KMLib: Towards Machine Learning For Operating Systems

Ibrahim Umit Akgun [1] **Ali Selman Aydin** [1] **Erez Zadok** [1]

Abstract

Despite the ever-changing software and hardware profiles of modern computing systems, many operating systems (OS) components adhere to designs developed decades ago. Considering the variety of dynamic workloads that modern operating systems are expected to manage, it is quite beneficial to develop adaptive systems that learn from data patterns and OS events. However, developing such adaptive systems in kernel space involves the bottom-up implementation of math and machine learning (ML) primitives that are readily available in user space via widely-used ML libraries. However, user-level ML engines are often too costly (in terms of CPU and memory footprint) to be used inside a tightly controlled, resource constrained OS. To this end, we started developing KMLib, a lightweight yet efficient ML engine targeting kernel space components. We detail our proposed design in this paper, demonstrated through a first prototype targeting the OS I/O scheduler. Our prototype's memory footprint is 804KB for the kernel module and 96KB for the library; experiments show we can reduce I/O latency by 8% on our benchmark workload and testbed, which is significant for typically slow I/O devices.

1 Introduction

Rapid changes in hardware that are interacting heavily with operating systems raise questions about OS design. OS development is a difficult and tedious task, and it is not able to keep with these hardware changes or new algorithmic techniques quickly. In addition, recent years have witnessed major changes in workloads. Contrary to these changes, most of the OS components' designs have changed little over the years.

One example of the divergence between hardware and software can be seen in storage technologies. Storage devices are getting faster and different every day. Keeping up with the changes to storage devices require either a complete re-design of some of the components in the storage stack or tuning parameters and developing more workload-aware data structures and algorithms. In the past few years, we have witnessed such a paradigm shift in data management systems and computer architectures. Both OS research and these fields tackle similar tasks such as caching, indexing, and scheduling. For example, in the data management system research, researchers have developed learned structures to improve performance and adaptability(Kraska et al., 2018).

This is followed by work on data management systems that are optimized for workloads and underlying system specifications (Kraska et al., 2019). In computer architecture research, researchers realized that predicting memory access patterns can be formulated as an ML problem, and they developed cache-replacement models to improve the system performance (Hashemi et al., 2018; Shi et al., 2019). OS page-cache management is a similar problem as cache-replacement in CPUs. In addition, operating systems use hash tables in numerous places, which might be enhanced with learned structures (Kraska et al., 2018).

Although it is possible to utilize well-known ML libraries to build ML approaches for data management systems, using ML in operating systems poses unique three challenges. (1) Developing ML solutions working in kernel space requires extensive kernel programming skills. (2) Debugging and fine-tuning ML models, which is an essential component of most ML development pipelines, could be quite challenging for ML models working only in kernel space, because the OS is naturally hard to debug and notoriously sensitive to bugs and performance overheads. (3) Certain QoS for operating system requirements could require ML models to be deployed in kernel space to avoid the extra costs incurred for user-kernel switches. There are kernel tasks that can not tolerate the overhead of user-kernel switches. Because these kernel tasks might be running under hard time limits, and adding extra overhead can cause timeouts. These challenges motivated us to design and develop an ML library targeted for adoption within the kernel, called KMLib. KMLib is an attempt to enable ML applications in a relatively unexplored yet challenging environment of the OS

---

[1]Department of Computer Science, Stony Brook University, USA. Correspondence to: Ibrahim Umit Akgun <iakgun@cs.stonybrook.edu>, Ali Selman Aydin <aaydin@cs.stonybrook.edu>, Erez Zadok <ezk@cs.stonybrook.edu>.

kernel. Researchers have proposed interesting ideas related to ML for task scheduling (Negi & Kumar, 2005; Smith et al., 1998), I/O scheduling (Hao et al., 2017), and storage parameter tuning (Cao et al., 2018). However, to the best of our knowledge, there is no previous work that attempts to develop an ML ecosystem for operating systems.

KMLib aims to (i) enable easy to develop ML applications with low computational cost and memory footprint and (ii) make it easier to debug and fine-tune ML applications by providing primitives that behave identically in user space and in kernel space. We believe that a library like KMLib could enable numerous ML based applications targeting operating systems and help us to rethink how to design adaptive and self-configured operating systems.

2 BACKGROUND AND RELATED WORK

While mainstream machine learning libraries like Tensor-Flow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019) has gained widespread use in research and production, there have also been several attempts to build machine learning libraries to address specific needs. Embedded Learning Library (ELL) (ELL) by Microsoft is one example, targeting embedded devices. TensorFlow Lite(TensorFlow Lite) by Google is a library for running machine learning applications on resource constrained devices. For using ML to improve operating systems, there has been several proposals (Zhang & Huang, 2019).

Researchers have investigated to tune file system parameters (Cao et al., 2018). Because this work performs the optimization in an offline manner, it is not designed to adapt to workload changes. Another work has attempted to improve I/O schedulers by predicting whether the I/O request meets the deadline or not (Hao et al., 2017). But, the predictions for I/O request deadlines were based on the result of a linear regression model that is trained on synthetically generated data in an offline manner. These examples suggest that having a machine learning library that works in kernel can help to build adaptive operating system components.

3 MACHINE LEARNING LIBRARY FOR OPERATING SYSTEMS

3.1 Machine Learning Library Design

Overview. There are several points and design choices worth mentioning regarding our machine learning library that will power ML applications in kernel space. First, the lack of access to standard math floating-point functions in the kernel means we have to implement nearly all math functions (including common functions such as *pow* and *log*) ourselves. Second, following the design choice seen in numerous mainstream deep-learning libraries (Abadi

et al., 2016; Paszke et al., 2019), we decided on a common tensor-like representation for matrices and model parameters. Functionality for manipulating matrices, such as matrix addition-multiplication and l_2 norm has also been implemented as part of the library. Third, neural networks are represented as a collection of layers, each of which implement `forward()` for forward propagation and `backward()` for backward propagation. Whenever a new layer is to be added to the library, `forward()` and `backward()` functions need to be implemented. In addition, our plan is to use lock-free data structures when implementing the layers to allow for parallel processing by breaking down the computation DAG when possible. Finally, neural networks implemented with this library will use an API similar to the individual layers, where `forward()` will facilitate forward propagation of input through the computation DAG, and `backward()` will apply backward propagation via chain-rule, using `backward()` method in each layer for computing the derivatives of the corresponding layer. In our design, the loss functions are treated like the other layers in terms of implementation. Our library will implement reverse-mode automatic differentiation to compute the gradients, which are then used to update the model weights using gradient-based learning algorithms such as gradient descent.

Our initial goal is to provide users with the implementations of most widely-used linear layers, such as fully-connected and convolutional (LeCun et al., 1998) layers, and widely-used non-linearities such as ReLU (Nair & Hinton, 2010) and Sigmoid, in addition to sequential models like LSTMs (Hochreiter & Schmidhuber, 1997). We also provide users with widely used losses such as cross entropy and mean square error. Users are able to extend the library with their own layers and loss functions by providing their own implementations.

Adapting to new workloads. The ever-changing workloads of modern computing systems means that machine learning models developed to exploit patterns in any workload must be adaptive. This could be achieved by constantly training the model, which incurs extra computational costs and memory footprint. Hence, there is a trade-off between the power of adaptation and computational efficiency. For the low-dimensional and less challenging machine learning problems where convergence could be achieved after a small number of steps, one could employ a simple feedback mechanism to control the training schedule. The goal of this mechanism is to perform inference only when the performance is better than random guess by a pre-defined threshold. More formally, for a classification task we perform inference only when the classification accuracy over the last k batches is at least p_{margin} higher than the most frequent label in these k batches. k and p_{margin} are adjustable,

with implications on memory footprint, computational cost and higher stability.

While the simple mechanism described above could be effective in a low-dimensional problem where converging to reasonable performance is not likely to take significant amounts of computational power, high-dimensional and more challenging machine learning problems in kernel space could require taking into account other aspects of the problem. More specifically, learning the ever-changing workloads on an edge device is a multi-objective problem, where some of the objectives are more obvious, *i.e.,* computation time, memory footprint, and energy consumption. However, optimizing for these objectives while there are non-zero computation costs and memory footprints incurred by training and inference makes it necessary to consider multiple other factors. Ideally, one would like to deploy a machine learning system that spends the least amount of time training, using the smallest number of samples for training. Reducing the training time and samples could be achieved by borrowing ideas from few-shot learning (Wang & Yao, 2019) when applicable. The relatively high cost of training makes it necessary to avoid using samples that are not likely to improve model performance. This could be approach using ideas from active learning (Settles, 2009), where the learning is performed on a promising subset of the labeled data. Effective utilization of methods for both of these problems could result in models that spend the least amount of time in training and used more for inference.

3.2 Operating System Integration

Low-precision training. Computation overhead is one of our biggest concerns while designing KMLib. There are operating system tasks that must be completed in submicrosecond time and any extra latency for these tasks may cause timeouts and serious performance degradations. One of the ways to reduce computation overhead for KMLib is by using low-precision training techniques (Choi et al., 2019; De Sa et al., 2018; Gupta et al., 2015; Sa et al., 2017).

KMLib can support different data types for `tensor` structures and is also flexible to adapt custom data types. One of these data types is `float`. As we mentioned above, KMLib can work in both user and kernel spaces. But, there are some challenges in using floating-point operations in kernel space. It is well-known that floating-point operations are not allowed in the Linux kernel. One way to perform floating point operations in the kernel is to enable the x86 architecture's floating-point unit by calling `kernel_fpu_begin`. Once floating point operations are finalized, use of floating points can be disabled by calling `kernel_fpu_end`. (For KMLib ARM v8 integration, floating-point enable/disable functions are `kernel_neon_begin` and `kernel_neon_end`.) We

tried to minimize the size of the floating-point enabled code block, because the more time KMLib spends in a floating-point–enabled regions, the higher the chance of being context-switched to other tasks. When the floating-point unit is enabled, the kernel must save floating point registers on a context switch, and adds additional overheads.

We are working to support 16-bit and 8-bit wide fixed-point numbers (Wang et al., 2018) with KMLib. Low-precision training not only helps to reduce computational overhead but also lowers memory consumption, which is another critical point when we started designing KMLib. We now explain more how KMLib handles capping memory and computation overheads.
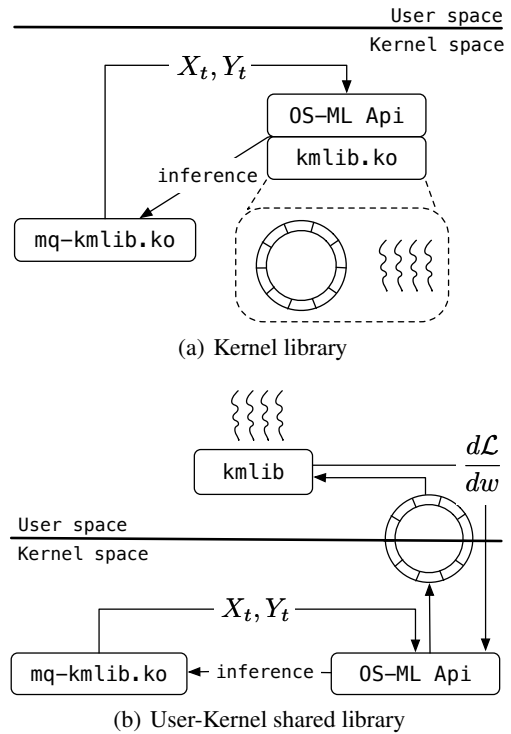

(a) Kernel library

(b) User-Kernel shared library

Figure 1. KMLib architectures: mq-kmlib.ko is a reference ML implementation for KMLib library and kmlib.ko refers to KMLib kernel space library.

Computation and memory capping. KMLib is designed to create as little as possible interference in the running system. KMLib is capable of training and inference operations while the underlying operating system is running. KMLib offloads the training computation to library threads to reduce interference. The only interference that KMLib adds is to save the input data and the predictions for training.

We used lock-free circular buffers to store training data. Users can configure the size of the circular buffers. Circular buffers have two running modes: blocking and dropping. The blocking mode helps the user to process every single

input piece data, but if the frequency of computation requests is high, this blocking mode might add extra overhead by blocking additional inputs from being processed. The dropping mode overruns unprocessed input data: it does not add extra overhead, but KMLib then loses data, which may hurt training quality. Using these features, the user can cap memory overhead based on their ML application needs.

The computational overhead of training varies based on the complexity of the learning model. We designed offloading training computation to KMLib library threads, but there are other challenges of partitioning computation DAG. Even though KMLib uses lock-free data structures to reduce multi-threaded communication and synchronization overhead, there might be dependencies in the computational DAG, which might cause latencies. That is why we also allow the user to choose how many threads can be used for (i) training and (ii) inference. All these features that related to offloading training/inference computation can be disabled and can be done in the original thread context as well.

User space vs. kernel space. The first question that comes to mind is why we started implementing a machine learning library from scratch for optimizing operating system tasks, rather than using a well-known user space library with data collected from the operating system. It is possible to collect data from the operating system and feed into user space ML implementations. But, there are challenges with that approach. For example, offloading training and inference should be running sub-microsecond because of the nature of operating system tasks. KMLib can be deployed in two different modes: (i) kernel mode (Figure 1(a)) and (ii) kernel-user memory mapped shared mode (Figure 1(b)) . In kernel mode, both training and inference happens in the kernel space. In kernel-user memory mapped shared mode, KMLib collects data from the kernel space and trains using user-space threads. For the inference, KMLib still runs the operations in kernel space to reduce the latency. We are using user-kernel shared lock-free circular buffers (Desnoyers & Dagenais, 2012) for collecting training data. But, KMLib threads can drain training request only when it gets scheduled because KMLib threads are working in a polling manner. We continue improving the user-space approach because we believe that it improves developer productivity, and developing, debugging, and testing learning models is much easier in user space than developing in the kernel.

4 EVALUATION

We developed a sample application of KMLib to fine-tune `mq-deadline` I/O scheduler. To predict whether the I/O request will meet the deadline or not, we train a linear regression model. The regression model predicts issue time for a given I/O request using normalized block number and ordinalized operation type as features. The predicted issue time is then thresholded to predict whether the I/O request should be early-rejected or not. We hypothesize that this should reduce the overall latency.

We have conducted the experiments on QEMU with I/O throttling running on Intel(R) Core(TM) i7-7500U and 8GB RAM and Intel SSD(256GB). We use our modified version of Linux Kernel v4.19.51+ for all experiments.

For the workload generation, we ran the FIO (FIO) micro-benchmark which is configured to perform random read and write operations with 4 threads on a 1GB dataset. Each experiment is executed on a fresh QEMU instance. We cloned the `mq-deadline` I/O scheduler as `mq-kmlib` and integrated it with KMLib. We made three key changes in the `mq-kmlib` I/O scheduler compared to `mq-deadline`: (i) In the `dd_init_queue` function, we inserted initialization code fragments to set the learning rate, batch size, momentum, and number of features to learn. Initial weights are also set randomly here. (ii) In the `__dd_dispatch_request` function, we call the functions that collect X_t and Y_t and perform the training steps. (iii) In the `dd_insert_request` function, we invoke an inference function; and based on the prediction we decide whether to early-reject the I/O request or not.

We observed that the thresholded regression output could predict with an accuracy of 74.62% whether the I/O requests miss the deadline or not: this reduced the overall I/O latency by 8%, a promising result given that I/O is so much slower than memory or CPU (and hence I/O should be the first to optimize). Our test involved a single synthetic workload that does not cover a large number of use cases, and our performance may not generalize to other workloads. Further, the emulated environment provided by QEMU may not represent a realistic use case, due to artificial throttling in QEMU. This is why our next step would be to investigate if these results generalize to other workloads under more realistic conditions (e.g., physical machines). We are also planning to apply machine learning models to other storage stack components like the page cache.

We wrote nearly 3,000 lines of C/C++ code (LoC). Because the current set of machine learning tools we have implemented is small, the memory footprint size of the KMLib user-space library is just 96KB, and the size of the KMLib kernel module is only 804KB. However, we expect these numbers to increase as additional functionality is implemented.

5 CONCLUSION

Adapting operating system components to running workloads and hardware has been done by tuning parameters or changing the critical data structure properties empirically.

We have proposed that *lightweight* machine learning approaches may help to solve these problems. Our preliminary evaluation show some promising results. Our plan is to expand on the work, apply it to other OS components, and evaluate and optimize the ML library for a wide range of workloads.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 265–283, 2016.

Cao, Z., Tarasov, V., Tiwari, S., and Zadok, E. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pp. 893–907, 2018.

Choi, J., Venkataramani, S., Srinivasan, V., Gopalakrishnan, K., Wang, Z., and Chuang, P. Accurate and efficient 2-bit quantized neural networks. In *Proceedings of the 2nd SysML Conference*, 2019.

De Sa, C., Leszczynski, M., Zhang, J., Marzoev, A., Aberger, C. R., Olukotun, K., and Ré, C. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

Desnoyers, M. and Dagenais, M. R. Lockless multi-core high-throughput buffering scheme for kernel tracing. *Operating Systems Review*, 46(3):65–81, 2012.

ELL. Embedded Learning Library (ELL), January 2020. https://microsoft.github.io/ELL/.

FIO. Flexible I/O Tester, January 2020. https://fio.readthedocs.io/en/latest/.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pp. 1737–1746, 2015.

Hao, M., Li, H., Tong, M. H., Pakha, C., Suminto, R. O., Stuardo, C. A., Chien, A. A., and Gunawi, H. S. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware OS interface. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pp. 168–183, 2017.

Hashemi, M., Swersky, K., Smith, J. A., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., and Ranganathan, P. Learning memory access patterns. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pp. 1924–1933, 2018.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 489–504. ACM, 2018.

Kraska, T., Alizadeh, M., Beutel, A., Chi, E. H., Kristo, A., Leclerc, G., Madden, S., Mao, H., and Nathan, V. Sagedb: A learned database system. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pp. 807–814, 2010.

Negi, A. and Kumar, P. K. Applying machine learning techniques to improve linux process scheduling. In *TENCON 2005-2005 IEEE Region 10 Conference*, pp. 1–6. IEEE, 2005.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019.

Sa, C. D., Feldman, M., Ré, C., and Olukotun, K. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pp. 561–574, 2017.

Settles, B. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

Shi, Z., Huang, X., Jain, A., and Lin, C. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pp. 413–425, 2019.

Smith, W., Foster, I. T., and Taylor, V. E. Predicting application run times using historical information. In *Job Scheduling Strategies for Parallel Processing, IPPS/SPDP'98 Workshop, Orlando, Florida, USA, March 30, 1998, Proceedings*, pp. 122–142, 1998.

TensorFlow Lite. TensorFlow Lite, January 2020. `https://www.tensorflow.org/lite`.

Wang, N., Choi, J., Brand, D., Chen, C., and Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pp. 7686–7695, 2018.

Wang, Y. and Yao, Q. Few-shot learning: A survey. *arXiv preprint arXiv:1904.05046*, 2019.

Zhang, Y. and Huang, Y. "Learned": Operating systems. *SIGOPS Oper. Syst. Rev.*, 53(1):40–45, July 2019.