CEML: a Coordinated Runtime System for Efficient Machine Learning on Heterogeneous Computing Systems

Jihoon Hyun, Jinsu Park, Kyu Yeun Kim, Seongdae Yu, and Woongki Baek(✉)

School of ECE, UNIST, Ulsan, Republic of Korea
{jhyun0812,jinsupark,kyuyeunk,sd3392,wbaek}@unist.ac.kr

Abstract. Heterogeneous computing is rapidly emerging as a promising solution for efficient machine learning. Despite the extensive prior works, system software support for efficient machine learning still remains unexplored in the context of heterogeneous computing. To bridge this gap, we propose CEML, a coordinated runtime system for efficient machine learning on heterogeneous computing systems. CEML dynamically analyzes the performance and power characteristics of the target machine-learning application and robustly adapts the system state to enhance its efficiency on heterogeneous computing systems. Our quantitative evaluation demonstrates that CEML significantly improves the efficiency of machine-learning applications on a full heterogeneous computing system.

1 Introduction

Heterogeneous computing is a promising solution for efficient machine learning [7]. Heterogeneous computing systems can effectively improve the efficiency of the target machine-learning application by concurrently executing its operations across the heterogeneous computing devices that exhibit different performance and power characteristics.

Prior works have extensively investigated the system software [4,7,9] and architectural support [5,6,10,16] for efficient machine learning. While insightful, the prior works have limitations in that they lack the runtime support for controlling all the heterogeneous computing devices in a coordinated manner [4,7,9] and/or require intrusive hardware modifications, making it difficult to apply them to existing commodity computer systems [5,6,10,16].

To bridge this gap, this work proposes CEML, a coordinated runtime system for efficient machine learning. CEML dynamically analyzes the performance and power characteristics of the target machine-learning application and generates the accurate performance and power estimators without requiring any per-application offline profiling. Guided by its performance and power estimators, CEML robustly finds the efficient system state and accordingly configures the underlying heterogeneous computing system to significantly improve the efficiency of the target application.

M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 781–795, 2018.
https://doi.org/10.1007/978-3-319-96983-1_55

Specifically, this paper makes the following contributions:

- We propose CEML, a coordinated runtime system for efficient machine learning on heterogeneous computing systems. CEML consists of the estimators that accurately estimate the performance and power consumption of the target machine-learning application for a system state of interest. The runtime manager of CEML explores the system state space, determines the efficient system state, and runs the target application with the efficient system state to significantly improve its efficiency in terms of the user-specified optimization metric (e.g., energy optimization, performance maximization under the power limit). To the best of our knowledge, CEML is the first runtime system that holistically controls all the heterogeneous computing devices for efficient machine learning.
- We implement a prototype of CEML. Since the CEML prototype is implemented as a user-level runtime system, it requires no modification to the underlying OS kernel or GPU device driver. The CEML prototype implements two search algorithms (i.e., the local and exhaustive search algorithms), each of which explores the system state space with different coverage and runtime overheads.
- We quantify the effectiveness of CEML using various machine-learning applications on a full heterogeneous computing system. Through quantitative evaluation, we demonstrate that CEML consumes significantly less energy (e.g., 30.8% less on average) than the baseline version that uses the maximum device frequencies, which is a commonly-used configuration in heterogeneous computing. We also show that the energy efficiency of CEML is comparable with the static best version, which requires extensive offline profiling across the applications.

2 Background and Motivation

2.1 Heterogeneous Computing

A heterogeneous computing system comprises multiple computing devices that show functional and performance/power heterogeneity. Heterogeneous computing devices exhibit the functional heterogeneity in the sense that they implement different instruction-set architectures and the performance/power heterogeneity in that they have different performance and power characteristics.

In this work, we assume that the underlying heterogeneous computing system is equipped with a single-chip heterogeneous application processor that consists of a multi-core CPU and a multi/many-core GPU. We also assume that the CPU and GPU communicate through the main memory. This architectural configuration is widely used in various computing domains including the embedded computing domain [1,2].

We assume that the CPU, GPU, and memory in the underlying heterogeneous computing system provide N_{f_C}, N_{f_G}, and N_{f_M} voltage and frequency (V/F) levels. The V/F level of each device can be dynamically controlled in software, similarly to commodity embedded systems [1]. A *system state* is defined

as a tuple of the device frequencies (i.e., (f_C, f_G, f_M)). The *system state space* is then defined as the set of all the possible system states.

2.2 The TensorFlow Machine-Learning System

TensorFlow is a widely-used machine-learning system [4]. TensorFlow allows for programmers to express their machine-learning algorithms as dataflow graphs. A *dataflow graph* mainly consists of tensors and operations. *Tensors* are multi-dimensional arrays, whose elements have one of the basic primitive data types such as `int32` or `float32`. An *operation* takes zero or more input tensors and produces zero or more output tensors [4].

When all the input tensors for an operation are produced, the operation becomes ready to be executed. The TensorFlow scheduler schedules the operation on one of the computing devices in the underlying heterogeneous computing system. The scheduling decision is made based on various factors such as the computational complexity of the operation, the utilization of each computing device, and the scheduling hint provided by the programmer [4]. Independent operations can be executed across the computing devices in a concurrent manner. One of the main design goals of the TensorFlow scheduler is to maximize the utilization of all the computing devices by concurrently executing as many independent operations as possible across the computing devices.

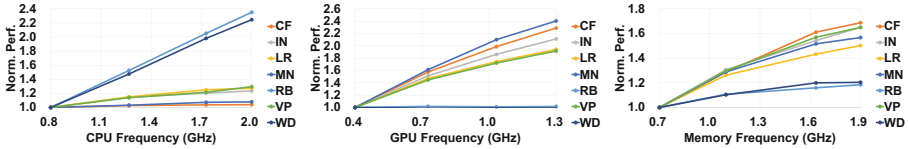
In this work, we focus on the efficiency optimization of the training phase of machine-learning applications. In each *training epoch* (or epoch), a machine-learning application iterates all the training data to train its model. In each *training step* (or step), the machine-learning application processes a batch of the training data. For instance, if 20,000 images are used as the training data and a batch size of 10 is used, an epoch consists of 2,000 steps.

We assume that the target machine-learning application is implemented as a TensorFlow application. While we evaluate the effectiveness of CEML using TensorFlow, we believe that the design of CEML is sufficiently generic to be readily applicable to other widely-used machine-learning platforms such as Caffe [9].

We have implemented a user-level, low-overhead API similar to the Application Heartbeats API [8] and instrumented each of the evaluated benchmarks (Sect. 3) with the API to make it generate a heartbeat every time it finishes a predefined number of steps. CEML employs the heartbeat data to dynamically track the current performance of the target machine-learning application.

2.3 Need for Coordinated Runtime Support

Machine-learning applications exhibit widely different performance and power characteristics on heterogeneous computing systems. To illustrate this, Figs. 1 and 2 show the performance and power characteristics of the seven machine-learning benchmarks (i.e., `CF`, `IN`, `LR`, `MN`, `RB`, `VP`, and `WD`) on the target heterogeneous computing system evaluated in this work (see Sect. 3 for details). We observe the following data trends.



(a) CPU frequency (b) GPU frequency (c) Memory frequency

Fig. 1. Performance characteristics of the machine-learning applications

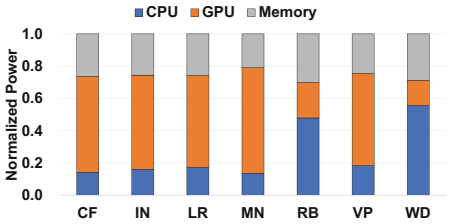


Fig. 2. Power characteristics at the maximum device frequencies

First, the performance sensitivity of each benchmark is widely different to device frequencies. For instance, the performance of `CF` is highly sensitive to the GPU frequency but insensitive to the CPU frequency, whereas the performance of `WD` is highly sensitive to the CPU frequency but insensitive to the GPU frequency.

Second, the power consumption of each device in the heterogeneous computing is widely different across the evaluated benchmarks. For example, the GPU consumes significantly more power than the other devices with `CF`. In contrast, with `WD`, the CPU is the device that consumes the highest power among the three devices (i.e., CPU, GPU, and memory).

These data trends show that static approaches require the offline performance and power profile data for every application to achieve high efficiency, which is nearly infeasible. To summarize, this case study clearly demonstrates the need for a coordinated runtime system that efficiently analyzes the performance and power characteristics of the target machine-learning application at runtime, robustly generates the accurate performance and power estimation models, and effectively manages all the devices in the underlying heterogeneous computing system to significantly enhance the overall efficiency without extensive per-application offline profiling.

3 Experimental Methodology

For all the experiments performed in this work, we use a full heterogeneous computing system, the NVIDIA Jetson TX2 embedded development board [1]. In this work, we employ the dual-core Denver processor based on the ARMv8-A architecture and the 256-core NVIDIA Pascal GPU equipped in the heterogeneous

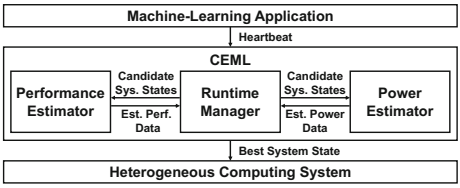


Fig. 3. Overall architecture of CEML

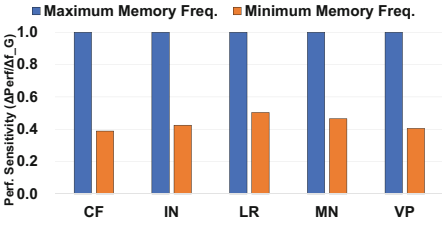


Fig. 4. Performance interaction between GPU and memory

computing system. The evaluated frequency ranges of the CPU, GPU, and memory are 0.81–2 GHz, 0.42–1.3 GHz, and 0.67–1.87 GHz, respectively. The heterogeneous computing system includes sensors, each of which periodically samples the power consumption of the CPU, GPU, or memory. We use the sensor data to generate the power estimation model of CEML and measure the power consumption of the target machine-learning application. The heterogeneous computing system is installed with Ubuntu 16.04 and Linux kernel 4.4.38.

We use seven machine-learning benchmarks (i.e., CIFAR-10 (`CF`), ImageNet (`IN`), Learning to Remember Rare Events (`LR`), MNIST (`MN`), REBAR (`RB`), Video Prediction (`VP`), and Wide & Deep (`WD`)), which are available in the official TensorFlow code repository [3]. We configure `CF`, `IN`, `LR`, `MN`, `RB`, `VP`, and `WD` to run 4000, 2000, 1000, 30000, 10136, 1000, and 32550 training steps, respectively.

4 Design and Implementation

CEML comprises the three main components – (1) the performance estimator, (2) the power estimator, and (3) the runtime manager. Figure 3 illustrates the overall architecture of CEML.

The main design principles applied to CEML are as follows. First, CEML controls the V/F level of each device in the underlying heterogeneous computing system in a coordinated manner to significantly improve the efficiency of the target machine-learning application. Second, CEML is designed as a versatile system in that it can support various optimization scenarios (e.g., energy optimization, performance maximization under the power limit). Third, CEML eliminates the need for per-application offline profiling. Fourth, the online profiling and adaptation functionalities of CEML are designed and implemented in a lightweight manner to minimize the potential runtime overheads.

4.1 Performance Estimator

The *performance estimator* estimates the performance of the target machine-learning application for a system state of interest. Specifically, the performance of the target machine-learning application is defined as the *training steps performed per second*.

We first investigate the performance sensitivity of various machine-learning applications to device frequencies to guide the design of the performance estimator. As shown in Fig. 1, the performance of machine-learning applications is (largely) linearly proportional to the frequency of each device when the frequencies of other devices are fixed (at the maximum frequency), which is intuitive.

Further, the performance sensitivity to the frequency of a device varies when the frequencies of the other devices change. For instance, as shown in Fig. 4, the performance of the GPU-intensive benchmarks becomes less sensitive to the GPU frequency with the decreasing memory frequency because the memory gradually becomes the overall performance bottleneck as its frequency decreases. This indicates that there is performance interaction between devices.

Based on the aforementioned observations, the performance estimator employs Eq. 1 to estimate the performance of the target application for a system state of interest (i.e., (f_C, f_G, f_M)). Equation 1 has a linear term for each device frequency to model the linear relationship between the performance and the device frequency. Equation 1 also has an interaction term for each device pair (e.g., $\alpha_{C,G}$ for f_C and f_G) to model the performance interaction between the pair of devices.

$$\begin{aligned} Perf = &\alpha_C \cdot f_C + \alpha_G \cdot f_G + \alpha_M \cdot f_M + \alpha_{C,G} \cdot f_C \cdot f_G + \\ &\alpha_{G,M} \cdot f_G \cdot f_M + \alpha_{M,C} \cdot f_M \cdot f_C + \beta \end{aligned} \tag{1}$$

To compute the coefficients in Eq. 1, seven performance data samples are required because there are seven unknown coefficients. Each performance data sample is collected with a different system state. Section 4.3 discusses how CEML collects the performance data samples to generate the performance estimation model at runtime.

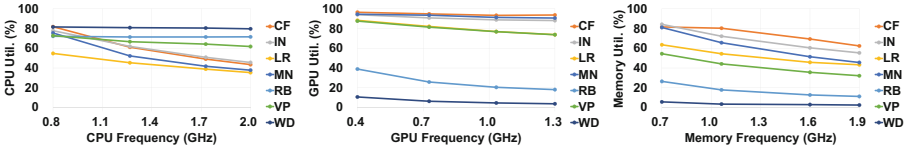
4.2 Power Estimator

The *power estimator* estimates the power consumption of the target machine-learning application for a system state of interest. In line with prior works, the power estimator assumes that the power consumption of each device is proportional to the device utilization because it is simple and accurate [14,17].

Specifically, CEML employs Eq. 2 to estimate the power consumption of the device D (i.e., CPU, GPU, or memory) when the device frequency is set to f_D. We experimentally determine the regression coefficients (i.e., ϵ_{D,f_D} and ζ_{D,f_D}) for each device based on the offline profiling using the stress benchmarks that we have developed. Each of the stress benchmarks is designed and implemented to stress the CPU, GPU, or memory.

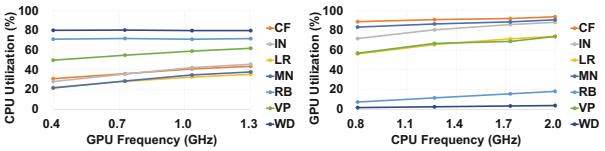
$$P_{D,f_D} = \epsilon_{D,f_D} \cdot U_D + \zeta_{D,f_D} \tag{2}$$

To estimate the power consumption of the target machine-learning application for a system state of interest, the power estimator estimates the utilization of each device. We first investigate the device utilization sensitivity to the device frequencies. Our experimental results show the following data trends.



(a) CPU frequency (b) GPU frequency (c) Memory frequency

Fig. 5. Device utilization sensitivity to its frequency



(a) CPU util. vs. GPU freq. (b) GPU util. vs. CPU freq.

Fig. 6. Device utilization sensitivity to the other device frequency

First, as shown in Fig. 5, the utilization of each device is (largely) linearly proportional to the device frequency when the frequencies of the other devices are fixed (at the maximum frequency). This is mainly because the device stays active for a shorter (or longer) time duration at higher (or lower) frequency by processing the assigned work faster (or slower).

Second, as shown in Fig. 6, the utilization of each device is (largely) linearly proportional to the frequency of the frequencies of the other devices. When the other devices run faster (or slower), they produce more (or less) data to be processed by the device per unit time, making its utilization higher (or lower).

We now discuss how the power estimator estimates the utilization of each device. For instance, the power estimator uses Eq. 3 to estimate the CPU utilization for a system state of interest (i.e., (f_C, f_G, f_M)).[1] In Eq. 3, the first-order and interaction terms are used to model the individual effect of each device and the interaction between devices, respectively.

$$U_C = \gamma_{C,C} \cdot f_C + \gamma_{C,G} \cdot f_G + \gamma_{C,M} \cdot f_M + \gamma_{C,CG} \cdot f_C \cdot f_G + \gamma_{C,GM} \cdot f_G \cdot f_M + \gamma_{C,MC} \cdot f_M \cdot f_C + \delta_C \quad (3)$$

To compute the coefficients in Eq. 3, seven utilization data samples are required because there are seven unknown coefficients. Section 4.3 discusses how CEML collects the utilization data samples to generate the power estimation model at runtime. Finally, the power estimator estimates the total power consumption of the underlying heterogeneous computing system by summing the estimated power consumption of all the devices (i.e., $P = P_C + P_G + P_M$).

[1] While omitted, the power estimator employs the equations similar to Eq. 3 to estimate the GPU and memory utilization.

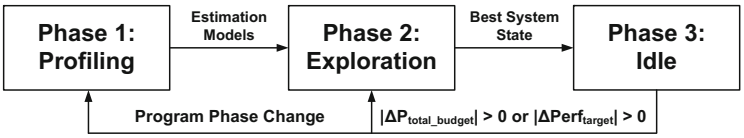


Fig. 7. Overall execution flow of the runtime manager

4.3 Runtime Manager

The *runtime manager* of CEML dynamically profiles the performance and power consumption data samples. It then builds the performance and power estimation models and determines the efficient system state that significantly enhances the efficiency of the target machine-learning application by exploring the system state space based on the estimation models.

The runtime manager mainly comprises the three phases – (1) profiling, (2) exploration, and (3) idle phases. Figure 7 shows its overall execution flow.

During the *profiling phase*, the runtime manager executes a small portion of the total training steps of the target machine-learning application with the following seven system states that cover a wide range of the device frequencies (in GHz) – (2, 1.3, 1.87), (1.42, 0.83, 0.67), (1.42, 0.42, 1.33), (0.81, 0.83, 1.33), (0.81, 0.42, 1.06), (0.81, 0.62, 0.67), and (1.11, 0.42, 0.67). Specifically, the runtime manager starts with the initial system state. When the runtime manager collects N heartbeats[2] generated by the target application, it stores the performance and utilization data, configures the system with the next one among the seven system states, and repeats the data collection process. When the performance and utilization data is collected with all the seven system states, the runtime manager proceeds with the exploration phase.

During the *exploration phase*, the runtime manager constructs the performance and power estimation models using the data collected during the profiling phase. Specifically, it computes all the coefficients in Eqs. 1 and 3 using the efficient equation solver that we have developed.

The runtime manager then explores the system state space to determine the system state, which is estimated to significantly improve the efficiency[3] of the target machine-learning application. We propose two search algorithms, each of which explores the system state space with different coverage and runtime overheads.

The first algorithm is the exhaustive search algorithm shown in Algorithm 1. It explores all the feasible system states in an exhaustive manner and selects

[2] In this work, N heartbeats contain the performance and utilization data collected during the execution of 1% of the total training steps for each benchmark.

[3] We use the generic term "efficiency" because CEML can be extended to perform optimizations using the metrics (e.g., energy-delay product) other than energy (i.e., $\frac{Power}{Performance}$ (Joules per training step)) by customizing the `estimateScore` function in Algorithms 1 and 2.

Algorithm 1. The exhaustive search function

```
1:  procedure exploreWithExhaustiveSearch
2:      bestState ← getInitialState()
3:      bestScore ← estimateScore(bestState)
4:      for f_C ∈ F_C
5:          for f_G ∈ F_G
6:              for f_M ∈ F_M
7:                  cState ← (f_C, f_G, f_M)
8:                  cScore ← estimateScore(cState)
9:                  if cScore > bestScore ∧ checkConstraint(cState)
10:                     bestState ← cState
11:                     bestScore ← cScore
12:                 end if
13:             end for
14:         end for
15:     end for
16:     setSystemState(bestState)
17: end procedure
```

the best system state, which is estimated to maximize the efficiency of the target machine-learning application without violating the user-specified constraint[4] such as the total power budget (Line 9). The time complexity of the exhaustive search algorithm is $O(N_{f_C} \cdot N_{f_G} \cdot N_{f_M})$.

While the time complexity of the exhaustive search algorithm is rather high, it may be still practically used for commodity heterogeneous computing systems. For example, the system state parameters of the heterogeneous computing system evaluated in this work are $N_{f_C} = 9$, $N_{f_G} = 10$, and $N_{f_M} = 6$. In this case, the total number of the candidate system states is 540, which can be explored in 176.4 microseconds (on average) on the evaluated heterogeneous computing system.

Since the exhaustive search algorithm has high time complexity, we also propose a local search algorithm that explores the system state space using a variant of the hill-climbing algorithm (Algorithm 2). Specifically, the local search algorithm starts with the initial system state. It estimates the efficiency of all the neighbor system states and selects the system state, which is estimated to achieve the maximum efficiency without violating the user-specified constraint (Line 5).[5] If the best neighbor state is estimated to be more efficient than the current state, it selects the best neighbor state as the next system state to

[4] We assume that there are one or more system states (in the system state space) that satisfy the user-specified constraint. The `getInitialState` function returns one of such system states.

[5] If there is no neighbor state that satisfies the user-specified constraint, the `getBestNeighborState` function returns `invalidState`. If the input parameter is set to `invalidState`, the `estimateScore` function returns the minimum score.

Algorithm 2. The local search function

```
1: procedure EXPLOREWITHLOCALSEARCH
2:     bestState ← getInitialState()
3:     bestScore ← estimateScore(bestState)
4:     while true
5:         cState ← getBestNeighborState(bestState)
6:         cScore ← estimateScore(cState)
7:         if cScore > bestScore
8:             bestState ← cState
9:             bestScore ← cScore
10:        else
11:            break
12:        end if
13:    end while
14:    setSystemState(bestState)
15: end procedure
```

transition and continues the search process (Lines 7–9). Otherwise, it terminates the search process (Line 11).

Once the best system state is selected by the search algorithm, the runtime manager accordingly configures the system to significantly enhance the efficiency of the target machine-learning application (Line 16 in Algorithm 1 and Line 14 in Algorithm 2). The runtime manager then transitions into the idle phase.

During the *idle phase*, CEML keeps monitoring the target machine-learning application to detect its phase changes without performing any adaptation activities. Specifically, CEML periodically collects the heartbeats from the target application and computes the differences between consecutive data samples to detect a program phase change. When detecting a program phase change, CEML terminates the idle phase and re-triggers the adaptation process to determine a new efficient system state.

Further, CEML keeps monitoring the underlying system to detect any change in the total power budget or performance target. When detecting a change, CEML immediately triggers the re-adaptation process to discover an efficient system state for the new constraint.

5 Evaluation

We quantify the effectiveness of CEML. Specifically, we aim to investigate the following – (1) the estimation accuracy, (2) the energy efficiency, (3) the effectiveness of re-adaptation, and (4) the performance overheads.

We first investigate the accuracy of the performance and power estimators of CEML. To quantify the accuracy of the estimators, we generate 25 test datasets for each benchmark by executing each benchmark with 25 different system states and compute the average estimation error across all the test datasets.

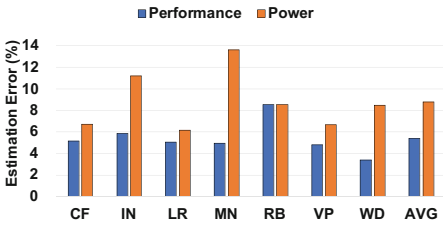


Fig. 8. Accuracy of the estimators

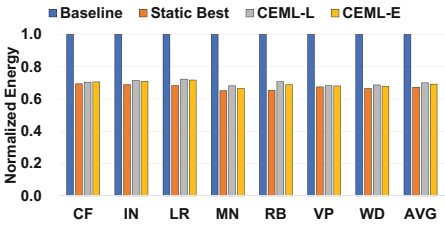


Fig. 9. Energy consumption

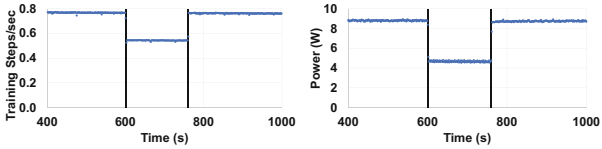
Figure 8 shows the average performance and power estimation errors. We observe that the performance and power estimators achieve high estimation accuracy. Specifically, the average estimation errors of the performance and power estimators are 5.4% and 8.8% across all the benchmarks. The use of the first-order and interaction terms in the performance and power estimators effectively models the linear and interactive effects of each device in the underlying heterogeneous computing system, achieving high estimation accuracy.

We now investigate the effectiveness of CEML in terms of energy consumption (i.e., Joules per training step). We evaluate four versions for each benchmark. The baseline version executes each benchmark at the maximum device frequencies. The static best version selects the best frequency of each device based on the extensive offline experiments (i.e., 32 system states for each benchmark). The `CEML-L` and `CEML-E` versions are managed by CEML using the local and exhaustive search algorithms, respectively.

Figure 9 shows the energy consumption of each version of the benchmarks, normalized to the baseline version. The rightmost bar shows the geometric mean of each version.

First, the CEML versions significantly reduce the energy consumption across all the evaluated machine-learning benchmarks. For instance, `CEML-E` consumes 30.8% less energy (on average) than the baseline version. The baseline version achieves low energy efficiency because it executes the target machine-learning application at the maximum device frequencies without considering the performance and power characteristics of the target machine-learning application. In contrast, CEML robustly finds the efficient system state based on its performance and power estimators and accordingly configures the system, consuming significantly less energy than the baseline version.

Second, the CEML versions achieve the energy efficiency similar to the static best version. For instance, the energy consumption of `CEML-E` is 2.9% higher (on average) than the static best version. The CEML versions consume slightly more energy than the static best version because it executes the target machine-learning application with suboptimal system states during the profiling phase (e.g., `RB`) and finds a slightly less efficient system state due to the estimation errors with some of the evaluated benchmarks (e.g., `IN` and `LR`). Nevertheless, our experimental results demonstrate the effectiveness of CEML in that the CEML



(a) Performance (b) Power consumption

Fig. 10. Effectiveness of re-adaptation

versions achieve the energy efficiency comparable with the static best version without requiring any extensive per-application offline profiling.

Third, `CEML-E` achieves slightly higher energy efficiency than the `CEML-L` (i.e., 2.6% on average). This is mainly because the local search algorithm of `CEML-L` may converge to a less efficient state (e.g., `MN`). However, `CEML-L` achieves the energy efficiency comparable with `CEML-E` across all the evaluated benchmarks, demonstrating the potential for `CEML-L` in that its local search algorithm has significantly lower average-case time complexity than the exhaustive search algorithm of `CEML-E`.

To investigate the effectiveness of the re-adaptation functionality of CEML, we design a case study in which the total power budget allocated to the underlying heterogeneous computing system changes during the execution of the `LR` benchmark with CEML. In this case study, CEML is configured to maximize the performance of `LR` while satisfying the power constraint.

Figure 10 shows the runtime behavior of CEML. Initially, the benchmark runs with a high power budget. Since the power budget is sufficient, CEML runs the benchmark at the maximum device frequencies while satisfying the power constraint. At $t = 600.1$, the total power budget changes to a low power budget. CEML robustly detects the total power budget change and adapts to the new system state that is efficient (i.e., similar performance to the static best version) for the low power budget, guided by its performance and power estimators. At $t = 760.1$, the total power budget changes back to the high power budget. Again, CEML robustly detects the total power budget change and accordingly performs adaptations to find the efficient system state for the new total power budget.

Finally, we quantify the performance overheads of CEML. Our experimental results demonstrate that CEML incurs insignificant performance overheads. Specifically, the CPU utilization of CEML is 1.0% on average, which is low. In addition, the system state exploration times with the `CEML-L` and `CEML-E` versions are 7.2 and 176.4 microseconds on average, which are insignificant.

In summary, our quantitative evaluation shows that CEML is effective in the sense that it consumes significantly less energy than the baseline version, achieves the energy efficiency similar to the static best version, robustly adapts to the external events such as total power budget changes, and incurs small performance overheads.

6 Related Work

Prior works have extensively investigated the architectural and system software techniques to improve the efficiency of heterogeneous computing systems [11–15,17]. While insightful, the prior works manage a subset of heterogeneous computing devices (i.e., CPU [11,13,17], CPU and GPU [12,14], GPU and memory [15]) with multithreaded and gaming workloads.

Our work significantly differs in that it investigates the performance and power characteristics of machine-learning applications with various device frequencies, presents the accurate performance and power estimators, proposes a coordinated runtime system that robustly controls all the devices (i.e., CPU, GPU, memory) in the underlying heterogeneous computing system, and demonstrates the effectiveness of CEML using a full heterogeneous computing system.

Prior works have proposed the system software support for efficient machine learning [4,7,9]. The prior works mainly focus on the design and implementation of parallel and distributed programming platforms for machine learning with support for task scheduling [4,7,9]. In contrast, our work investigates the coordinated runtime support that robustly manages the hardware resources in heterogeneous computing systems for efficient machine learning.

Prior works have investigated the design and implementation of hardware accelerators for machine learning [5,6,10,16]. While effective, the prior works cannot be directly applied to existing commodity systems because they require intrusive hardware modifications. Our work differs in that CEML is designed and implemented as a coordinated runtime system to enable efficient machine learning on commodity heterogeneous computing systems. When the hardware accelerators for machine learning become widely available in upcoming commodity systems, coordinated runtime systems such as CEML can be effectively used to robustly manage a variety of heterogeneous computing devices including the hardware accelerators.

7 Conclusions

In this paper, we propose CEML, a coordinated runtime system for efficient machine-learning on heterogeneous computing systems. CEML dynamically analyzes the performance and power characteristics of the target machine-learning application and adapts the system state to enhance its efficiency on heterogeneous computing systems. Our experimental results demonstrate that CEML consumes significantly less energy than the baseline version that employs the maximum device frequencies and achieves the energy efficiency comparable with the static best version that requires extensive per-application offline profiling. As future work, we plan to apply and extend our proposed techniques for efficient machine learning in heterogeneous distributed computing environments.

Acknowledgements. This research was partly supported by the National Research Foundation of Korea (NRF-2016M3C4A7952587, PF Class Heterogeneous High Performance Computer Development), Basic Science Research Program through the National

Research Foundation of Korea (NRF-2018R1C1B6005961), and Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-16-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development).

References

1. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html
2. http://www.samsung.com/semiconductor/products/exynos-solution/application-processor/EXYNOS-5-OCTA-5422
3. https://github.com/tensorflow/models
4. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016) (2016)
5. Chen, Y.-H., Emer, J., Sze, V.: Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. In: Proceedings of the 43rd International Symposium on Computer Architecture (2016)
6. Han, S., et al.: EIE: efficient inference engine on compressed deep neural network. In: Proceedings of the 43rd International Symposium on Computer Architecture (2016)
7. Hauswald, J., et al.: DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture (2015)
8. Hoffmann, H., Eastep, J., Santambrogio, M.D., Miller, J.E., Agarwal, A.: Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In: Proceedings of the 7th International Conference on Autonomic Computing (2010)
9. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia (2014)
10. Jouppi, N.P., et al.: In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture (2017)
11. Muthukaruppan, T.S., Pricopi, M., Venkataramani, V., Mitra, T., Vishin, S.: Hierarchical power management for asymmetric multi-core in dark silicon era. In: Proceedings of the 50th Annual Design Automation Conference (2013)
12. Park, J., Baek, W.: RCHC: a holistic runtime system for concurrent heterogeneous computing. In: 2016 45th International Conference on Parallel Processing (ICPP) (2016)
13. Park, J., Baek, W.: HAP: a heterogeneity-conscious runtime system for adaptive pipeline parallelism. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 518–530. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_38
14. Pathania, A., Irimiea, A.E., Prakash, A., Mitra, T.: Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In: Proceedings of the 52nd Annual Design Automation Conference (2015)
15. Sethia, A., Mahlke, S.: Equalizer: dynamic tuning of GPU resources for efficient execution. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (2014)

16. Song, L., Wang, Y., Han, Y., Zhao, X., Liu, B., Li, X.: C-brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization. In: Proceedings of the 53rd Annual Design Automation Conference (2016)
17. Yun, J., Park, J., Baek, W.: HARS: a heterogeneity-aware runtime system for self-adaptive multithreaded applications. In: Proceedings of the 52nd Annual Design Automation Conference (2015)