Managing Computational Resources with Machine Learning Policies

Alexandros Patras,
Nikolaos Bellas, Christos D. Antonopoulos, Spyros Lalis
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece
{*patras, nbellas, cda, lalis*}*@uth.gr*

***Abstract*—Efficiently allocating computational resources for deep learning applications is a key challenge in cloud and edge environments. These applications can perform inference on multi-core CPUs, hardware accelerators, or resource constrained Edge devices, allowing for scalable performance and energy efficiency. However, deciding the optimal configuration is a complex undertaking. This paper proposes an adaptive, lightweight scaling engine for energy-efficient deep learning inference on a dual-socked cloud node with multicore CPUs. Using a custom reinforcement learning algorithm, it continuously learns to identify the most efficient execution configuration based on the neural network characteristics and available system resources.**

1. Introduction

Machine learning inference workloads vary widely when executed on heterogeneous computing devices, differing in performance, power efficiency, communication overhead, and user/environmental constraints. For example, some components may run more efficiently on GPUs, FPGAs, or TPUs, while others benefit from CPU execution to minimize communication overhead. Inefficient scheduling across these resources can lead to missed deadlines and reduced efficiency. Additionally, modern ML models offer configuration options—such as different versions, quantization bitwidths, and pruning ratios—to balance inference latency and model accuracy.

This paper introduces reinforcement learning (RL) mechanisms to enable efficient ML inference on a datacenter compute node by dynamically allocating CPU cores to inference jobs to minimize power dissipation under various latency constraints. We demonstrate the feasibility of our approach by running a widely used workload on a real cloud node consisting of dual socket multicore CPUs.

2. System Design

2.1. Architecture

The system follows the Monitor-Analyse-Plan-Execute (MAPE) model for autonomic systems [1]. Our work adapts the MAPE model, by building a managing system comprising the following functional components, illustrated in Figure 1: i) a *state* component responsible for assimilating input from external sources, such as administrator policies, energy costs or even carbon intensity from the electricity grid, in addition to receiving job-related data and incorporating system state information obtained through telemetry subsystems. This data describes the state of the platform and the applications. ii) A *machine learning* agent tasked with data analysis and decision making. iii) A *node manager* that creates optimal plans based on the current resource state, aided by the output of the machine learning module. iv) *Action* adapters that implement the plan of task allocation and resource configuration by directly interfering with the underlying compute node.

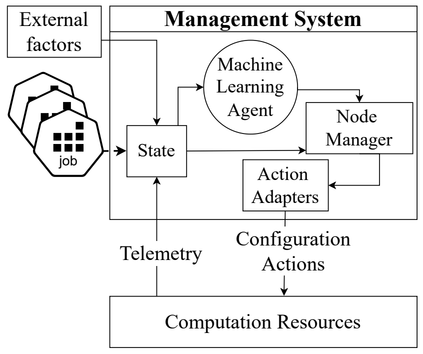

Figure 1. High-level architecture overview.

2.2. Conguration Action Space

The Node Manager in Figure 1 operates in the following three dimensions:

Task Scheduling: Determines the specific timing when each job will utilize the computational resources.

Resource Allocation: Specifies the computational resource assigned to each job and tackles the challenge of optimizing software/hardware accelerator co-scheduling.

Application Version: With multiple versions of the same application job, each with different performance, Quality of Results (QoR), or energy consumption, node manager selects the most suitable version, balancing the other two factors.

3. Machine Learning Policy

In our exploration of optimizing resource allocation, we opted to use a reinforcement learning (RL) model. RL is a machine learning approach focused on learning a sequence of decisions to maximize cumulative rewards [2]. The primary objective of the RL agent is to determine an optimal or near-optimal task allocation. To accomplish this goal, a suitable **reward** function is necessary to direct the RL agent in achieving the application requirements without over-provisioning and exceeding (beyond what is minimally required) the allocated computation resources and power consumption.

In Equation 1, we present the reward utilized in the proof-of-concept evaluation to train the RL agent, based on relevant prior work [3].

$$reward = \begin{cases} -100 & \text{if } A_M > A_T \\ -\sqrt{Power_C} & \text{if } A_M \leq A_T \end{cases} \tag{1}$$

The A_M term is the average inference latency (per image) across the batch of the processed images. A_T corresponds to the application target set by the application job. The RL agent's primary goal is to allocate sufficient resources to meet this target; failing to do so results in a significant penalty. Once the target is met, the agent is rewarded based on the $Power_C$ term, representing the power consumed by the allocated resources. Lower power consumption incurs a greater reward.

The **state** of the RL agent includes a variety of node-level metrics captured via the telemetry system, specifically (i) the average CPU utilization for each logical core, (ii) the CPU latency to run ML inference potentially dividing this task across multiple CPU cores in both sockets, and (iii) the power dissipation per core to perform this task (Table 1). The **action** of the RL agent is to decide how many and which CPU cores will be allocated to run inference on a new batch of images.

4. Experimental Evaluation

To validate our design, we have developed a proof-of-concept RL agent, that was trained to determine an optimal ML task allocation across CPU cores using the reward function shown in Equation 1.

In this experiment, a containerized ResNet model [4], a pre-trained convolutional neural network for image classification, was used to process batches of images. The container ran for 60 seconds on a datacenter-grade compute node with dual AMD EPYC processors (64 physical cores, 128 logical cores). Real-time telemetry data were gathered and stored using OpenTelemetry [5], with parameters detailed in Table 1. Logical cores used ranged from 1 to 100.

In each run, different core count were allocated for the container. Figure 2 shows the execution time (left axis) and the dissipated power (right axis) assuming a target execution time of 7ms. The execution time is the average inference latency over multiple images. The inference latency decreases by utilizing more cores, increasing the power consumption at the same time, however, there is a point at around 20 cores where the latency does not improve further, but the power consumption continues to increase.

TABLE 1. TELEMETRY METRICS RECORDED.

Metric	**Values range [Units]**
Average CPU utilization for each logical core	0.00 - 1.00 [%]
Total Average CPU Power Consumption for both sockets	90.00 – 388.00 [Watts]
Application metric (Inference latency)	4.00 – 115.00 [ms]

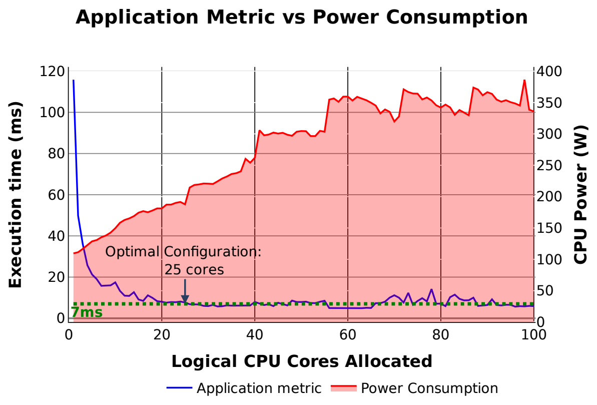

Figure 2. Telemetry metrics concerning the number of cores allocated. An example of a handpicked optimal configuration is shown with the arrow for an application target of 7ms (indicated by the green dashed line).

Using offline recorded telemetry data, we trained RL agents with varying application targets. Utilizing real-time telemetry data, online training is an option; however, the extensive training duration makes it impractical for initial training. Table 2 shows results for four application targets using the same dataset. After training each agent for 200,000 timesteps, optimal results were achieved in 75% of cases. The exception was the latency target of 5ms, which neared the CPU's performance limit across many configurations.

TABLE 2. OPTIMAL AND PREDICTED CONFIGURATION FOR THE CORE ALLOCATION.

Latency Target (ms)	**ML Predicted Configuration**	**Optimal Handpicked Configuration**
5	55	61
7	25	25
10	14	15
15	10	11

5. Conclusions & Future Work

The node level RL model presented in this paper determines an optimal CPU core allocation to optimize power dissipation and, at the same time, satisfy application latency constraints. In the future this model will be enhanced to incorporate several new features to be used by the RL agent for task allocation and core configuration: i) usage of hardware accelerators (GPUs, FPGAs), ii) thread affinity and simultaneous multithreading, and (iii) voltage/frequency scaling of the CPU cores and the hardware accelerators. Another potential direction for future research is applying the same reinforcement learning (RL) approach to different environments, such as varying hardware characteristics.

References

[1] J. Kephart and D. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: A Bradford Book, 2018.

[3] M. Han and W. Baek, “Herti: A reinforcement learning-augmented system for efficient real-time inference on heterogeneous embedded systems,” in *30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 90–102.

[4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[5] “Open telemetry,” https://opentelemetry.io.