	CPU Page Fault
11	GPU Page Fault
12	HtoD
13	DtoH

In Table II, the 14 data and kernel level features that have been chosen are listed. In the Xplacer work, labels for each training data point was by choosing the lowest average computation time for each kernel with each data object. In this work, we keep this labelled data but also compute the cost for each kernel and each data object using each of the seven advices. We store this as a vector for each training point. The previous work used Weka to build the offline trained models and experimented with a variety of models [3]. For this work, we utilize the scikit-learn library from python and restrict ourselves to decision trees as they can easily be adapted for use in the cost-sensitive framework [4].

III. COST-SENSITIVE LEARNING EXTENSION

Here we set-up our extension to [2]. We first discuss how we construct a cost vector for each data point, which captures how costly each label is for each example in the training dataset. We then discuss how to use these in three different ways to construct a cost matrix for the entire data set, which gives a dataset wide estimated cost for classifying an instance with a true label i as some label j. Finally we show how these are used to construct a weight vector which is then used to modify the classification problem to incorporate cost-sensitive learning.

A. Example cost vector

For this first step, we take the same data files as inputs as the previous Xplacer work. Each row of input data represents a specific kernel and data object, with advises numbered 0 to

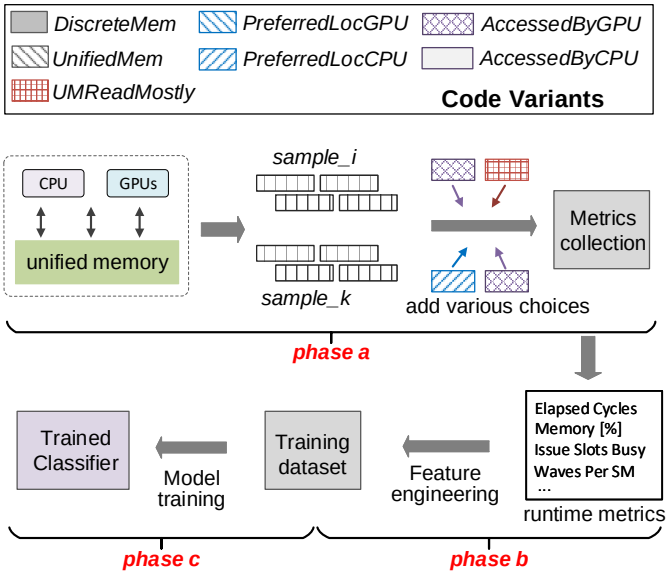

Fig. 2. The workflow of the online inference in `XPlacer`.

6 (representing the choices in II for up to three arrays (these are called advise0, advise1, advise2). For each case, multiple runs are performed and the average computation time (in ms) over all runs is stored as AVG. In [2], a label was chosen by iterating through all rows and choosing the one with the lowest AVG for each array. An example of a couple rows of data is shown in Figure 3.

For our method of cost sensitive learning we first aim to compute the cost of each advise for each data point, which represents a specific kernel with a specific data object and a specific array. We do this by constructing what we can an "example cost vector". This vector $\boldsymbol{v}$ is length 7 and for a given data point i and label j is defined such that $\boldsymbol{v}_i[j]$ is equal to the cost of computation in ms of labelling data point i and j. We compute this for each data point by taking the mean of AVGs for every run for that given data point with the advise set for that label. We still maintain the true label for each data point which is simply $\arg\min_j \boldsymbol{v}_i[j]$.

advise0	advise1	advise2	variant	TimePerc	Time	calls	AVG
0	2	1	21	68.8	192.6389	6000	0.032106
0	2	2	22	68.11	193.5288	6000	0.032254
0	2	3	23	77.56	304.7278	6000	0.050787
0	2	4	24	69.01	195.0802	6000	0.032513
0	2	5	25	84.32	479.5308	6000	0.079921

Fig. 3. Subset of data (others columns omitted for clarity). For each array (advise0, advise1, advise2), different memory advises (numbered 0 to 6) are given and the code is run multiple times to construct an average computation time in ms. This is used to either label the data (taking the minimum) or used to construct the example cost vectors (by taking the average of the AVG of all runs with a given memory advice set).

B. Cost Matrix Estimation

To perform traditional cost-sensitive learning, it is necessary to have a cost matrix. With l labels, the cost matrix is defined as a lxl matrix C where $C(i,j)$ is the expected cost of labelling an example with a true label of j as if it had label i. This cost matrix is assumes a known and fixed estimate for the cost for misclassification. The heart of this work is experimenting with ways to best approximate the matrix and in this section we introduce three methods we use to do so. For each of these (as well as the previous method of using cost-insensitive learning, i.e. not using any costs) we assign a "method id" (M0,M1,M2 and M3) to make it easier to refer to them later in the paper. Performing cost-insensitive learning is called M0 (as it is the base method we compare other methods to).

The first is to simply to use the estimate of the sample mean over all data points. We call this the mean estimator and designate it M1. Define the number of samples is n, the number of samples classified as label j is n_j, and the cost of classifying a sample labelled as j with label i is $c_j(i)$. The mean estimate computes the cost matrix as:

$$C(i,j) = \frac{1}{n_j}\sum_{i=1}^{n_j} c_j(i). \tag{1}$$

The hope with this estimate is that for a given true label and predicted label there are enough data points to approximate the underlying distribution and that the true costs are clustered around this mean. However, this method is highly dependent on the sample data fed in.

When feeding in features to a machine learning algorithm, it is often the case they are scaled in a variety of ways. The cost matrix will be fed into a final weight vector for each label, which gives the relative importance of each label. Therefore, even in cases where the costs don't represent the absolute penalty of missclassifications the cost matrix still has meaning if it captures relative differences between them well. Thus, we propose first normalizing the costs before we use them to compute the mean in two ways in hopes of dealing with potential issues in the underlying data. This can help prevent certain instances with large values dominating the expected cost that the mean estimate may be sensitive to[3].

Min-max normalization is one commonly used method to normalize feature data in machine learning. Different scales can be used by we choose to scale all values to be between 0 and 1. This normalization tends to work well when data is not Gaussian and reduces the absolute differences in outliers (but not does remove the outliers). In this case, we estimate the mean after we have used the normalization scheme on the costs for each data point. We call this method M2 and the cost matrix in the min-max normalization scheme is computed as follows:

$$C(i,j) = \frac{1}{n_j}\sum_{i=1}^{n_j} \frac{c_j(i) - min_{k\in[1,l]}(c_j(k))}{max_{k\in[1,l]}(c_j(k)) - min_{k\in[1,l]}(c_j(k))}. \tag{2}$$

[3]In [5], the authors also discuss a case where costs are unknown. They introduce the concept of cost-free learning, where they attempt to derive equivalent costs. Our attempts at normalizing costs and estimating over them could be viewed as attempting something similar.

Mean normalization is another technique utilized to modify feature values. Values are defined between -1 and 1 with a mean of 0. We call this method M3. It is similar to the min-max normalization but the scaling is made proportional to the mean rather than the overall range. As before, we normalize the data and then take the mean over the normalized data to construct the cost matrix. We denote this method as M3 and compute the cost matrix in the mean normalization scheme with the following equation:

$$C(i,j) = \frac{1}{n_j}\sum_{i=1}^{n_j} \frac{c_j(i) - \frac{1}{k}\sum_{k=1}^{l} c_j(k)}{max_{k\in[1,l]}(c_j(k)) - min_{k\in[1,l]}(c_j(k))}. \tag{3}$$

C. Weight Vector

The cost matrix is finally used to construct a single weight vector which will be used to conduct the actual cost-sensitive learning. The length of the weight vector is equal to the number of labels (7 in this case) and represents the relative importance of each label. The weight vector is set such that for any two labels i and j we have

$$\frac{w[i]}{w[j]} = \frac{C(j,i)}{C(i,j)}.$$

This is meant to internalize the fact that we want to make classifications to reduce the expected cost of a miss, not just the number of misclassifications. In the traditional classification problem, we can view this weight vector as being equal to 1 for all entries, i.e. the cost of missclassification is the same for all data points.

We use the SimpleCost method to calculate a weight vector introduced in [6]. Define $cost(i) = \sum_{j=1}^{j=l} C(i,j)$. The weight vector is calculated as

$$w[i] = \frac{n * cost(i)}{\sum_{j=1}^{j=l} n_j * cost(j)}. \tag{4}$$

This weight vector is passed into a standard machine learning algorithm (in our case a decision tree) and represents the relative importance of each class based on how costly it is to miscalssify them. In the decision tree case, this is used to modify the impurity method used to choose splits in the decision tree: the more highly weighted classes result in a higher impurity if they are misclassified than less highly weighted ones.

We summarize each method in Table III.

TABLE III
METHOD SUMMARY

Method ID	Cost matrix	Weight vector
M0	None	$w = [1,1,1,1,1,1]$
M1	(1)	(1)
M2	(2)	(1)
M3	(3)	(1)

IV. EXPERIMENTS

A. Set-up

TABLE IV
EXPERIMENT PLATFORMS

Configuration	IBM Machine	Intel Machine
CPU	Power 9	Xeon E5-2686
Freq.	3.45GHz	2.30GHz
Cores	44	8 (vCPUs)
Main Memory	256 GB	64 GB
GPU	4 V100-SXM2	1 V100-SXM2
GPU Memory	16 GiB	16 GiB
InterConnect	NVLink	PCIe
OS	RHEL 7.6	Ubuntu 18.04 LTS
CUDA	v10.1.243	v10.2.89

Using the Sci-kit library in Python, we use the decision tree classifier with the weight vectors computed using each method as described in Table III [4]. We use four benchmarks from the Rodinia suite: computational fluid dynamics (CFD), breadth first search (BFS), Gaussian Elimination and HotSpot [7]. We run these experiments on two machines with specifications denoted in Table IV. We look at three sets of data: the four Rodinia suites computed on the IBM machine, the same four on the Intel machine, and the combination of these two data sets (which we call combined data).

We define the cost of labelling a specific data point i with label j as $c_i(j)$. For each method we compute the average cost using their predictions when trained on the entire data set, which for N data points is simply the mean, i.e. $\bar{c} = \sum_{i=1}^{N} c_i(j_{predicted})$. The max missed cost is defined as the largest difference in computation cost using a predicted label versus the true label. This is used to give us a sense of how poorly each method performs in the worst case. We also compute the prediction accuracy when on the entire dataset and the average accuracy using 10-fold stratified cross validation.

Costs	label	predicted label	Predicted cost
0.05334601749271136, 0.0512	0	0	0.053346017
0.036406696793002916, 0.031	4	4	0.056592551
0.15778656851311956, 0.2046	0	0	0.157786569
0.20442413994169092, 0.1778	0	0	0.20442414
0.13510127988338183, 0.1443	0	3	0.145111452
0.16257679008746348, 0.1018	5	5	0.310145085

Fig. 4. Example data (excluding featutes)

Figure 4 gives an example of how the data is stored. With the cost vector and true label stated, we save the predicted label and the predicted cost (derived from the predicted label and example cost vector) and use that to compute accuracy, average cost, and max missed cost.

[4]We use the default hyperparameters. Another interesting course of study would be to look at how optimizing these hyperparamters differs in the cost-sensitive and cost in-sensitive cases.

B. Results

TABLE V
IBM MACHINE (2688 DATA POINTS) COSTS

Method ID	Average Cost (ms)	Max missed cost (ms)
M0	0.3379	0.0539
M1	0.3652	18.1855
M2	0.3381	0.01466
M3	0.3378	0.01466

TABLE VI
IBM MACHINE (2688 DATA POINTS) ACCURACIES

Method ID	Accuracy	Accuracy (10-fold)
M0	92.82%	88.09%
M1	77.86%	75.41%
M2	92.26%	87.46%
M3	92.26%	87.61%

Looking at the results on the IBM machine results in Table V and Table VI, we see a couple interesting outcomes. When using the mean-estimate for the cost, accuracy (both over the total data set and with cross-validation) drops dramatically. This is not counterbalanced by a reduced average cost: in fact, average cost is higher than the cost-insensitive case and the largest miss is far higher. However, with the min-max and mean normalization cases, we see a very small decrease in overall accuracy. The largest miss is decreased in both cases by a factor of almost three, and while average cost is slightly higher in the min-max normalized case, it is slightly lower in the mean normalized case.

TABLE VII
INTEL MACHINE (4878 DATA POINTS) COSTS

Method ID	Average Cost (ms)	Max missed cost (ms)
M0	0.39277	2.51067
M1	0.47635	122.063
M2	0.38756	2.51067
M3	0.38462	2.51067

TABLE VIII
INTEL MACHINE (4878 DATA POINTS) ACCURACIES

Method ID	Accuracy	Accuracy (10-fold)
M0	96.57%	94.03%
M1	37.166%	36.06%
M2	96.37%	94.11%
M3	96.43%	94.93%

We see some similar trends with the Intel results in Table VII and Table VIII with a few key differences. The mean estimate case underperforms even more than before, with extremely low accuracy and large average cost. Both the min-max and mean normalized case however both reduce average cost, with the mean normalization doing the best. While the accuracy on the overall dataset is of course lower with both normalized cost estimates (as it should be for any weight vector that is not comprised of all 1s), the cross validated accuracy is actually larger in both cases. Using the cost estimate may have reduced any potential overfitting.

TABLE IX
COMBINED DATA (7566 DATA POINTS) COSTS

Method ID	Average Cost (ms)	Max missed cost (ms)
M0	0.37326	2.51067
M1	0.46871	122.063
M2	0.36896	2.51067
M3	0.36805	2.51067

TABLE X
COMBINED DATA (7566 DATA POINTS) ACCURACIES

Method ID	Accuracy	Accuracy (10-fold)
M0	95.07%	91.25%
M1	35.59%	33.28%
M2	94.95%	91.83%
M3	94.95%	91.55%

When we combine the data, the results in Table IX and Table X are similar to the Intel machine. The mean estimate again performs the worst, while both normalization schemes reduce average cost and increase cross validated accuracy.

While we note the differences in average costs seem small, this is an artifact of the cost in-sensitive doing so well with the given data. Examining how close each method gets to performing completely optimally is, however, revealing. For example, in the combined data set the mean normalized method (M3) comes within 0.02% of the optimal average cost (i.e. the case where every single label was predicted accurately) while the cost insensitive method comes within 1.5%. In data sets with larger potential misses and/or with lower accuracies we would anticipate that cost sensitive learning could have a larger absolute impact.

Lastly, we would like to discuss how our attempts at utilizing cost-sensitive data the cost sensitive learning is affected by small changes in datasets. For the IBM data we removed the Hotspot benchmark which consisted of 16 data points with by far the largest computation times (this means 2872 data points remain). When we attempted to use the mean estimate for costs, the prediction accuracy was 8.45% for the overall dataset, as it classified every point as 0. Both normalization methods still performed well, with 92.21% accuracy on the dataset compared to 92.77% for the cost-insensitive case. Normalizing the estimate may help in a similar way that normalizing features can help in traditional classification machine learning settings. By reducing the distribution of costs between data points, the estimate is less sensitive to fluctuations in training data. In this case, the hotspot data is such an outlier and that its exclusion affected the mean-sensitive case in a catastrophic way.

The weight vector represents a relative ranking of labels, and so while normalizing the data may seem to affect the

accuracy of the cost estimate, it seems it in fact gives a better relative estimate of labels than a method with lower bias in these experiments.

V. Related Works

The use of machine learning for compiler optimization has widely studied since the 1990s [8]–[12]. In [8], the authors introduce MILEPOST GCC which they claim was the first machine learning tuner for compilers (as the name implies it was built specifically for GCC). They show results on the MiBench programs, with execution speeds up two twice as quickly with reductions in compilation time and code size as well. A survey of work in this space is offered in [10]. They mention open problems in the space, notably what optimizations to use, what parameters to achieve them, and what order to use them in. With Xplacer we have chosen what optimization to choose and what parameters to achieve them, with this work focused on how to improve the optimization once the parameters have been set.

The use of machine learning in a variety of high-performance computing (HPC) applications has be examined by a variety of authors [13]–[18]. The authors of [14] develop an ML model to predict the power consumption of jobs run on supercomputers to facilitate power capping (reducing workload when power usage approaches a threshold). In [18], Netti et al. trained offline classifiers for different computer resource types to detect faults in HPC nodes. Given the increasing use of GPUs in HPC setting, a related field of study is work focused on using machine learning to guide memory decisions in GPUs [19]–[23]. For example, [22] introduces an ML guided load balancer for CPU and GPU connected systems called Troodon, which extracts 23 features from openCL to guide jobs towards either the GPU or CPU depending on the job suitability at a given time. The work on Xplacer attempts to fit in to this by improving computation times in GPU accelerated HPC applications. This specific work offers an extension to this by offering further optimizations to the offline training section, and the use of cost-sensitive learning may help other authors in this space to meet their objectives more effectively.

Finally, there is much literature on cost-sensitive learning in general settings [5], [6], [24]–[31]. Elkan describes the theory of cost-sensitive learning in [24] and discusses how to implement it in practice. The authors in [29] utilize cost-sensitive learning to attempt to reduce re-admissions to a hospital by identifying patients who need further preventative care. They found their algorithm performed better than state-of-the-art competitors and it has been utilized in a real life setting. Our work attempts to demonstrate the value of cost-sensitive learning in optimization memory options in GPU accelerated settings as well as propose the best method to do so.

VI. Conclusion

In this work we took an existing machine learning application in the HPC space (Xplacer) and modified it to utilize cost-sensitive learning to better meet the objective of reducing computation times. As fixed costs for using the wrong labels are unknown, we looked at three different ways to attempt to estimate a cost matrix. We showed that a simple mean estimate performed extremely poorly, but that by normalizing the cost data and taking the expectation over this normalized data average costs can be reduced. Furthermore, we showed some evidence that the trained models performed better with out of data samples.

Further work could focus on more complicated transformations of the cost data (for example Z-scaling) or attempt to utilize the cost of each example to tune the learning algorithm. We would also be interested to see how our normalized estimates could be used in other work we mentioned that use ML algorithms in the HPC space to see if they improve upon their results. Some more theoretical work on estimating costs could also be done.

Acknowledgement

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.

References

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[2] H. Xu, M. Emani, P.-H. Lin, L. Hu, and C. Liao, "Machine learning guided optimal use of gpu unified memory," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 64–70.

[3] F. Eibe, M. A. Hall, and I. H. Witten, "The weka workbench. online appendix for data mining: practical machine learning tools and techniques," in *Morgan Kaufmann*, 2016.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[5] X. Zhang and B.-G. Hu, "A new strategy of cost-free learning in the class imbalance problem," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 2872–2885, 2014.

[6] Z.-H. Zhou and X.-Y. Liu, "On multi-class cost-sensitive learning," *Computational Intelligence*, vol. 26, no. 3, pp. 232–257, 2010.

[7] 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 *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[8] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[9] B. Taylor, V. S. Marco, and Z. Wang, "Adaptive optimization for opencl programs on embedded heterogeneous systems," *ACM SIGPLAN Notices*, vol. 52, no. 5, pp. 11–20, 2017.

[10] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[11] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[12] A. K. A. Al Ghadani, W. Mateen, and R. G. Ramaswamy, "Tensor-based cuda optimization for ann inferencing using parallel acceleration on embedded gpu," in *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, 2020, pp. 291–302.

[13] E. R. Rodrigues, R. L. Cunha, M. A. Netto, and M. Spriggs, "Helping hpc users specify job memory requirements via machine learning," in *2016 Third International Workshop on HPC User Support Tools (HUST)*. IEEE, 2016, pp. 6–13.

[14] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "Predictive modeling for job power consumption in hpc systems," in *International conference on high performance computing*. Springer, 2016, pp. 181–199.

[15] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in hpc applications using machine learning," in *International Supercomputing Conference*. Springer, 2017, pp. 355–373.

[16] B. Nie, J. Xue, S. Gupta, T. Patel, C. Engelmann, E. Smirni, and D. Tiwari, "Machine learning models for gpu error prediction in a large scale hpc system," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 95–106.

[17] M. Tanash, B. Dunn, D. Andresen, W. Hsu, H. Yang, and A. Okanlawon, "Improving hpc system performance by predicting job resources via supervised machine learning," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, 2019, pp. 1–8.

[18] A. Netti, Z. Kiziltan, O. Babaoglu, A. Sîrbu, A. Bartolini, and A. Borghesi, "Online fault classification in hpc systems through machine learning," in *European Conference on Parallel Processing*. Springer, 2019, pp. 3–16.

[19] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2014, pp. 254–261.

[20] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar, "Machine-learning-based performance heuristics for runtime cpu/gpu selection," in *Proceedings of the principles and practices of programming on the Java platform*, 2015, pp. 27–36.

[21] H. Zhu and X. Wang, "A cost-sensitive semi-supervised learning model based on uncertainty," *Neurocomputing*, vol. 251, pp. 106–114, 2017.

[22] Y. N. Khalid, M. Aleem, U. Ahmed, M. A. Islam, and M. A. Iqbal, "Troodon: A machine-learning based load-balancing application scheduler for cpu–gpu system," *Journal of Parallel and Distributed Computing*, vol. 132, pp. 79–94, 2019.

[23] A. Iranmehr, H. Masnadi-Shirazi, and N. Vasconcelos, "Cost-sensitive support vector machines," *Neurocomputing*, vol. 343, pp. 50–64, 2019.

[24] C. Elkan, "The foundations of cost-sensitive learning," in *International joint conference on artificial intelligence*, vol. 17, no. 1. Lawrence Erlbaum Associates Ltd, 2001, pp. 973–978.

[25] B. Zadrozny, J. Langford, and N. Abe, "Cost-sensitive learning by cost-proportionate example weighting," in *Third IEEE international conference on data mining*. IEEE, 2003, pp. 435–442.

[26] G. M. Weiss, K. McCarthy, and B. Zabar, "Cost-sensitive learning vs. sampling: Which is best for handling unbalanced classes with unequal error costs?" *Dmin*, vol. 7, no. 35-41, p. 24, 2007.

[27] O. Mac Aodha and G. J. Brostow, "Revisiting example dependent cost-sensitive learning with decision trees," in *Proceedings of the IEEE International Conference on Computer Vision*, 2013, pp. 193–200.

[28] A. Krishnamurthy, A. Agarwal, T.-K. Huang, H. Daumé III, and J. Langford, "Active learning for cost-sensitive classification," in *International Conference on Machine Learning*, 2017, pp. 1915–1924.

[29] H. Wang, Z. Cui, Y. Chen, M. Avidan, A. B. Abdallah, and A. Kronzer, "Predicting hospital readmission via cost-sensitive deep learning," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 15, no. 6, pp. 1968–1978, 2018.

[30] Y.-X. Wu, X.-Y. Min, F. Min, and M. Wang, "Cost-sensitive active learning with a label uniform distribution model," *International Journal of Approximate Reasoning*, vol. 105, pp. 49–65, 2019.

[31] X. Zhu, J. Yang, C. Zhang, and S. Zhang, "Efficient utilization of missing data in cost-sensitive learning," *IEEE Transactions on Knowledge and Data Engineering*, 2019.