# Learning *for* and *with* Efficient Computing Systems

*Submitted in partial fulfillment of the requirements for*

*the degree of*

*Doctor of Philosophy*

*in*

*Electrical and Computer Engineering*

Zhuo Chen

B.S., Electronics Engineering and Computer Science, Peking University

Carnegie Mellon University
Pittsburgh, PA

May 2019

# Acknowledgements

The past few years at CMU have been an invaluable life journey to me, and I sincerely appreciate all the support, patience and motivation from these brilliant and sincere people.

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Diana Marculescu, for her mentoring and support over the entire course of my pursuit of PhD. She encourages me to work on important problems, explore new ideas, collaborate with brilliant students, and can always point me in the right direction with her insightful mind and broad knowledge. I have been incredibly lucky to have such a wise and caring advisor to guide me over the years.

I would like to thank Prof. Radu Marculescu, Prof. Shawn Blanton and Dr. Jinwon Lee for being my thesis committee members and for their insightful comments and encouragement through this process.

My sincere thanks also goes to my collaborators: Prof. Radu Marculescu, Prof. Partha Pande, Dr. Ryan Kim, Mr. Dimitrios Stamoulis, Mr. Ruizhou Ding and Mr. Ting-Wu Chin for their outstanding research work and inspiring thoughts and ideas.

I would also like to thank my internship mentors: Dr. Jinwon Lee, Dr. Shankar Sadasivam, Dr. John Dorsey and Mr. Bryan Hinch, for having me on their teams and teaching me how to apply my knowledge to solve real-world problems.

Besides academic life, I do want to thank my friends who made my personal life full of sunshine. They are: Dr. Da-Cheng Juan, Dr. Ermao Cai, Mr. Guangshuo Liu and Mr. Ahmet Fatih Inci from EnyAC; Dr. Xuanle Ren, Dr. Shaolong Liu, Dr. Huang-Kai Peng, Mr. Zeye Liu, Ms. Qicheng Huang, Ms. Chenglei Fang, Mr. Chao Rong.

Thank you, Dr. Chieh Luo, Mr. Xiaoliang Li and Mr. Vincent Chung for being my great basketball teammates.

Thank you, Dr. Renzhi Liu, Dr. Liang Tang and Mr. Xi He for cooking and hosting parties during holidays.

Thank you, Dr. Yanfei Chen for being an incredible roommate for five years.

I would like to acknowledge the funding support from National Science Foundation (Grants CNS1564022, CCF1514206, CCF1314876), Samsung Electronics, Qualcomm Innovation Fellowship and Carnegie Mellon University for making my pursuit of research possible.

Last but not least, I sincerely thank my family, especially my then girlfriend and now wife, Dr. Nan Bi, for their unconditional love and support. All of these will not be possible without them.

# Abstract

Machine learning approaches have been widely adopted in recent years due to their capability of learning from data rather than hand-tuning features manually. We investigate two important aspects of machine learning methods, *i.e.*, (i) applying machine learning in computing system optimization and (ii) optimizing machine learning algorithms, especially deep convolutional neural networks, so they can train and infer efficiently.

As power emerges as the main constraint for computing systems, controlling power consumption under a given Thermal Design Power (TDP) while maximizing the performance becomes increasingly critical. Meanwhile, systems have certain performance constraints that the applications should satisfy to ensure Quality of Service (QoS). Learning approaches have drawn significant attention recently due to the ability to adapt to the ever-increasing complexity of the system and applications. In this thesis, we propose On-line Distributed Reinforcement Learning (OD-RL) based algorithms for many-core system performance improvement under both power and performance constraints. The experiments show that compared to the state-of-the-art algorithms, our approach: 1) produces up to **98%** less budget overshoot, 2) up to **23%** higher energy efficiency, and 3) **two orders of magnitude** speedup over state-of-the-art techniques for systems with hundreds of cores, while an improved version can better satisfy performance constraints. To further improve the sample-efficiency of RL algorithms, we propose a novel Bayesian Optimization approach to speed up reinforcement learning-based DVFS control by **37.4x while maintaining the performance of the best rule-based DVFS algorithm**.

Convolutional Neural Networks (CNNs) have shown unprecedented capability in visual learning tasks. While accuracy-wise CNNs provide unprecedented performance, they are also known to be computationally intensive and energy demanding for modern computer systems. We propose Virtual Pooling (ViP), a model-level approach to improve inference speed and energy consumption of CNN-based image classification and object detection tasks, with provable error bound. We show the efficacy of ViP through exten-

sive experiments. For example, ViP delivers **2.1x** speedup with less than 1.5% accuracy degradation in ImageNet classification on VGG-16, and **1.8x** speedup with 0.025 mAP degradation in PASCAL VOC object detection with Faster-RCNN. ViP also reduces mobile GPU and CPU energy consumption by up to **55%** and **70%**, respectively. We further propose to train CNNs with fine-grain labels, which not only improves testing accuracy but also the training data efficiency. For example, a CNN trained with fine-grain labels and only **40%** of the total training data can achieve higher accuracy than a CNN trained with the full training dataset and coarse-grain labels.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Challenges

While historically the major goal of processor designers was to gain better performance by continuously shrinking device size, adding more pipeline stages, and speeding up the clock frequency, the power wall was eventually reached and energy is now the main design constraint. The ever growing power consumption increases the burden of heat dissipation, lowers the chip reliability, and decreases battery life of mobile devices. As Dennard scaling breaks down, multi-core systems have become the solution to mitigate the high power problem. For highly parallel applications, multiple small cores with lower voltage and frequency can offer similar throughput as one large core at a much higher voltage and frequency. Indeed, a multi-core system consumes less power according to the $V^2f$ Scale Law [3]. While multi-core systems are now mainstream market products, the desire for higher performance is again pushing the envelope toward higher power consumption. In order to ensure the safety and reliability of multi-core systems, a Thermal Design Power (TDP) constraint is imposed on the system power consumption. As a result, improving performance under the TDP constraint has become one of the main directions in power/performance optimization. In addition to increasing the number of cores at design time, Dynamic Voltage Frequency Scaling (DVFS) is devel-

oped to save power at runtime. By being smart in tuning the Voltage and Frequency (VF) levels of the cores, on-chip computation can be performed in a more energy efficient manner. However, finding the optimal VF level assignment can be formulated as an integer linear programming problem which is NP hard [4]. Therefore, the exact solution cannot be implemented as an on-line algorithm, especially when the number of cores increases. Many algorithms have been proposed to find near-optimal solutions in polynomial time, however, they did not take the budget overshoot problem into consideration and may only be efficient for moderately sized multi-core systems rather than systems with hundreds of cores. Therefore, Reinforcement Learning (RL) methods may become an effective tool for solving this online problem, as they are able to learn and adapt to the changing workload running on the system and successfully suppress power budget overshoot while maximizing performance.

Although RL-based DVFS methods are effective, they may require several thousands of iterations or more to train the model before being deployed. Data collection in real world can be expensive due to physical limitations, *e.g.*, power meters are used for measuring device power, but are limited by measuring speed and quantity. For instance, measuring ten thousand iterations for a five-minute application takes one month. Hence, reducing training iterations is critical for learning the model in reasonable amount of time to make RL-DVFS practical and reduce time to market.

The dual problem of applying machine learning for improving system efficiency is enhancing the efficiency of machine learning algorithms themselves. Deep Convolutional Neural Networks (CNNs) have become increasingly important due to their unprecedented effectiveness on tasks such as image classification, object detection, image segmentation, *etc*. However, CNNs usually require millions of data samples during training and high-end GPUs during inference due to the high computation and storage cost. As a result, CNNs are often considered very computationally intensive and energy demanding [5, 6, 7]. With the prevalence of mobile devices, being able to run CNN-based visual tasks efficiently, in terms of both speed and energy, becomes a critical

enabling factor of various important applications, *e.g.*, augmented reality, self-driving cars, Internet-of-Things, *etc*, which all heavily rely on fast and low energy CNN computation.

## 1.2 Thesis contributions

In light of the aforementioned challenges, this thesis investigates two important aspects of machine learning algorithms.

The first aspect is on applying machine learning, especially Reinforcement Learning (RL), in many-core system optimization. By using RL, we are able to learn and adapt to the changing workload in the system and hence successfully suppress power budget overshooting while maximizing system performance with very low runtime overhead. Furthermore, to improve the training efficiency of RL-based Dynamic Voltage Frequency Scaling (DVFS) algorithms, we propose to use Bayesian Optimization (BO) which is very sample-efficient and achieve significant speedup over conventional approaches.

The second aspect is on optimizing machine learning algorithms, specifically deep Convolutional Neural Networks (CNNs), for efficient training and inference. For any CNN approach, there are usually two phases involved: training and inference. CNN often requires millions of data samples during training in order to achieve high accuracy and avoid overfitting, and is very computing intensive and energy demanding during the testing phase. In this thesis, we first propose the virtual pooling method to siginificantly reduce the CNN inference energy and latency, and then introduce the method of training with fine-grain labels which drastically reduces the number of training data samples required to achieve high testing accuracy.

A detailed list of contributions of each work is included in the following chapters.

## 1.3 Thesis organization

The rest of this thesis is organized as follows. Chapter 2 presents the background knowledge required for this thesis. In Chapter 3, we introduce the reinforcement learning-based approach for performance optimization for power and performance constrained many-core systems. Chapter 4 presents our approach for speeding up training of RL-based DVFS control algorithms through Bayesian optimization. In Chapter 5 and 6, we detail our methods of efficient CNN inference and training, respectively. Chapter 7 describes and discusses related works, and we conclude this thesis in Chapter 8.

# Chapter 2

# Background

## 2.1 Reinforcement learning

Reinforcement Learning (RL) [8] is inspired by the trial-and-error method humans used for making decisions for millions of years. In RL, the agent interacts with the system (Fig. 2.1), *e.g.*, committing actions based on the state of the system and also observing the new state of the system. The goal of RL is to find the best actions under different states such that by following those best actions, the agent can optimize the long-term reward. The probability of selecting an action $a$ under a state $s$ is called a *policy* $\pi(s, a)$. RL determines how the agent should change its policy by experiencing the states. As

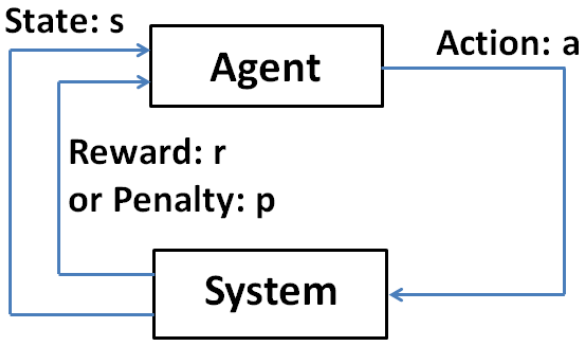



Figure 2.1: Agent-system interaction in RL. The systems is characterized by state $s$. The action $a$ is taken by an agent to change the state, with resulting outcomes $r$ or $p$ and new state $s'$.

the problem is sequential, the long-term reward starting from time $t$ is defined as $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, where $r_i$ is the *immediate reward* at time $i$. The *discount factor* $\gamma$ determines how important the future reward is. If $\gamma = 1$, future reward is as important as the immediate reward $r_t$. If $\gamma = 0$, the agent is oblivious to future rewards. $Q^\pi(s,a)$ is the expected value of $R_t$ when starting from state $s$, taking action $a$ and then following the policy $\pi$: $Q^\pi(s,a) = \mathbb{E}\{R_t | s_t = s, a_t = a\}$. The optimal $Q$ value is defined as $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ which is the maximum long-term reward. More formally,

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\{r + \gamma \cdot \max_{a'} Q^*(s',a') | s_t = s, a_t = a\} \\ &= \sum_{s'} P^a_{ss'} \cdot [R^a_{ss'} + \gamma \cdot \max_{a'} Q^*(s',a')] \end{aligned} \tag{2.1}$$

where $P^a_{ss'}$ is the transition probablity (probability of transitioning to state $s'$ by taking action $a$ at state $s$), and $R^a_{ss'}$ is the expected reward of reaching state $s'$ by taking action $a$ at state $s$. This equation is known as Bellman optimality equation for $Q^*$ [8] which finds the action $a'$ that can maximize the $Q^*$ value of the next state while averaging over all the possible next states. However, one usually does not know the value of $P^a_{ss'}$ and $R^a_{ss'}$.

Addressing this, Q-learning [9] has become one of the most important breakthroughs in RL [8], because (1) it converges to the Bellman optimal solution in an on-line and incremental manner; and (2) it does not require the model of system, *e.g.*, prior knowledge of $P^a_{ss'}$ and $R^a_{ss'}$. The following equation gives the updating rule of Q-learning:

$$\begin{aligned} Q(s,a) &= Q(s,a) \\ &\quad + \theta \cdot \{r + \gamma \cdot \max_{a'} [Q(s',a')] - Q(s,a)\} \end{aligned} \tag{2.2}$$

The agent starts from state $s$ and chooses action $a$. By observing the reward $r$ and the next state $s'$, the sum $r + \gamma \cdot \max_{a'} [Q(s',a')]$ provides the expected long-term reward of the state-action pair $(s,a)$. Q-learning updates the Q value incrementally (suitable when the agent experiences the state-action pairs sequentially) by using the difference $\{r + \gamma \cdot \max_{a'} [Q(s',a')] - Q(s,a)\}$. Repeating this procedure, Q-learning is guaranteed to converge to the Bellman optimal solution [9], *i.e.*, solution to equation 2.1.

## 2.2 Bayesian optimization

Bayesian Optimization (BO) is a well-known sample-efficient optimization method for black-box functions, and has been extensively studied [10, 11, 12, 13, 14, 15, 16]. Figure 2.2 [1] shows an illustration of how BO works, in which the blue solid curve is the true (and unknown) target function from which we want to find the maximum value. Figure 2.2 (a) illustrates how BO finds the maximum value through exploring various input values $x$ and Figure 2.2 (b) shows the utility function BO uses to determine which input values to choose.

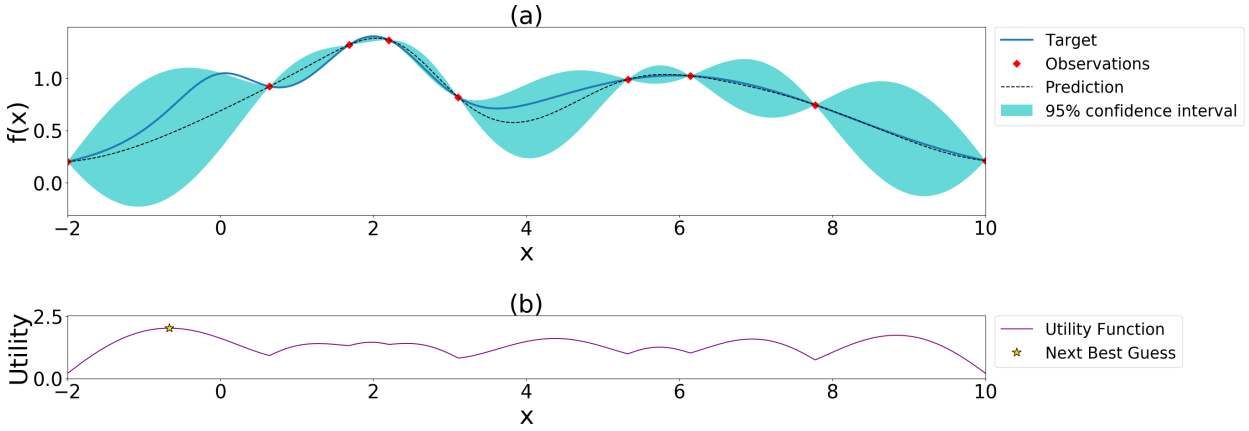



Figure 2.2: Illustration of Bayesian optimization [1] .

BO first constructs a prior distribution of functions, usually a Gaussian process $f(x) \sim \mathcal{GP}(m(x), k(x, x'))$ with mean and co-variance as functions of $x$ (and $x'$). This means that at each input point $x$, our belief of the output function $f(x)$ follows a Gaussian distribution, and its mean and variance (95% confidence interval) are illustrated as the dotted curve and the cyan-colored area in Figure 2.2 (a). Then BO determines the next point to observe based on an acquisition function (shown in Figure 2.2 (b)), the intuitive purpose of which is to "guess" a good candidate to probe by balancing the prediction (mean of Gaussian) and uncertianty (variance of Gaussian). For example, Upper Confidence Bound (UCB) uses a combination of mean and variance: $\mathcal{UCB}(x) = \mu(x) + \kappa\sigma(x)$. When $\kappa = 2$, it is the upper bound of the cyan-colored area. With this new point probed,

we observe the real output from the experiments without uncertainty, shown as the red points in Figure 2.2 and the cyan-colored area collapses to none at those points. Then we update the posterior distribution, *i.e.*, mean and co-variance functions of the Gaussian process, following Bayes' Theorem.

The procedure described above is repeated. As the number of observations grows, the posterior distribution improves, and the algorithm becomes more certain of which regions in the input space are more likely to be the global maximum and hence worth exploring.

## 2.3 Deep convolutional neural networks

Deep Convolutional Neural Networks (CNNs), as illustrated in Figure 2.3, have seen great success in computer vision tasks, *e.g.*, image classification, object detection and image segmentation.

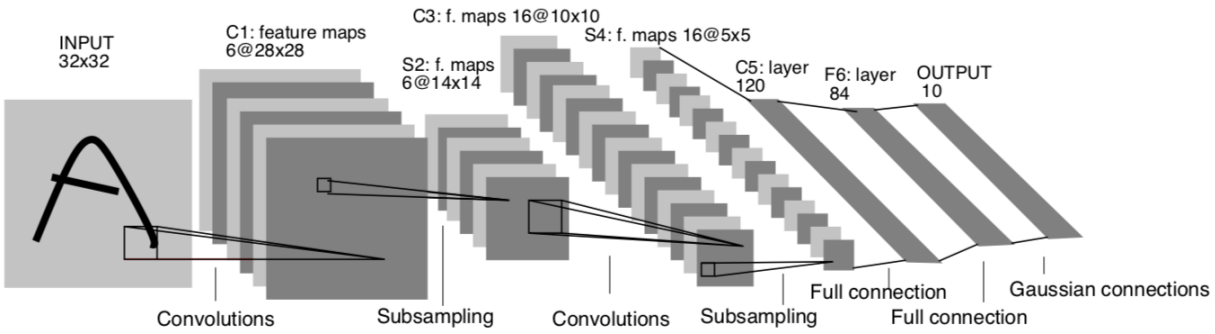



Figure 2.3: Illustration of convolutional neural networks.

A CNN usually consists of multiple convolution layers, in which 2D convolution is performed over the input, and (max-)pooling layers, in which input is downsampled through max operation over a small sliding window. Each convolution layer is followed by a non-linear ReLu (Rectified Linear unit: $\text{ReLu}(x) = \max(0, x)$) layer to avoid collapsing into a linear system. In the application of image classification, the output of above layers are usually converted into a one-dimensional vector $\mathbf{O} = [O_1, O_2, ..., O_n]$.

**O** then goes through a Softmax layer and is transformed into a probability distribution as $p_i = \frac{exp(O_i)}{\sum_{i=1}^{n} exp(O_i)}$. The element with the highest probability is chosen as the final prediction of the image class.

The parameters $w$ in CNN models, *e.g.*, filters in the convolution layers, are determined through a training process and they decide how CNN may eventually perform. In order to learn these parameters, in supervised learning, a loss function, which shall be minimized through tuning $w$, is defined as the distance between the prediction and the ground-truth. Cross entropy loss $\mathcal{L} = -log(p_c)$, where $c$ is the true class, is often used as the loss function for classification problems. Loss minimization, *i.e.*, training or learning, is typically done by using Stochastic Gradient Descent (SGD). SGD randomly selects a batch of training data samples, calculates the gradient of loss of this batch with respect to each trainable parameter, *i.e.*, $\nabla_w \mathcal{L} = \frac{\partial Avg_{batch}(\mathcal{L})}{\partial w}$, and then updates each of the parameters through gradient descent: $w \leftarrow w - \alpha \cdot \nabla_w \mathcal{L}$, where $\alpha$ is learning step size.

# Chapter 3

# Learning for efficient many-core systems

## 3.1 Chapter overview

Historically the major goal of processor designers was to achieve better performance by continuously shrinking device size, adding more pipeline stages, and speeding up the clock speed. However, all these performance increasing mechanisms have pushed most designs over the power wall and energy is now the *de facto* main design constraint.

The ever growing power consumption directly impacts heat dissipation, lowers chip reliability, and decreases battery life of mobile devices. As Dennard scaling breaks down, multi-core systems have become the solution to mitigate the issue of increased power density. For highly parallel applications, multiple small cores with lower voltage and frequency can provide a similar throughput as a single large core at a much higher voltage and frequency, while they consume less energy according to the $V^2f$ Scale Law [3]. While multi-core systems are now used in mainstream consumer products, the desire for higher performance is again pushing the envelope toward higher power consumption. In order to ensure the safety and reliability of multi-core systems, a Thermal Design Power (TDP) constraint for the system power consumption is imposed. It is known that high performance usually leads to high power consumption and lower performance consumes less power, therefore there is a clear trade-off between system power con-

sumption and performance. While reducing power to avoid TDP overshoot, we still need to ensure that a certain Quality-of-Service (QoS) level is maintained. In such a case, per performance constraints must be satisfied. As a result, maximizing performance while satisfying required minimum per-application performance and maximum system power constraint becomes one of the main directions in power/performance optimization. In addition to increasing the number of cores, Dynamic Voltage Frequency Scaling (DVFS) has been extensively used to adapt performance and power based on application requirements. By being smart in tuning the Voltage and Frequency (VF) levels of the cores, on-chip computation can be performed in a more energy efficient manner. However, prior work mostly focuses on DVFS control under either power or performance constraints. Finding the optimal VF level assignment can be formulated as an integer linear programming problem which is NP hard [4]. Therefore, an exact solution cannot be found using an on-line algorithm, especially when the number of cores increases. Several algorithms have been proposed to find near-optimal solutions in polynomial time, however, they either did not take the budget overshoot problem into consideration or were not considering performance constraints for all applications. They also may only be efficient for small-scale multi-core systems rather than systems with hundreds of cores. By exploiting both spatial and temporal hierarchy, we propose Online Distributed Reinforcement Learning (OD-RL) and its priority-aware counterpart, pa-OD-RL [17] that are able to improve performance under power constraints and performance requirements with significantly smaller TDP overshoot and runtime overhead.

### 3.1.1 Chapter contributions

To the best of our knowledge, the work presented in this chapter makes the following contributions:

- The proposed method for multi-core power budget overshoot control **overcomes the scalability issues of RL** while maintaining its powerful aspect of learning the

optimal action without building a full model of the system. The RL method is known to not scale, since the number of states increases exponentially with the number of cores. In this work, we exploit the *spatial and temporal hierarchical* structure, and propose the OD-RL method that combines RL and an efficient power budget reallocation algorithm. The number of states of OD-RL therefore no longer depends on the number of cores, and the algorithm complexity is reduced dramatically to $O(N \cdot log(N))$ which is also independent of the number of VF levels.

- Indeed, power constrained performance improvement is usually solved by pushing the average power close to the TDP constraint, regardless of the resulting TDP overshoot. Many methods do exceed the budget quite often especially because they rely on the $V^2f$ Scale Law or modeling methods that introduce inaccuracies leading to incorrect decisions. Our approach is able to adapt to the workload, mitigate the inaccuracy problem and minimize the likelihood of overshooting TDP.

- Our approach is **efficient and accurate** when compared to state-of-the-art approaches. We evaluate our proposed method, MaxBIPS [3], and Steepest Drop [18] on a wide spectrum of parallel, multi-threaded applications in systems with as many as 64 cores. The experimental results show up to **98%** less budget overshoot, **23%** energy efficiency improvement and **100x** speedup in a 512-core system when compared to Steepest Drop [18].

- We further propose*priority-aware* OD-RL (pa-OD-RL) for the case of applications with stringent performance requirements. Compared with OD-RL, pa-OD-RL can better satisfy performance constraints and provide an advantageous trade-off (**20.0x** better) when trying to satisfy power and performance requirements.

## 3.2 Methodology

RL algorithms have traditionally been used to find the optimal solution for sequential decision problem, and have been proved effective in a variety of problems from different areas [8]. However, RL often suffers from state space explosion and thus is too expensive for large scale problems. Our proposed OD-RL method [19] assigns V/F levels to each core to improve the global performance under a given power budget in a scalable manner. To address the scalability issue of RL, our approach uses a distributed RL agent working at a finer grain on each cluster of cores and a course grain efficient power budget reallocation algorithm.

When applied to power or performance optimization for large scale computing systems, one must define the machine states and actions globally, and therefore the state space grows exponentially. Therefore, if a centralized RL-based approach is used, the overall number of states and actions grows exponentially with the number of cores. To illustrate this exponential growth, let us assume that the system has $N$ cores, $K$ features per core (*e.g.*, instructions committed per cycle or IPC, misses per kilo-instructions or MPKI, *etc.*) and at most $D$ different values per feature (*e.g.*, high/low IPC or MPKI). The $k^{th}$ feature of core $i$ is denoted as $f_{ik}$ and the global state is therefore defined as $S_{global} = \{f_{11}, f_{12}, ..f_{1K}...f_{N1}, ...f_{NK}\}$. Thus, the number of global states of a $N$-core system becomes $|S_{global}| \propto D^{KN}$. The action for the $N$-core system will be a vector $A_{global} = (a_1, a_2, ...a_N)$ where $a_i$ is the action of the $i^{th}$ core. Without loss of generality, if we assume at most $\alpha$ possible actions for each core, then the number of possible actions is $|A_{global}| \propto \alpha^N$. As a consequence, in a centralized RL-based method, the total number of state-action pairs would be $|S_{global}| \cdot |A_{global}| \propto D^{KN} \cdot \alpha^N$ which is indeed exponential in the number of cores $N$. One way to mitigate this exponential growth is to distribute the actions to different cores, in which case each core makes its own independent decision. While the number of states is the same as before, the cardinality of the action

state space is reduced to $\alpha N$. Therefore, for each core, the number of state-action pairs would be $|S_{global}| \cdot |a_i| \propto D^{KN} \cdot \alpha$, which is much smaller than the previous case, but still exponential in the number of cores. The required training time and memory overhead will render both this and the centralized RL approach impractical in real applications.

We note, however, that a globally defined state space contains more information than necessary. One way to reduce the complexity is to retain only the features of the current core. By doing so, the state kept by each core $i$ is $S_i = \{f_{i1}, f_{i2}, ..f_{iK}\}$. The number of state-action pairs per core is further reduced to $|S_i| \cdot |a_i| \propto D^K \cdot \alpha$ or $N \cdot D^K \cdot \alpha$ for the entire system which is no longer exponential in the number of cores. Consequently, a distributed RL algorithm allows every core to run independently and learn the optimal DVFS control policy on its own. Since each RL agent can run independently, the time complexity of this method is only proportional to the number of actions $O(\alpha)$. On the downside, we achieve this high scalability at the cost of missing information on the other core states. To address this problem, we propose to allow local agents to run at a finer granularity and reallocate the power budget at a coarser granularity. In the latter reallocation step, all cores receive new power budget constraints from a global power budget reallocator.

Therefore, the problem of RL-based power constrained performance improvement is decomposed into two sub-problems.

1. At finer time granularity, when given a fraction of total power budget, each core learns the best policy that maximizes its performance under that budget, locally.

2. At coarser time granularity, a global power budget reallocator redistributes power across cores to better satisfy the performance constraints and ensure a better utilization of the total budget, globally.

Fig. 3.1 depicts the overall structure of our approach. Distributed RL agents work on each core locally and independently, while the global budget reallocator redistributes the budget among all the cores. The temporal granularity is also different: distributed RL agents operate at every control epoch, while the budget reallocator executes every

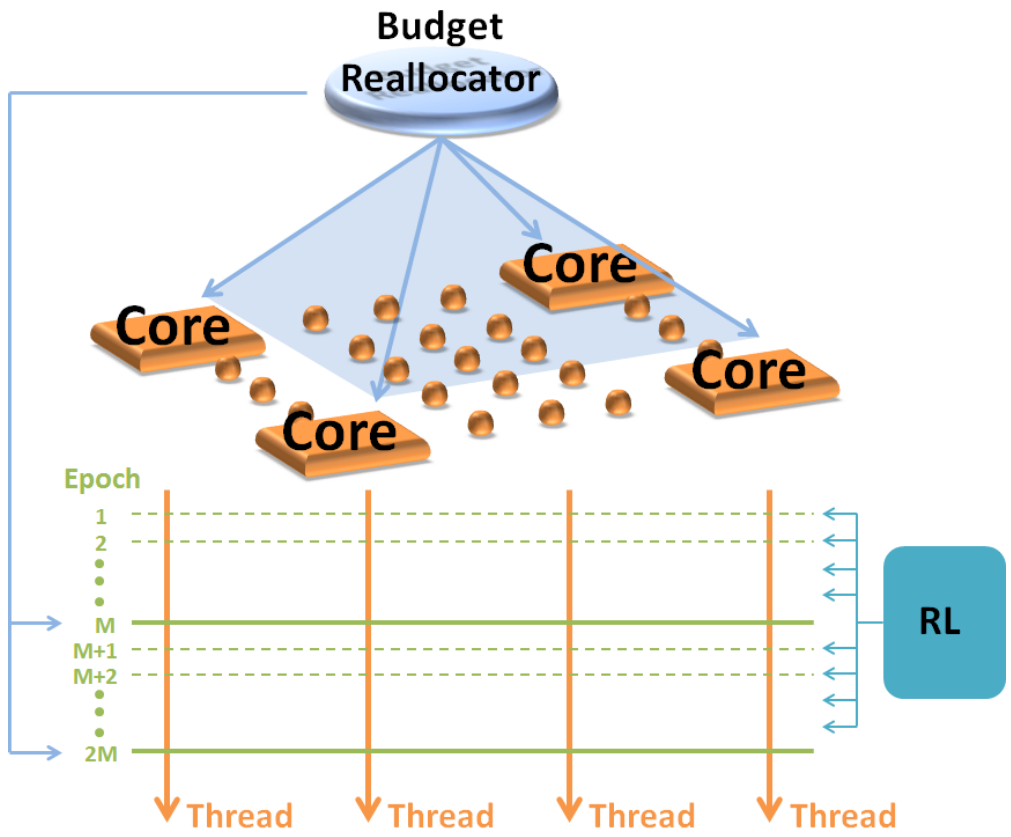



Figure 3.1: Structure of the algorithm. Spatial hierarchy: budget reallocation does global coordination while RL is distributed to each core. Temporal hierarchy: budget reallocation works at coarser gain while RL executes at finer grain.

$M$ epochs. The best $M$ value is typically found through experimentation. For the multi-threaded parallel benchmarks and the platforms we have considered herein, a suitable value was $M = 15$.

### 3.2.1 Fine temporal granularity: reinforcement learning local agents

Each core will run Q-learning once per control epoch so as to learn the best policy that maximizes its performance under a given power budget. We define the state of each core as $S$={Instruction Per Cycle (*IPC*), Million L2-cache-misses Per Kilo-Instructions (*MPKI*), current power value (*Power*), current VF level (*VF level*)}. Traditionally, the features are discretized uniformly across their value range. Instead, we discretize *IPC* by its sta-

tistical distribution to minimize inter-interval oscillation due to a poor discretization threshold. We define the reward $r$ as the core throughput, which quantifies the preference for an action that leads to higher performance. The budget-overshoot penalty is $-PF \cdot |power\ value - power\ budget|$ where $PF$ is the penalty factor. The penalty is proportional to the power budget overshoot, following the intuition that it is more desirable to eliminate a high power budget overshoot. We discuss the choice of the $PF$ value in section IV.B.

To accelerate the convergence of Q-learning, we propose a *batch-update method*. For any state $s_0$ and $s_1$, we define $s_1 > s_0$, when all features part of $s_1$ have larger values than those of $s_0$. By analyzing the program trace, we observe that, if a core exceeds the power budget for state-action pair $(s, a)$, it will also exceed for any $(S, a), \forall S > s$. Therefore, we penalize all state-action pairs $(S, a), \forall S > s$, when the budget is exceeded for $(s, a)$. This way, unnecessary budget overshoots are eliminated by taking advantage of the ordering between the states.

The proposed RL algorithm is able to suppress the budget overshoot by learning the transition probability of the workload. However, these actions can sometimes be conservative due to volatility of dynamic workload behavior. We observe that for some workloads a high frequency state selection is sometimes penalized due to bursty spike in power. However, overshooting the power budget for a short time (typically less than a few milliseconds) will not produce any thermal emergencies. To account for this, we develop a memory mechanism to quantify the presence and frequency of these short power bursts in a similar fashion to branch predictors [20]. In more detail, we maintain an $m$-bit memory queue for each state-action pair. If a state-action pair choice results in the core being under the power budget, the memory receives a 1, otherwise it gets a 0. When a state-action pair receives a 0, which means a budget overshoot, it will check its $m$-bit memory: (1) it will not penalize the pair ($Q$ value not changed) when all the bits in current memory are 1s. (2) it will penalize the pair (change the $Q$ value based on Q-learning algorithm) when at least one bit in the $m$-bit memory is 0. We experimentally

determine the best value for $m$ as 3. This proposed memory mechanism can accommodate short power budget violations and mitigate performance degradation due to bursty spikes while still being able to learn the periodic benchmark behaviors, *e.g.*, loops.

### 3.2.2 Coarse temporal granularity: power budget reallocation

As distributed RL maximizes the performance under a given budget per core, power budget reallocation algorithm serves as a global coordinator to further maximize the performance and power utilization at a coarser granularity. Following the same intuition as MaxBIPS [3] which favors CPU-intensive threads, we propose the Maximize-the-Max (MM) method which always maximizes the VF level of the busiest cores. Algorithm 1 gives the pseudocode of MM method. Let's associate a tuple (core number, IPC) = ($i$, $IPC_i$) to each core $i$. We can define a "core number-IPC" profile: $\mathbb{CORE} - \mathbb{IPC} = \{(0, IPC_0), (1, IPC_1), \ldots, (n-1, IPC_{n-1})\}$. Power of core $i$ at VF level $VF_i$ is denoted as $Pow(i, VF_i)$. At every power reallocation step, MM first estimates the power consumption of all the cores at their lowest VF level, and subtracts this value from the total budget. Then MM builds a max-heap out of $\mathbb{CORE} - \mathbb{IPC}$ sorted by the IPC value, and pops out the core with the highest utilization. The power budget required to boost this core to the highest feasible (*i.e.*, still within the remaining power budget) VF level is calculated and subtracted from the total power budget while the highest VF level is assigned to the core. If there is still residual budget after the first allocation, MM will pick the new top of the heap and will again allocate the budget to allow the highest possible VF level selection. MM repeats this process until no budget value is left. In the case of unbalanced workload, *e.g.*, when there is a critical thread running and the others are waiting for it, the other threads may stay in lower performance state, *e.g.*, stalling or spinning. The global budget re-allocator will then assign most of the power budget to the core running the most critical thread and the RL agent will hence boost its performance under the higher power cap, though our algorithm does not explicitly

balance the load. To implement this algorithm, we used the max-heap to sort the cores in the order of their utilization, with a time complexity of $N \cdot log(N)$. To estimate the power consumption of core $i$ at a different VF level $VF_i = q$, we use the $V^2f$ Scale Law [3] which states that dynamic power is proportional to $V^2f$ where $V$ is voltage and $f$ is frequency. Assuming core $i$ is originally at $VF_i = p$, we estimate

$$Pow(i,q) = Pow(i,p) \cdot \frac{Volt^2(q) \cdot freq(q)}{Volt^2(p) \cdot freq(p)} \cdot \kappa(i) \tag{3.1}$$

where $\kappa(i)$ is the discount factor of core $i$ accounting for the transition cost of VF level [3], and $Volt(p)$ and $freq(p)$ are the voltage and frequency of VF level $p$, respectively. We determine $\kappa(i) = 0.9 \; for \; \forall i$ experimentally. Since subthreshold leakage power usually remains stable for a specific voltage, to further increase the power estimation accuracy, we subtract it from the total power first, then do the $V^2f$ scale on dynamic power, and finally add the leakage power of the new voltage back to the total power. The subthreshold leakage power of different voltages are obtained during machine idle period.

### 3.2.3 Priority based performance requirements

We further consider required performance constraints that the applications should satisfy. We define the performance requirement for each application as a percentage out of the maximum possible throughput achieved when the application runs at the highest VF level. For example, if the maximum possible throughput of an application at time $t$ is $T_{max}(t)$, then to satisfy a performance requirement of 95% , the application should maintain a throughput of $95\% \cdot T_{max}(t)$. In a system with multiple applications, we may have several performance requirements $\chi_1 > \chi_2 > \chi_3 > ... > \chi_n$. We say that an application has *h*igher priority if its performance requirement $\chi$ is larger. To satisfy these performance constraints as well as better utilize the power budget, we propose a priority-aware MM (pa-MM) variant. The intuition behind pa-MM is as follows:

1. Application performance is boosted in descending order of priorities. In order words, the applications with highest priority is sped up first.

2. Within each application, threads are boosted based on core utilization by greedily assigning high VF levels.

Similar to MM, pa-MM first estimates the power consumption of all cores at the lowest VF level and subtracts the sum from the total power budget. Then, for each application, pa-MM builds a max-heap of elements in $\mathbb{CORE} - \mathbb{IPC}$ based on the IPC values. The core part of the algorithm consists of two loops. The outer loop (Line: 10) iterates over the applications that are part of the workload, from the highest priority to the lowest priority. The inner loop (Line: 11), assigns the highest VF level to the threads of each application in descending order of core utilization until the total application throughput satisfies its performance constraint. The aforementioned max-heap is used to pop out the core with the highest utilization among the remaining cores. If there is still power budget left (Line: 9) after iterating over these two loops, the algorithm will go through the two loops again to boost up threads that are not considered in the first round. This process terminates when there is no power budget left (Line: 19).

### 3.2.4 Power-performance model

As shown above, we use the $V^2f$ Scale Law [3] to estimate the power of a core $i$ at a different VF level $VF_i = q$ as a function of the core power in its original $VF_i = p$ level. To this end, to compute the current power value per core, we develop a multivariable polynomial regression model. As shown in [21], power consumption can be written as:

$$\mathcal{P} = P_{dyn} \cdot u + P_{sta} \tag{3.2}$$

where $P_{dyn}$ and $P_{sta}$ correspond to the peak dynamic and static power consumption from the design specifications, and $u$ encapsulates the processor utilization. As prior art has shown [21], this activity rate can be effectively approximated as a linear function of the total number of instruction per cycle (IPC). Hence, for the current VF level $VF_i = p$, we can write [22]:

$$\mathcal{P}_p = P_{dyn_p} \cdot (\mu_p \cdot \mathcal{IPC} + \nu_p) + P_{sta_p} \tag{3.3}$$

**Algorithm 1** Pseudocode of priority-aware MM method

Input: $\mathbb{CORE} - \mathbb{IPC}$, *Global Budget*, $N$, $N_{app}$, *Priority*
Output: $Budget_i$ for a core $i, 1 \leq i \leq N$
Variable: *Residual Budget* ($Res\_B$).
**for** $i \leftarrow 1; i \leq N; i \leftarrow i+1$ **do**
    $Budget_i \leftarrow Pow(i, VF_i = lowest)$
**end for**
$Res\_B \leftarrow Global\ Budget - \sum_{i=1}^{N} Pow(i, VF_i = lowest)$
For each application, build a max-heap of $\mathbb{CORE} - \mathbb{IPC}$ based on the IPC value
**while** $Res\_B > 0$ **do**
    **for** $App \leftarrow 1; App \leq N_{app}; App \leftarrow App+1$ **do**
        **while** $Thoughput\ of\ App < Priority_{App} * MaxThroughput$ **do**
            Pop the max-heap of application $App$ and get the tuple $(i, IPC_i)$
            $\Delta \leftarrow Pow(i, VF_i = highest) - Pow(i, VF_i = lowest)$
            **if** $\Delta \leq Res\_B$ **then**
                $Budget_i \leftarrow Budget_i + \Delta$
                $Res\_B \leftarrow Res\_B - \Delta$
            **else**
                $Budget_i \leftarrow Budget_i + Res\_B$
                return
            **end if**
        **end while**
    **end for**
**end while**

That is, given the design specifications for the peak dynamic and static power consumption per VF level, and the $\mathcal{IPC}$ values from the performance counters of the system, we can learn the $\mu$'s and $\nu$'s parameters across all the different VF levels from empirical data. Note that our approach is general and other (nonlinear or polynomial) approximations can be used to link utilization with performance counters like IPC. Moreover, the model is flexibly extendable to including process variations [21, 23] and aging phenomena [24, 22], whose integration we leave for future work.

To assess the accuracy of our power-performance model, we simulate a multi-core system with four cores for all the VF levels considered later in our experimental setup (*i.e.,* Table 3.2). At runtime, we collect the power traces, both dynamic and static power, from `McPAT` [25], and IPC traces from `Sniper` [26]. Based on the collected power data per application and per VF level, we train our model and we learn the $\alpha$ and $\beta$ parameters

(Equation 3.3). Table 3.1 shows the normalized root-mean-square error (NRMSE) of the fitted models per benchmark, averaged across all the considered VF levels. Indeed, the error margins are consistent with the accuracy reported in similar learning-based methods [21].

Table 3.1: Validation of learned $\alpha$'s and $\beta$'s: NRMSE error.

| Benchmark | NRMSE | Benchmark | NRMSE |
|---|---|---|---|
| fft | 8.29% | lu-ncont | 11.95% |
| canneal | 3.75% | cholesky | 11.48% |
| radiosity | 5.00% | fluidanimate | 4.17% |
| ocean-ncont | 10.13% | radix | 1.83% |
| swaptions | 3.65% | vips | 18.93% |
| barnes | 3.71% | ocean-cont | 9.81% |
| lu-cont | 12.81% | blackscholes | 6.83% |
| | | mean | 8.02 % |

## 3.3 Experimental results

### 3.3.1 Experiment setup

We verify the effectiveness of the proposed method by using `Sniper` v5.3 simulator [26] with its natively supported `PARSEC` and `SPLASH-2` multi-threaded benchmark suites. The simulation infrastructure is shown in Fig. 3.2.

`Sniper` is able to perform timing simulations for multi-threaded, shared-memory applications with tens to 100+ cores, and it has been validated against Intel Core2 and Nehalem systems. `Sniper` normally uses `McPAT` [25] for system power estimation. However, since we use 16nm technology and `McPAT` supports down to 22nm, we use the proposed power-performance model for power estimation. The system configuration is shown in Table 3.2. The architecture we considered has one DRAM controller for every four processing cores. In `Sniper` simulator, both RL agent and MM/pa-MM are implemented as Python scripts. In every control epoch, this script will be invoked by `Sniper` and executed to select the VF level of all the cores. We vary the epoch size from 500$\mu s$ to

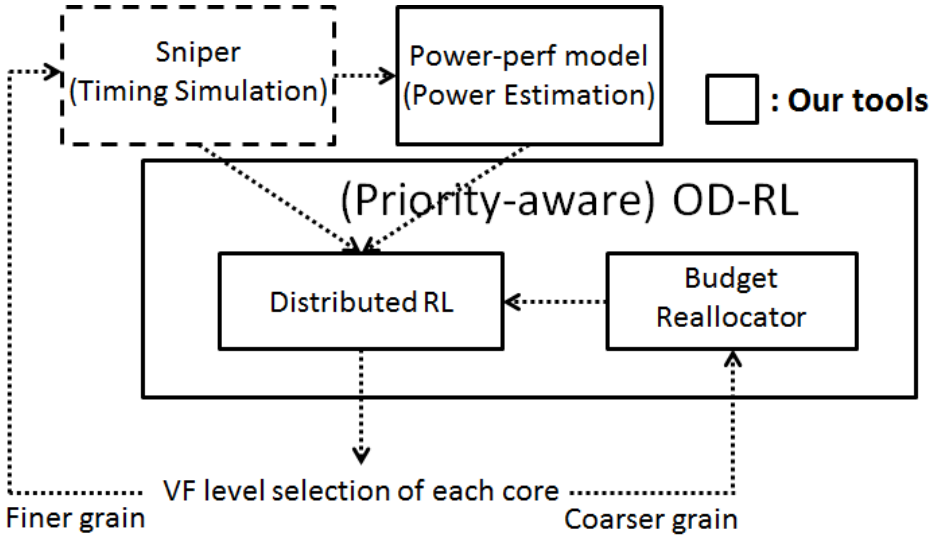



Figure 3.2: Simulation infrastructure: `Sniper` for timing simulation and our proposed power-performance model for power estimation. Distributed RL and budget reallocator are implemented as Python scripts to make the VF level decisions and interact with `Sniper`.

Table 3.2: Architectural parameters

| | |
|---|---|
| Number of cores | 16 |
| Architecture | Intel Gainestown |
| L1-I/D cache | 32KB, 4-way, LRU |
| L2 cache | 512KB, 8-way, LRU |
| L3 cache | 8MB, 16-way, LRU |
| VF levels (GHz/V) | 2.2/0.65, 2.4/0.75, 2.6/0.85, 2.8/0.95 |
| Nominal VF level | 2.6/0.85 |
| Control epoch | 500$\mu s$ to 10$ms$ |
| Budget reallocation period | every 15 epochs |
| Technology node | 16 nm |

10$ms$ across different benchmarks, because (i) we would like to explore the effectiveness of our approach across different epoch lengths; and (ii) we are using the large input set for each benchmark and therefore we require hundreds of epochs to demonstrate the effectiveness of the algorithm in a statistically significant manner.

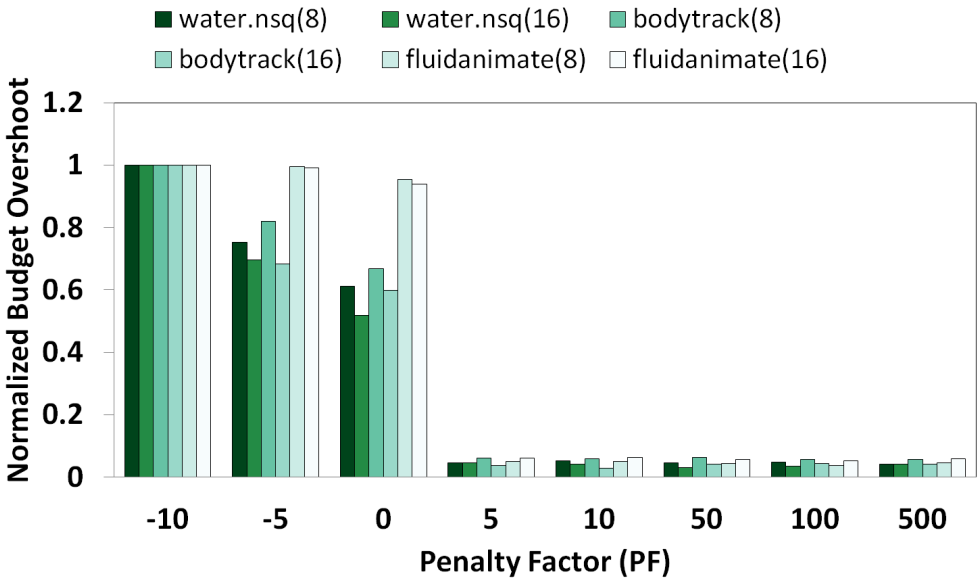



Figure 3.3: Budget overshoot quickly satures when Penalty Factor (PF) is larger than 5.

### 3.3.2 Budget overshoot control

The penalty term in each RL agent plays an important role in determining the algorithm behavior [8]. Higher penalty offers better overshoot suppression, but can also negatively impact the performance. Lower penalty is able to give better performance at the cost of larger budget overshoot. In order to find the best Penalty Factor ($PF$) value, we analyzed the impact of penalty value on budget overshoot for different benchmarks and core counts. The results in Fig. 3.3 show that the budget overshoot quickly saturates when $PF \geq 5$, therefore we select $PF = 5$ as it offers the best trade-off between penalty and performance.

We first evaluate our OD-RL together with four state-of-the-art algorithms: MaxBIPS, PullHighPushLow, Priority from [3], and Steepest Drop from [18] in terms of their budget overshoot control, relative performance improvement, and runtime overhead. MaxBIPS does combinatorial optimization according to core utilizations, while PullHighPushLow and Priority boost core processing speed greedily based on core utilization and priority, respectively. Steepest Drop uses the per-core energy efficiency as the heuristic to quickly optimize for system energy and performance with lower runtime

overhead. For MaxBIPS, we use Equation 3 for budget control as mentioned in the original paper [3]. Budget overshoot is defined as the total energy consumption over the TDP. This extra energy increases the risk of thermal emergencies and subsequent throttling, negatively impacts the chip reliability, and puts heavier burden on the cooling system. Fig. 3.4 shows the budget overshoot control of five algorithms (including OD-RL). Our algorithm is able to better suppress the budget overshoot in most cases, by a margin of several orders of magnitude. In *swaptions*, we see that our approach does not perform as well as Steepest Drop and is close to MaxBIPS. The reason for this behavior is that *swaptions* is a very balanced multi-threaded application in which all cores have the same utilization across chip and in time. The exception is core 0 which does nothing ($IPC_0 = 0$). In this case, the inevitable on-line learning overhead of OD-RL increases the relative budget overshoot. For *canneal* and *blackscholes*, OD-RL is slightly worse than Priority in budget overshoot control, while it has much better performance (almost double in blackscholes in Figure 3.5) under similar or less power consumption. In *Radiosity*, OD-RL saves **98%** budget overshoot due to its capability of adapting to the workload change and hence reducing the over-the-budget energy to a much lower value 0.06J compared to the other approaches. Our method achieves the lowest budget overshoot with an average budget overshoot reduction of **73.4**% comparing to the baseline MaxBIPS. OD-RL outperforms the state-of-the-art approaches because it is able to learn the workload transition probability and therefore suppress the budget overshoot.

In Figure 3.5 and 3.6, we show performance and power results of the five algorithms mentioned before. We point out that the lower budget overshoot our method provides may lead to lower average power, as well as lower performance. As we can see in Figures 3.5 and 3.6, the average performance and power are lower than prior work. However, compared to MaxBIPS, given a **7.2%** power reduction and **73.4%** budget overshoot reduction, we only sacrifice performance by **2.8%** considering that MaxBIPS is near-optimal in terms of performance. Therefore, we not only achieve better energy efficiency, but

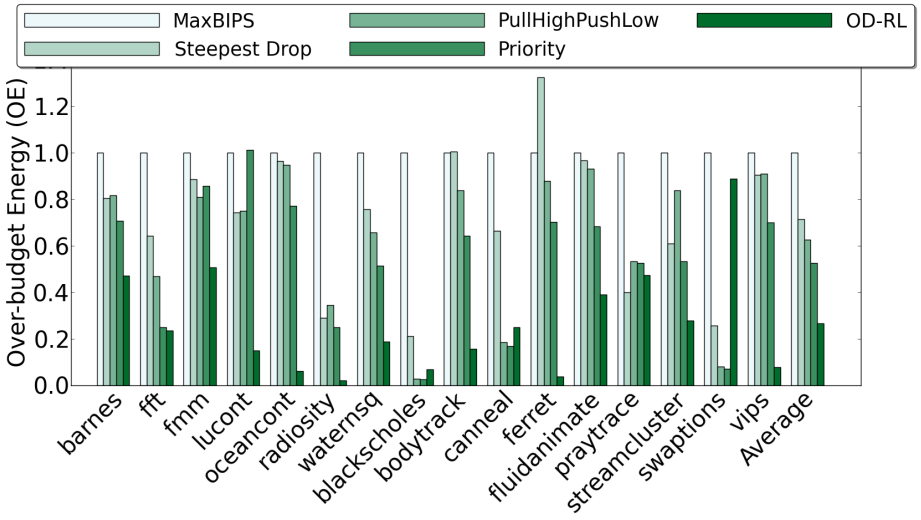



Figure 3.4: Comparison among five algorithms on budget overshoot control (Lower is better). Results normalized with respect to MaxBIPS.

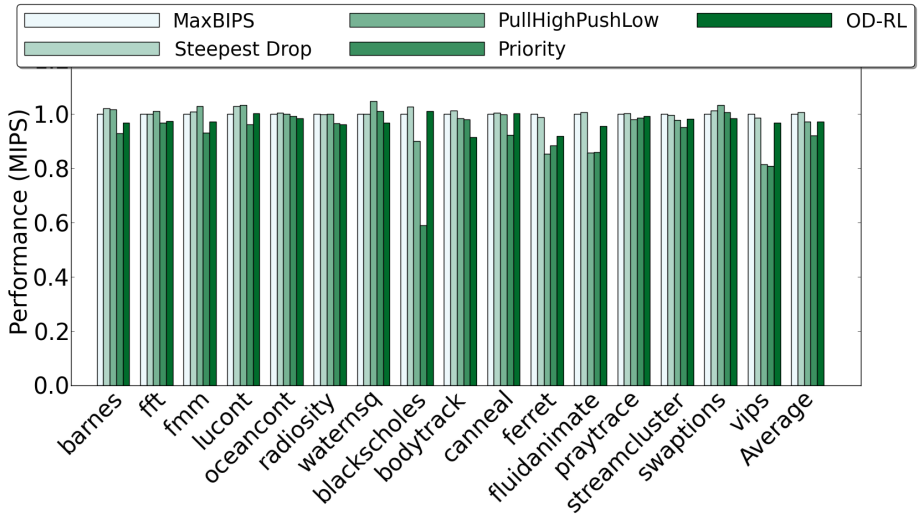



Figure 3.5: Comparison among five algorithms on system throughput (Lower is better). Results normalized with respect to MaxBIPS.

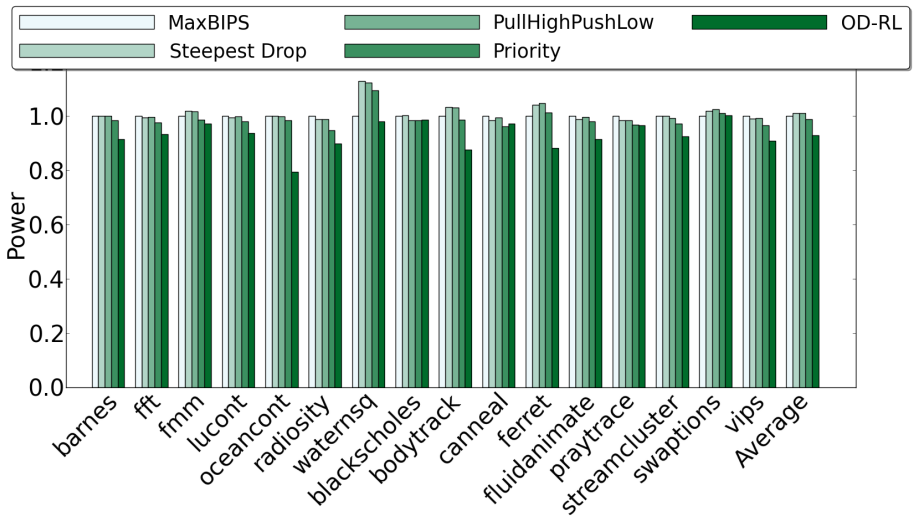



Figure 3.6: Comparison among five algorithms on power consumption (Lower is better). Results normalized with respect to MaxBIPS.

also much better budget overshoot control than prior work. We want to emphasize that, although both prior work and ours are maximizing performance under power budget, the former only consider that the average power across the entire run should be lower than the budget, while we also consider the budget overshoot within each run. We also point out that although prior work approaches try to control power at each epoch, inaccurate models make the decision inaccurate and this leads to temporary budget overshoot. Budget overshoot leads to extra heat generation and hence reliability issues. Budget overshoot is also linked to thermal emergencies which increase aging effects and decrease the mean time to failure (MTTF)[27]. We will explain why our method achieves better results, though we are using a similar modeling (we consider the leakage power as well).

As MaxBIPS and Steepest Drop are actually pushing the average power close to the budget line to achieve high performance regardless of the budget overshoot, we will also calculate $\frac{Throughput^2}{Average\ Power}(\propto \frac{1}{Energy-Delay\ Product})$ to evaluate the energy efficiency of all five methods. Fig. 3.7 compares the energy efficiency of all five approaches under different benchmarks, respectively. An 8-core system is used here because MaxBIPS (which uses

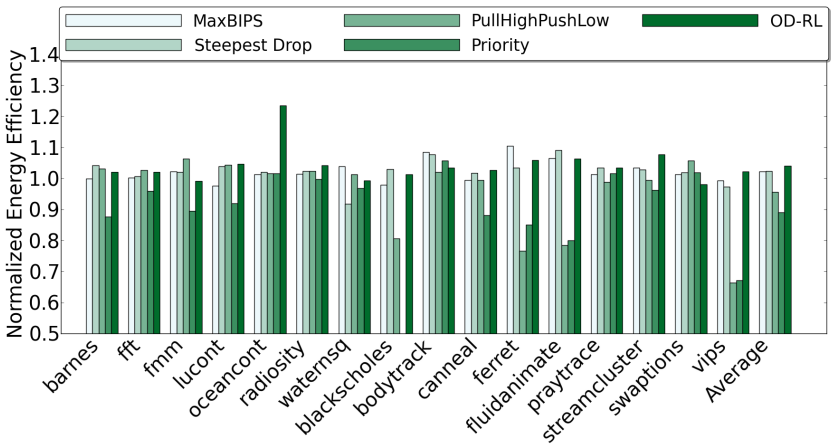



Figure 3.7: Comparison among three algorithms on energy efficiency (Higher is better). Results normalized with respect to MaxBIPS.

exhaustive search) is not scalable and thus becomes too slow for systems with more cores. We can see that OD-RL provides up to **23%** higher energy efficiency, and is slightly better than MaxBIPS and Steepest Drop, while much better than PullHighPushLow and Priority approaches on average, because MaxBIPS is already near optimal on average [3].

### 3.3.3 Power and performance constraints

In the following we consider both power and performance constraints. Since we have already shown that OD-RL outperforms Steepest Drop, MaxBIPS, Priority and PullHiPushLow in [3], we compare our pa-OD-RL with the priority-unaware OD-RL. Though our algorithm works for systems running any number of applications with arbitrary number of threads and priorities, we cannot experiment with all the possible combinations as they become prohibitively expensive to simulate. In the experiments, we simulate a 16 core system with two 8-thread applications randomly picked from `SPLASH-2` benchmark suites. Each of the applications has its performance requirement bounds, more specifically $p_{high}$ and $p_{low}$ for high priority and low priority applications, respectively. We set non-trivial performance bounds for both applications based on the VF set-

ting from Table 3.2. On one hand, the maximum throughput is obtained when running under 2.8GHz while the nominal frequency is 2.6GHz, so the high priority performance bound $p_{high}$ should be set larger than $2.6/2.8 = 92.8\%$. Otherwise, running at the nominal frequency will automatically satisfy both high and low priority performance bounds. On the other hand, $p_{high}$ should not be too close to 100%, because it will lead to most of the high priority application threads running at the highest VF level which is very likely to exceed the power budget and therefore be infeasible under both power and performance constraints. For the low priority application, the ratio of lowest frequency over highest frequency is set to $2.2/2.8 = 78.5\%$. Any bounds $p_{low}$ below 78.5% mean that the application will always be able to satisfy the constraint. However, setting $p_{low}$ too high means that there is little slack to give to the high priority application which may also yield no solutions. Based on the discussion above, we chose 95% performance bound for high priority applications and 80% for low priority applications.

As we have certain performance requirements, we define three additional metrics to provide a more comprehensive comparison involving both power and performance:

1. The **number of epochs** that both applications satisfy their prescribed performance constraints. We do not count the epochs during which at least one application does not satisfy the performance bound because in such cases, we consider that the system fails to satisfy the overall performance requirements.

2. **Accumulated Over-required-Throughput (AOT)**. This is an integral of the throughput values that are over the performance requirement, but only during epochs that both applications satisfy their performance requirements. This metric captures not only how well the algorithm satisfies the performance requirements, but also its ability to maximize throughput.

3. **Ratio of AOT over Budget overshoot (AOB)**. Since we have both power constraints and performance requirements, we use AOB as the metric characterizing the effec-

Table 3.3: Metrics ratio of pa-OD-RL over OD-RL.

| mix benchmarks | Budget overshoot | Energy efficiency |
|---|---|---|
| fft barnes | 0.664 | 1.010 |
| barnes radiosity | 0.790 | 1.022 |
| oceancont oceancont | 0.041 | 1.063 |
| lucont barnes | 0.566 | 1.024 |
| fft fft | 0.693 | 0.954 |
| radiosity barnes | 0.464 | 1.126 |
| lucont oceancont | 0.389 | 1.006 |
| fft lucont | 0.138 | 1.037 |
| cholesky lucont | 1.241 | 0.797 |
| radiosity oceancont | 0.198 | 1.249 |
| oceancont fft | 0.530 | 1.003 |
| barnes lucont | 0.255 | 0.989 |
| barnes oceancont | 0.380 | 0.894 |
| barnes radix | 0.452 | 0.957 |
| radiosity lucont | 0.031 | 0.963 |
| lucont lucont | 0.371 | 0.935 |
| fft oceancont | 0.095 | 0.971 |
| radix radiosity | 0.824 | 0.635 |
| fft radiosity | 1.089 | 0.996 |
| lucont cholesky | 0.769 | 0.926 |
| **Average** | **0.500** | **0.978** |

tiveness of the algorithm in satisfying both. We note that higher AOT and lower budget overshoot both lead to higher AOB values.

Table 3.3 shows the results of pa-OD-RL in terms of the budget overshoot and energy efficiency. All metrics are normalized with respect to the values of priority-unaware OD-RL. We can see that pa-OD-RL can achieve better budget overshoot control for workloads involving multiple applications with different priorities. Priority-unaware OD-RL cannot distinguish threads from different applications and hence is more likely to unbalance the threads within the same application. Pa-OD-RL achieves similar energy efficiency as the priority-unaware OD-RL on average while satisfying prescribed performance requirements.

Figures 3.8, 3.9 and 3.10 show log scale comparison between priority-unaware OD-RL

and pa-OD-RL in terms of the new three metrics, *i.e.*, number of epochs during which performance requirements are met, AOT and AOB. All bars are normalized with respect to the value of the priority-unaware OD-RL. In Figure 3.8, we see that in almost all cases, pa-OD-RL is better in satisfying both application performance requirements than the priority-unaware OD-RL version. This is expected since OD-RL is unaware of any performance requirements and prefers boosting up threads with the highest utilization. Figure 3.9 shows that pa-OD-RL achieves better AOT values which indicates better overall system performance under given per-application performance requirements. We see that for `radiosity` and `ocean.cont` mixed benchmark, pa-OD-RL has similar number of epochs satisfying both application performance requirements but much lower AOT. The reason behind this behavior is lower power budget overshoot (0.198x smaller) which leads to a lower throughput value. Figure 3.10 shows that pa-OD-RL is almost always better than OD-RL in terms of the combined power and performance requirements.

### 3.3.4 Overhead and scalability

To show how pa-OD-RL performs when the number of cores scales up, we compare Steepest Drop and OD-RL for a system of 16, 32 and 64 cores. We only compare Steepest Drop and OD-RL here because (i) pa-OD-RL has a similar scalability and runtime as OD-RL as we analyze and demonstrate below and (ii) MaxBIPS is unable to run for more than 8 cores. `Waternsq`, `bodytrack`, `ferret` and `fluidanimate` are used here because they are the most representative for our work. Figures 3.14, 3.15 and 3.16 show that OD-RL is constantly better when the number of cores scales up. OD-RL, on average, achieves around **70%** less budget overshoot and **3%** better energy efficiency. The average improvement for both metrics over different number of cores is relatively stable. Figures 3.11, 3.12 and 3.13 show the power and performance for the two approaches. OD-RL achieves around 13% reduction in power consumption, while only sacrifices around 4% of the performance. The consistent improvement across different number of cores

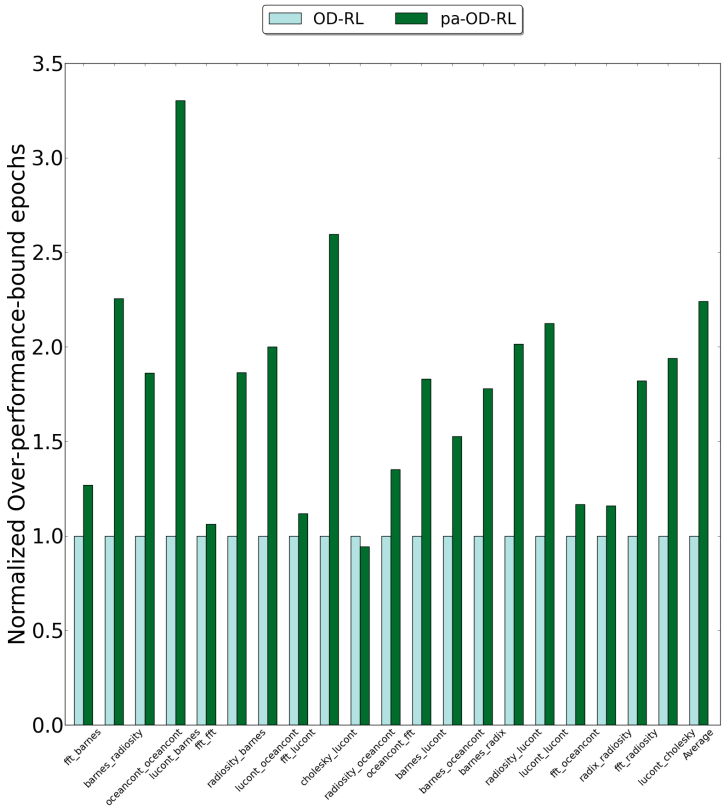



Figure 3.8: Comparison between OD-RL and pa-OD-RL for the number of epochs satisfying both performance constraints in a 16 core system running two 8-thread applications.

demonstrates that OD-RL method is highly scalable and largely independent on the number of cores.

Analyzing the time series of the performance and power values, we make the following observations: (i) MaxBIPS and Steepest Drop heavily rely on the accuracy of the $V^2f$ Scale Law. Inaccurate power estimation will make them choose actions that result in budget overshoot. Although OD-RL also suffers from that inaccuracy during the budget reallocation step. RL automatically adapts to the new budget and suppresses the budget overshoot. (ii) Even with perfect power estimation, MaxBIPS and Steepest Drop still suffer from making decisions based on the current machine state regardless of the next machine state. For example, let's assume a machine state $s_0$ at time $t_0$ and $s_1$ at time $t_1$ with $s_1 > s_0$. Here, a larger $s$ means a higher machine utilization. At time $t_0$, MaxBIPS and Steepest Drop will calculate the best actions based on $s_0$ which should

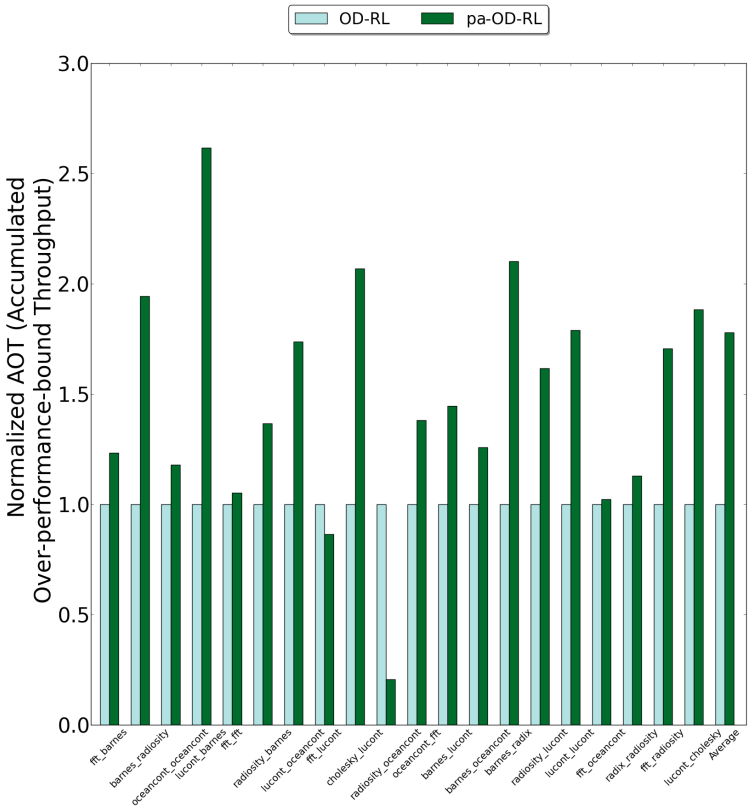



Figure 3.9: Comparison between OD-RL and pa-OD-RL for the AOT metric in a 16 core system running two 8-thread applications.

not exceed the budget. Unfortunately, $s_0$ quickly changes to $s_1$ under which the chosen actions cause a higher power consumption which actually exceeds the budget. The reason why RL performs better than MaxBIPS is because it has the capability of learning the transition probability of the machine state, and therefore, is able to predict the next machine state and choose the best action based on both current and future state.

We further analyze and test the overhead of the three approaches. MaxBIPS uses exhaustive search and its complexity is $O(N \cdot \alpha^N)$ where $\alpha$ is the number of VF levels. Steepest Drop is much more scalable with a complexity of $O(\alpha \cdot N \cdot log(N))$. In OD-RL, the distributed RL method is highly parallel and the only phase requiring global coordination is the power budget reallocation. MM actually does a heapsort with respect to the IPC value, and its worst case complexity is $O(N \cdot log(N))$ [28]. The priority-aware MM version does heapsort on the threads for each application, and the two

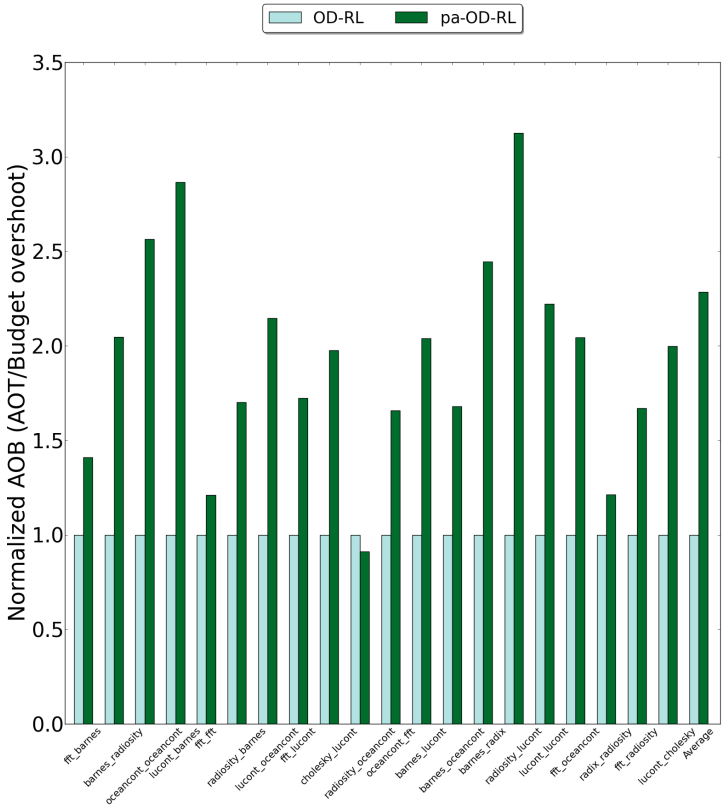



Figure 3.10: Comparison between OD-RL and pa-OD-RL for the AOB metric in a 16 core system running two 8-thread applications.

loops assigning power budget require $N$ operations at most. Assuming applications have similar number of threads ($N_{App}$), we can obtain the complexity of pa-MM as $O(N_{App} \cdot \frac{N}{N_{App}} \cdot log(\frac{N}{N_{App}}) + N) = O(N \cdot log(\frac{N}{N_{App}}))$. Therefore, if we have fixed number of applications, priority-aware MM and priority-unaware MM both have better scalability than MaxBIPS and Steepest Drop. In addition, since both OD-RL and pa-OD-RL invoke the power reallocation algorithm on a coarser temporal granularity, it has further smaller runtime overhead by a constant factor. Fig. 3.17 shows the execution time (log scale) of the Steepest Drop method, OD-RL and pa-OD-RL as a function of core count. The trend indeed follows the $O(N \cdot log(N))$ complexity. The execution time is averaged over 100 runs to account for the variation due to different algorithm inputs, *i.e.*, different values of IPC, MPKI, *etc*. The results show that OD-RL can achieve **100x** speedup in a 512-core system when compared to Steepest Drop. Pa-OD-RL is slightly slower than

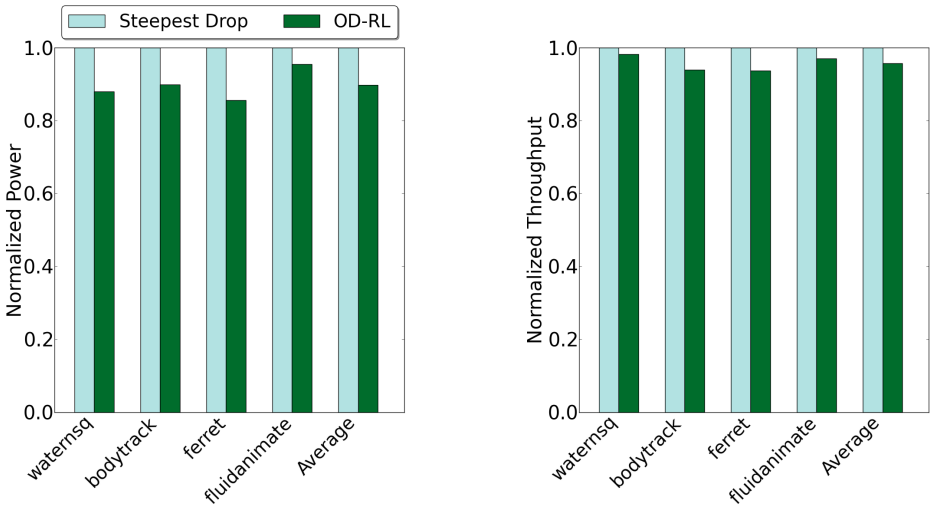



Figure 3.11: Comparison of power (left) and throughput (right) in a system of 16 cores.

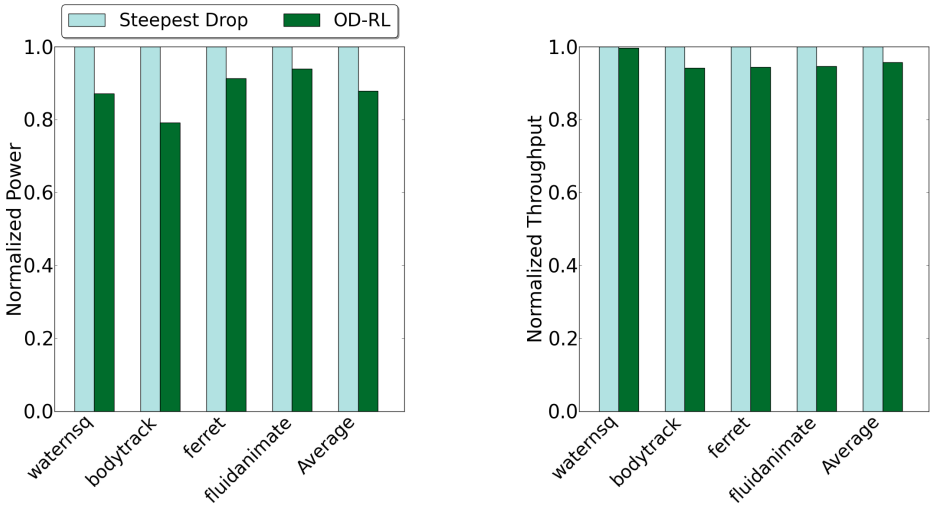



Figure 3.12: Comparison of power (left) and throughput (right) in a system of 32 cores.

OD-RL, because it needs to compute the total estimated throughput of each application after each budget assignment. Although the CO method [29] achieves a similar speedup, it is considering a different objective, which is the sum of frequencies, while we, similar to MaxBIPS and Steepest Drop, are optimizing the throughput defined as Instructions Per Second (IPS) = frequency * Instructions Per Cycle (IPC). Due to this fundamental difference in objective, we do not compare with CO in terms of speedup.

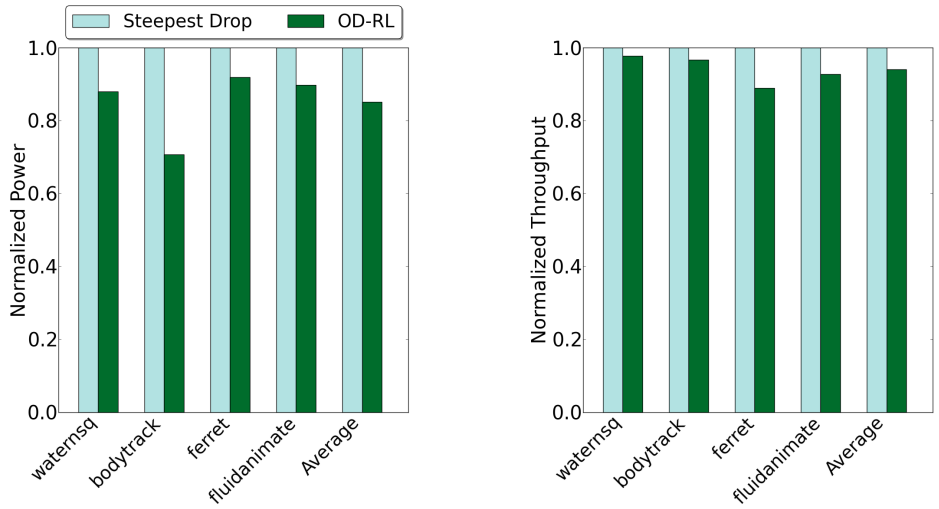



Figure 3.13: Comparison of power (left) and throughput (right) in a system of 64 cores.

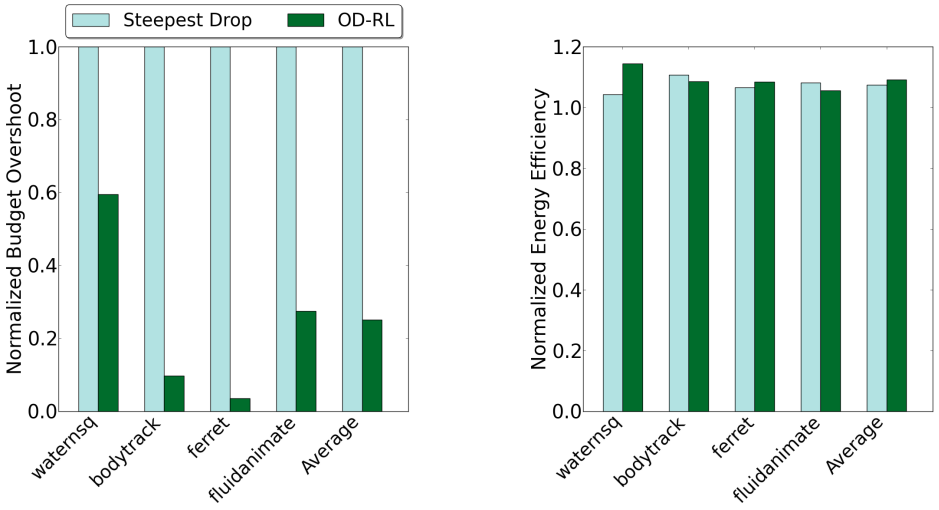



Figure 3.14: Comparison of budget overshoot (left) and energy efficiency (right) in a system of 16 cores.

## 3.4 Discussion

In this chapter, we propose an On-line Distributed Reinforcement Learning-based approach which decomposes the original power constrained, performance optimization problem into two sub-problems at different spatial and temporal granularities. The RL at finer grain is able to improve the performance while mitigating possible budget overshoot by learning and adapting to workload changes. The proposed Maximize-the-Max

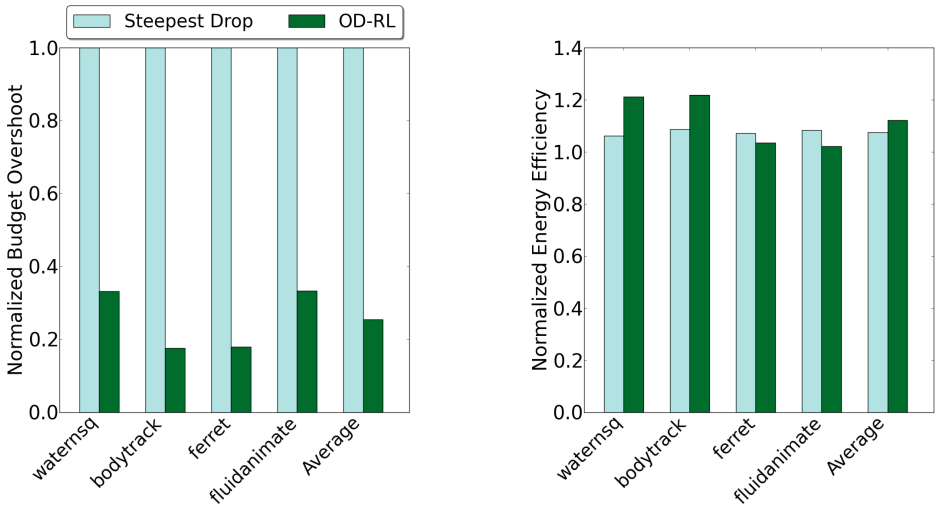



Figure 3.15: Comparison of budget overshoot (left) and energy efficiency (right) in a system of 32 cores.

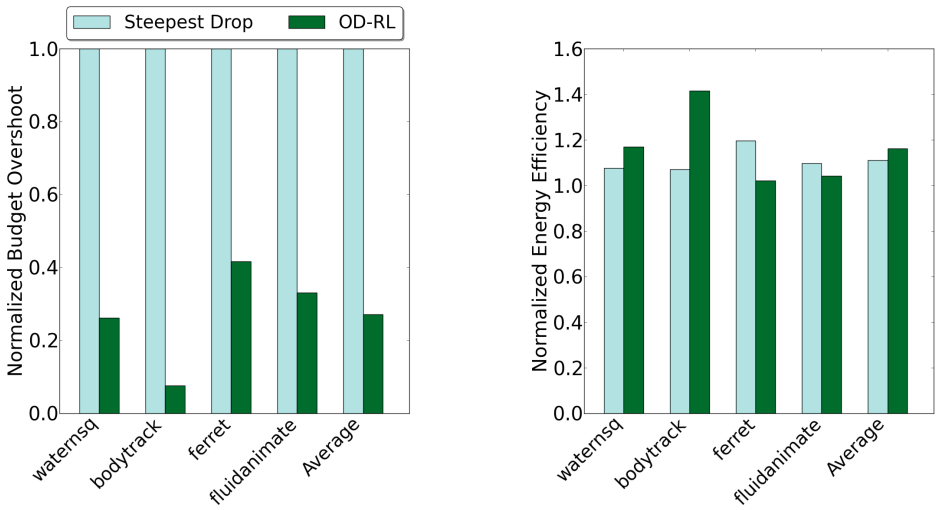



Figure 3.16: Comparison of budget overshoot (left) and energy efficiency (right) in a system of 64 cores.

(MM) method and its priority-aware variant run at a coarser granularity and are more scalable with respect to the number of VF levels compared to Steepest Drop, and thus, can be asymptotically faster since they are run less frequently. OD-RL is **100x** faster on a 512-core system comparing to Steepest Drop. Our approach achieves up to **98%** saving on budget overshoot, **23%** higher energy efficiency, and is consistently better than Steepest Drop when the number of cores scales up. Considering both global power con-

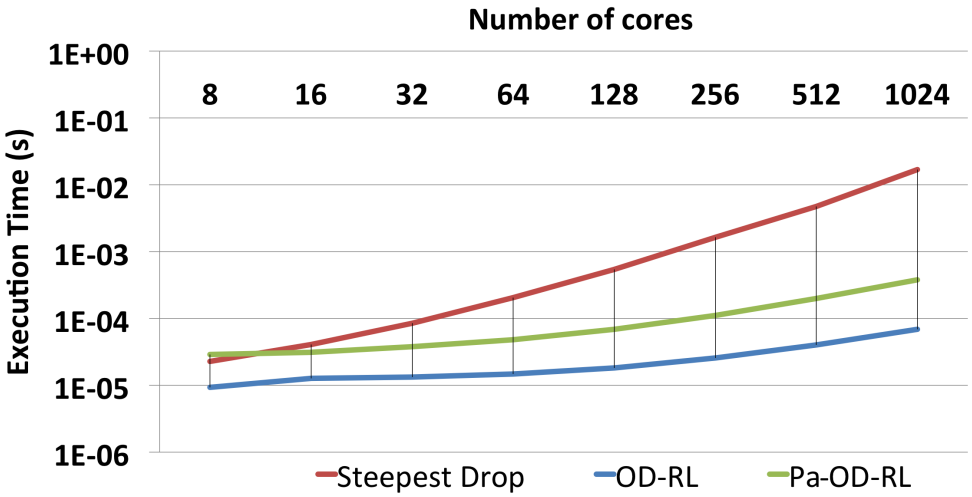



Figure 3.17: Log-log plot of approaches overhead in a system up to 1024 cores. Execution time is averaged over 100 runs.

straints and per application performance requirements, pa-OD-RL delivers, on average, (i) **17.8x** more epochs satisfying prescribed performance requirements, (ii) **5.6x** performance gain, and (iii) **20.0x** better performance-power trade-off (AOB metric) than the priority-unaware OD-RL, while both have similar runtime and scalability to thousand of cores.

# Chapter 4

# Speeding up training of RL-based DVFS algorithms

## 4.1 Chapter overview

In the age of mobile computing, reducing power consumption is in fact a major design goal as it extends battery life, improves thermal profile, increases device stability, *etc*. Dynamic Voltage Frequency Scaling (DVFS) is an effective technique for improving system energy efficiency and is widely available commercially [3, 19, 21, 30, 31, 32]. By strategically changing the Voltage and Frequency (VF) levels of the processing cores (CPUs) and/or main memory (DDR) as a function of the system state, one can save considerable amount of energy without sacrificing system performance, *e.g.*, a simple DVFS algorithm may lower the voltage and frequency of the cores when they process lightweight workloads to save energy. However, finding the optimal VF levels is NP hard [4]. Traditionally, engineers manually tune a rule-based DVFS (rule-DVFS) algorithm for each system platform and a set of benchmark applications, which requires long experimentation time and extensive human labor. The situation is compounded by the continual emergence of new and diverse applications, and more complex computing systems. In order to adapt to the ever changing world of computing, Reinforcement Learning-based

DVFS (RL-DVFS) algorithms have been proposed as they can learn the DVFS model and its parameters systematically through data [19, 33, 34]. RL-DVFS methods have been proven successful, however, they often require several thousands of iterations or more to train the model [35, 36]. Data collection in real world can be expensive due to physical limitations, *e.g.*, power meters are used for measuring device power, but are limited by measuring speed and quantity. For instance, measuring ten thousand iterations of one five-minute application takes one month. Hence, reducing training iterations is critical for learning the model in reasonable amount of time to make RL-DVFS practical, *i.e.*, reduce time to market.

Bayesian Optimization (BO) is a well-known method for sample-efficient optimization and has been extensively studied [10, 37]. In this chapter, we propose iterative-BO methods [38] for RL-DVFS algorithm training, resulting in a speedup in training time up to **37.4x**, relative to direct RL-DVFS training.

### 4.1.1 Chapter contributions

To the best of our knowledge, the work presented this chapter makes the following contributions:

- In BO-based approach for speeding up training, we **compress a traditional RL-based DVFS model** to fewer parameters, which not only speeds up RL training, but also enables the application of BO.
- We then propose an iterative-BO method and an improved version of the same. The iterative-BO method uses a dimensionality reduction strategy to enable better BO convergence properties. The improved version of iterative-BO method, which we call iterative-BO with restart, additionally leverages a **novel history forgetting strategy** to achieve an increased speedup of **37.4x** in RL-based DVFS training.

## 4.2 Methodology

### 4.2.1 RL-based DVFS

Dynamic tuning of the VF levels of CPU and DDR (also extensible to other parts of the mobile device) to maximize the system energy efficiency can be modeled as a Markovian Decision Process [19] [33] that traditionally admits solutions via RL [8]. The goal of RL is to find the best actions under different states such that a long-term reward $R$ is optimized. In our problem, since we care about both performance and power of the system, we define $R$ as the product of a performance reward and a power reward. We define the application execution time (a proxy for performance) and average power consumption (directly related to mobile device battery life) under our method as $\mathcal{T}_{RL}$ and $\mathcal{P}_{RL}$, respectively, and the execution time and power under a rule-DVFS as $\mathcal{T}_{rule}$ and $\mathcal{P}_{rule}$. Performance reward $R_{perf} = 1$ when $\mathcal{T}_{RL} <= \mathcal{T}_{rule}$ and $R_{perf} = \mathcal{T}_{rule} - \mathcal{T}_{RL}$, which is a negative value, when $\mathcal{T}_{RL} > \mathcal{T}_{rule}$. Power reward $R_{power}$ is defined as $max(0, 1 - \frac{\mathcal{P}_{RL}}{2\mathcal{P}_{rule}})$ which indicates a lower power preference. Accordingly, rule-DVFS has a reward of 0.5, and any algorithm with a reward higher than 0.5 achieves better energy efficiency than rule-DVFS. We further define the state of the system in RL as $\mathcal{S} = [s, f]$, where $s$ is a feature vector containing values of various counters measuring different attributes in the system and $f$ is the index of current VF level [19]. We model the next VF level decision $f$ as a function of $\mathcal{S}$.

### 4.2.2 Cross-entropy method

Cross-Entropy Method (CEM) is a well-known RL algorithm [39], and we use it as the baseline RL-DVFS method for our problem. We model the CPU and DDR control policies separately as they are distinct components in the mobile device, and this also helps dealing with the BO curse of dimensionality later: $f_{cmpt} = argmax(W_{cmpt} \cdot \mathcal{S}')$, where $f$ is the index of selected VF level, $cmpt$ is device component (CPU or DDR), $W_{cmpt}$ is a

2D parameter matrix. $\mathcal{S}' = [\mathcal{S}, 1]$ includes a bias term and seven system state counter values, *e.g.*, communication on the bus connecting CPU and DDR, and the system cache miss rate (indicating how often the DDR got accessed). We call this the CEM-LC model, characterized by 184 parameters, with $N_{fcpu} = 13$ CPU frequencies and $N_{fddr} = 10$ DDR frequencies. In our experiments, the CEM-LC model takes 4,000 iterations of training to surpass a rule-DVFS method. In order to speed up training, we first propose a simplified model: $f_{cmpt} = round[(N_{fcmpt} - 1) \cdot Sigmoid(V_{cmpt} \cdot \mathcal{S}')]$, where $V_{cmpt}$ is a 1D parameter vector. We use $(N_{fcmpt} - 1)$ to scale up the sigmoid output because the frequency index starts from zero. We call this the CEM-LR model, characterized by only 16 parameters. Our experimental results show that CEM-LC and CEM-LR deliver similar energy efficiency as will be shown in Figure 4.1.

### 4.2.3 Iterative-BO

We apply BO to speed up the training of our DVFS algorithm, that uses the CEM-LR model described above. Although CEM-LR drastically reduces the number of parameters from 184 to 16, it is still not small enough in dimensionality for BO to be effective [40, 15]. To tackle this problem, we propose the iterative-BO method (see Algorithm 2 with *Method* set to *iterative-BO*) to decouple CPU and DDR model optimization steps, so that we effectively only optimize eight parameters at a time, *i.e.*, while the CPU model parameters are being optimized using BO, the DDR model parameters are held fixed to the optimal values from the previous iteration (and vice versa). In each iteration, we optimize each component using $M = 50$ iterations, which is determined by experiments. This is different from simply converting to low dimensional subspace by picking a subset of variables, because by iterating, we maintain all variables and still optimize both components in the system as they have interactions between them. We use a Gaussian Process (GP) as the prior for our BO algorithm. Since the choice of acquisition function, covariance kernel and their hyperparameters are problem dependent and may highly

affect the effectiveness of BO method, we do grid search to find the best choices for them.

**Algorithm 2** Pseudocode of iterative-BO method (with restart)

Input: $MaxIter$, $N_{fcpu}$, $N_{fddr}$, $Method$, $M$
Output: $V_{CPUbest}$, $V_{DDRbest}$, $MaxReward$

//Randomize parameters for CPU and DDR models at iteration zero
$V_{CPUbest} \leftarrow random()$; $V_{DDRbest} \leftarrow random()$;
$cmpt \leftarrow [CPU,\ DDR]$; $MaxReward \leftarrow 0$;
Initiate $BO_{CPU}$, $BO_{DDR}$
**for** $i \leftarrow 1; i \leq MaxIter; i \leftarrow i+1$ **do**
    **for** $j \leftarrow 0; j \leq 1; j \leftarrow j+1$ **do**
        **if** $Method$ == iterative-BO **then**
            $reward, V_{cmpt[j]} \leftarrow$ resume $BO_{cmpt[j]}$ on $System(V_{cmpt[j]}, V_{cmpt[1-j]} \leftarrow V_{cmpt[1-j]best})$ for $M$ iterations ▷ //Resume $BO_{cmpt[j]}$, *i.e.*, keep previous observations
        **else if** $Method$ == iterative-BO with restart **then**
            $reward, V_{cmpt[j]} \leftarrow$ restart $BO_{cmpt[j]}$ on $System(V_{cmpt[j]}, V_{cmpt[1-j]} \leftarrow V_{cmpt[1-j]best})$ for $M$ iterations ▷ //Restart $BO_{cmpt[j]}$, *i.e.*, clear previous observations
        **end if**
        **if** $reward > MaxReward$ **then**
            $MaxReward \leftarrow reward$; $V_{cmpt[j]best} \leftarrow V_{cmpt[j]}$;
        **end if**
    **end for**
**end for**

When iterating between $BO_{CPU}$ and $BO_{DDR}$, note that we retain the observations from previous iterations as they may help guide the learning of the target function. However, our later experiments showed that because at the beginning of iterative-BO, parameters are close to random, they change drastically after a few iterations. Those early observations therefore quickly become wrong estimations of the reward function and hinder the optimization process. Accordingly, we propose *iterative-BO with restart*, which restarts BO at the beginning of each iteration (see Algorithm 2 with *Method* set to *iterative-BO with restart*). With restart, results show greatly improved speedup of RL-DVFS training (see Figure 4.1), confirming that our intuition to forget the historical BO context is helping significantly.

## 4.3 Experimental results

### 4.3.1 Experiment setup

We show the effectiveness of our proposed methods by testing on extensive combinations of smartphone workloads. Our training and testing datasets comprise of several diverse real-world workload snippets. We concatenate workloads randomly to produce a test benchmark. The power and performance values of the rule-DVFS algorithm as well as the features of all benchmark snippets are measured on real mobile chipsets. We implement CEM-LC and CEM-LR following [39], and use the BO package: BayesianOptimization [1] for BO optimization. We modify the source code to experiment with different kernel hyperparameters and implement our iterative-BO methods.

### 4.3.2 Experimental results

CEM has two hyperparameters: batch size and noise. We performed grid search to find the best value of batch size $= 200$ and noise $= 0.01$ that surpasses the rule-based method with fewest iterations. As we pointed out before, the choice of acquisition functions, kernels and their parameters may highly affect the results of BO [10]. We experiment with various choices to determine the best values for them. For the acquisition function, we tried Expected Improvement (EI) and Upper Confidence Bound (UCB), both of which have been widely used [11]. EI has no hyperparameters, while UCB has one hyperparameter $\kappa$ that trades off between exploitation and exploration. Squared Exponential kernel is often used in BO, however it is considered unrealistically smooth for many engineering problems [11]. As a result, we pick Matern kernel and experiment with hyperparameter $\nu = 0.5,\ 1.0,\ 2.5$ as suggested by [10]. Also, these values compute considerably faster due to the modified Bessel function in Matern [41][10]. Our results show that a GP prior with Matern kernel $\nu = 2.5$ and unit scale length, and UCB with $\kappa = 0.5$ as the acquisition function gives the best results.

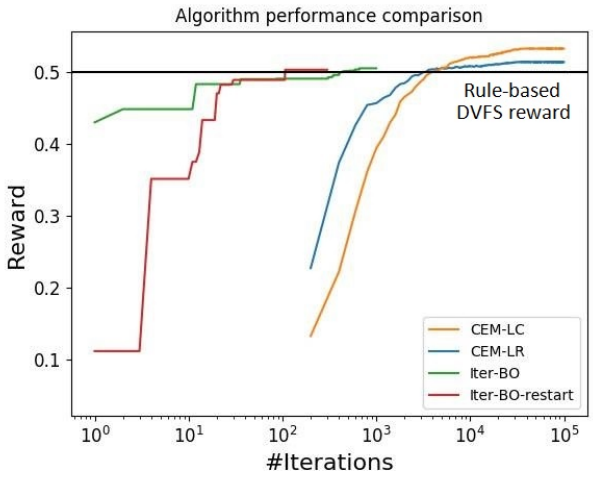



Figure 4.1: Sample efficiency of various RL DVFS training methodologies.

CEM and BO solve for the model parameters to maximize the reward $R$, the indicator of system energy efficiency. Rule-DVFS method has a reward value of 0.5, and this is the value our methods aim to surpass with fewer iterations. Figure 4.1 shows the reward *vs.* iteration (in log scale) for all four methods under the best parameters chosen above. We can see that all methods surpass rule-DVFS (horizontal line at 0.5). CEM methods start from 200 iterations because the reward values are evaluated after each batch which has a size of 200.

When compared to CEM-LC, model reduction (CEM-LR) gives a **1.2x** speedup (blue curve in Figure 4.1), while iterative-BO delivers a **9.1x** speedup (green curve in Figure 4.1) based on the reduced model. The benefits of iterative-BO come from two sources: 1) the reduced model size from CEM-LR; indeed the number of parameters is now reduced to 16 from 184 and hence the searching space of BO is much smaller; and 2) decomposition of the system into two devices: CPU and DDR, each with eight parameters. This is a critical step because BO typically handles problems with less than ten parameters due to the scalability issue [15, 37]. By iterating between two devices, each with fewer parameters, BO can successfully and quickly find the solution that is better than rule-DVFS. We also experimented with BO on the joint CPU+DDR system (without splitting the 16-parameter search space into two 8-parameter search space). BO on the

joint system does not even reach the rule-DVFS method in the experiment time horizon considered here. We believe this is due to the curse of dimensionality problem for BO.

*Iterative-BO with restart* is able to further boost the speedup to **37.4x** (red curve in Figure 4.1), which demonstrates that discarding incorrect history helps learn the target function much faster. Initial values do not matter much for *Iterative-BO with restart* as its reward value improves much faster than Iterative-BO.

## 4.4 Discussion

Dynamic Voltage and Frequency Scaling (DVFS) is an important technique widely available in improving modern computing system energy efficiency. Although RL-based DVFS methods are effective in system efficiency optimization, they can be sample inefficient. In this chapter, we proposed a hybrid method combining model complexity reduction and iterative-Bayesian Optimization method to overcome the curse of dimensionality of BO methods and speed up training of RL-based DVFS control algorithms. Based on deeper investigation of the iterative-BO method, we propose *iterative-BO with restart* which further boosts speedup to **37.4x**.

# Chapter 5

# Convolutional neural networks: efficient inference

## 5.1 Chapter overview

Deep Convolutional Neural Networks (CNNs) have gained tremendous traction in recent years thanks to their outstanding performance in visual learning tasks, *e.g.*, image classification and object detection. However, CNNs are often considered very computationally intensive and energy demanding [5, 6, 7, 42, 43]. With the prevalence of mobile devices, being able to run CNN-based visual tasks efficiently, in terms of both speed and energy, becomes a critical enabling factor of various important applications, *e.g.*, augmented reality, self-driving cars, Internet-of-Things, *etc*, all of which heavily rely on fast and low energy CNN computation. To alleviate the problem, engineers and scientists proposed various solutions, including sparsity regularization, connection pruning, model quantization, low rank approximation, *etc*. In this work, we propose a complementary approach, called *Virtual Pooling* (ViP), which takes advantage of pixel locality and redundancy to reduce the computation cost originating from the most computationally expensive part of CNN: convolution layers. As illustrated in Figure 5.1, ViP reduces computation cost by computing convolution with a larger (2x) stride size. While natu-

rally this operation quickly shrinks the output feature map, and thus can only be done a few times before the image vanishes, we overcome this problem by recovering the feature map via *linear interpolation* with provable error bound. The succeeding layer hence observes the same size of input with or without ViP, and no architectural change is needed. Our experimental results on different CNN models and tasks show that we can achieve **2.1x** speedup with 1.5% accuracy degradation in image classification, compared to the 1.9x speedup with 2.5% degradation from prior work [44], and **1.8x** speedup with 0.025 mAP degradation in object detection.

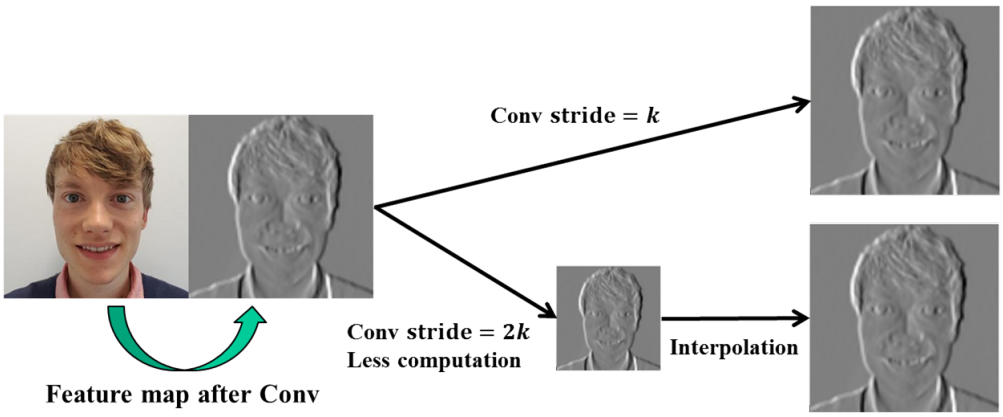



Figure 5.1: Illustration of virtual pooling [2]. By using a larger stride, we save computation in convolution layers and, to recover the output feature map, we use linear interpolation which is fast to compute.

### 5.1.1 Chapter contributions

To the best of our knowledge, the work presented this chapter makes the following contributions:

- We are the first to propose and implement the Virtual Pooling (ViP) method with provable error bound. ViP is independent of the dataset and can be applied to accelerate any convolution layer.

- Plug-and-play: ViP is a self-contained custom layer. Without modifying the deep learning framework, it works simply by doubling the stride of the convolution layer and inserting the ViP layer after it.

- Rather than providing a single CNN configuration, ViP is able to generate a set of CNN models with varying speedup/energy-accuracy trade-offs that a machine learning practitioner can select the right model for the task at hand.

- Most CNN acceleration techniques consider only the image classification task, while they lack evidence on how their performance may translate to the object detection task, which has its own unique properties. In this work, we conduct experiments to show that ViP also works well under the state-of-the-art faster-rcnn object detection framework.

## 5.2 Methodology

Virtual Pooling (ViP) relies on the idea of reducing CNN computation cost by taking advantage of pixel spatial locality and redundancy. CNNs are often comprised of multiple convolution layers (accompanied by non-linear ReLU functions) interleaved with pooling layers. Pooling layers are considered essential for reducing spatial resolution such that computation cost is reduced and robustness to small distortions in images is enhanced. However, the widely-used stride-two non-overlapping pooling method [45, 46] reduces image size by half in each of the two dimensions, and thus quickly shrinks the image. As a result, the maximum number of pooling operations that can be done in a CNN is limited by the size of the input image. For example, an input image of size $224 * 224$ is shrunk to size $7 * 7$ after only five pooling layers [45], while the current state-of-the-art CNNs usually have several tens to hundreds of layers [45, 46]. There is an opportunity to reduce computation further if we can bridge the gap between the number of pooling-like operations we can do and the number of layers in the network.

### 5.2.1 ViP layer

To this end, we propose ViP, a method that can maintain the size of output of each layer, while using a non-unit-stride sliding-window operation, *e.g.*, a stride-two convolution. Consequently, we can have as many ViP layers as possible while not encountering the problem of diminishing image size in the real pooling operation. While it is possible to increase the stride of an early layer and remove a later pooling layer (without interpolation) to achieve a similar effect, our experiments show that ViP is constantly better than pooling removal with 1.42% higher accuracy on average. Furthermore, this method can only reduce computation in consecutive convolutional layers prior to pooling, while ViP works in any order (as we will show later, accuracy sensitivity is non-monotonic with the network layer) which gives a better accuracy-speedup curve. As illustrated in Figure 5.1, the idea of ViP is that we save computation by performing a larger stride convolution in the layer before ViP, and then recover the output size by simple linear interpolation which can be computed very efficiently. For example, VGG-16 uses stride-one convolution for all convolution layers. By applying ViP after each convolution layer, we can increase the stride of the preceding convolution layer to two for reduced computation, and recover the image size by linear interpolation, so the succeeding layer observes an input feature map of exactly the same size. With a stride-two convolution, we have a theoretical speedup of 4x because we reduce the number of convolution operations by half in each of the two dimensions of the input. Though it is possible to increase the stride to a large number for higher latency improvement, we expect a bigger accuracy drop, and as a result, the stride-to-use should be determined based on the requirements of the application.

To be more specific, let's use $\mathcal{I}$ to denote the input to the convolution layer and $\mathcal{O}$ to denote the output. Without loss of generality, although $\mathcal{I}$ and $\mathcal{O}$ are often four-dimensional, we omit the first dimension of batch index because ViP is applied to all images in the batch independently, and therefore, $\mathcal{I}_{c,h,w}$ and $\mathcal{O}_{c,h,w}$ both have three di-

mensions: channel $c \in [1, C]$, height $h \in [1, H]$, and width $w \in [1, W]$. We consider convolution filters, $\mathcal{W}_{c',c,m,n}$, with the same height and width with odd values $M$ as are commonly used in CNNs [45, 46], and with $c'$ representing the index of the filter. For the purpose of simplicity, we further assume $H$ and $W$ are even numbers, *e.g.*, input image size of ImageNet is usually $224 * 224$, and in the case of odd numbers, we have special cases only on the boundaries of the image that are easy to deal with. Furthermore, we use $\mathcal{O}^{Orig}_{c',h,w}$ to represent the output of the original stride-$s$ convolution without ViP, and $\mathcal{O}^{ViP}_{c',h,w}$ to denote the output of using ViP method, *i.e.*, the output of stride-2$s$ convolution plus linear interpolation. A smaller $||\mathcal{O}^{Orig}_{c',h,w} - \mathcal{O}^{ViP}_{c',h,w}||_2$ indicates a smaller perturbation of the truth output and hence, less accuracy degradation for the ViP method. According to the definition of convolution:

$$\mathcal{O}^{Orig}_{c',h,w} = \sum_{c=1}^{C} \sum_{m,n=-\lfloor \frac{M}{2} \rfloor}^{\lfloor \frac{M}{2} \rfloor} \mathcal{I}_{c,s\cdot h-m,s\cdot w-n} * \mathcal{W}_{c',c,m,n} \tag{5.1}$$

If we double the stride, we have an output with reduced size:

$$\mathcal{O}^{Red}_{c',h,w} = \sum_{c=1}^{C} \sum_{m,n=-\lfloor \frac{M}{2} \rfloor}^{\lfloor \frac{M}{2} \rfloor} \mathcal{I}_{c,2s\cdot h-m,2s\cdot w-n} * \mathcal{W}_{c',c,m,n} \tag{5.2}$$

For ease of explanation, we use an auxiliary function $\mathcal{O}^{Zero}_{c',h,w}$ which is zero-spaced to enlarge $\mathcal{O}^{Red}_{c',h,w}$ to the same size of $\mathcal{O}^{Orig}_{c',h,w}$ in the following way:

$$\mathcal{O}^{Zero}_{c',h,w} = \begin{cases} \mathcal{O}^{Red}_{c',h/2,w/2} & h,\ w \text{ are even numbers} \\ 0 & \text{Otherwise} \end{cases} \tag{5.3}$$

We approximate the output with the ViP method $\mathcal{O}^{ViP}_{c',h,w}$ by using the mean of its non-zero immediate neighbors (including itself, if computed exactly) in $\mathcal{O}^{Zero}_{c',h,w}$:

$$\mathcal{O}^{ViP}_{c',h,w} = \frac{\sum_{m,n=-1}^{1} \mathcal{O}^{Zero}_{c',h+m,w+n}}{\sum_{m,n=-1}^{1} \mathbb{1}(\mathcal{O}^{Zero}_{c',h+m,w+n} \neq 0)} \tag{5.4}$$

This is actually a convolution with $3 * 3$ filters, however with variable weight values depending on the number of non-zero neighbors. We can simplify the above computation

by considering four different cases similar to Equation 5.3:

$$\mathcal{O}^{ViP}_{c',h,w} = \begin{cases} \mathcal{O}^{Red}_{c',h/2,w/2} & h \text{ even, } w \text{ even} \\ \frac{1}{2}(\mathcal{O}^{Red}_{c',\lfloor h/2 \rfloor,w/2} + \mathcal{O}^{Red}_{c',\lceil h/2 \rceil,w/2}) & h \text{ odd, } w \text{ even} \\ \frac{1}{2}(\mathcal{O}^{Red}_{c',h/2,\lfloor w/2 \rfloor} + \mathcal{O}^{Red}_{c',h/2,\lceil w/2 \rceil}) & h \text{ even, } w \text{ odd} \\ \frac{1}{4}(\sum_{\substack{h=\lfloor h/2 \rfloor \text{ or } \lceil h/2 \rceil \\ w=\lfloor w/2 \rfloor \text{ or } \lceil w/2 \rceil}} \mathcal{O}^{Red}_{c',h,w}) & h \text{ odd, } w \text{ odd} \end{cases} \tag{5.5}$$

The above equations are embarrassingly parallel and hence fast to compute on GPU. We implemented our custom ViP layer based on Equation 5.5.

Based on the definition of ViP operation above, we can further provide an error bound, considering the case where we apply ViP to layer $l_s$.

**Proposition 1.** *Assume the output of layer $l_s$ (hence input to layer $l_{s+1}$), $\mathcal{O}^{(l_s)}$, is L-Lipschitz continuous [47] on height and width dimensions $(h, w)$, i.e.,* $|\mathcal{O}^{(l_s)}_{c,h_1,w_1} - \mathcal{O}^{(l_s)}_{c,h_2,w_2}| \leq L||(h_1, w_1) - (h_2, w_2)||_2$, *for* $\forall h_1, h_2 \in [1, H], w_1, w_2 \in [1, W]$. *Assume that $\forall c', l$, the $c'$-th convolutional filter of the $l$-th layer, denoted as $\mathcal{W}^{(l)}_{c'}$, has a bounded $l_2$-norm: $||\mathcal{W}^{(l)}_{c'}||_2 = \sqrt{mean^2(\mathcal{W}^{(l)}_{c'}) + std^2(\mathcal{W}^{(l)}_{c'})} \leq B^{(l)}$. Then, the $l_2$-norm of the output error is bounded by:*

$$\begin{aligned} &||\mathcal{O}^{(l_e)ViP} - \mathcal{O}^{(l_e)Orig}||_2 \\ \leq &\sqrt{2}L\sqrt{C'^{(l_e)}H^{(l_e)}W^{(l_e)}} \prod_{l=l_s+1}^{l_e} \sqrt{C^{(l)}}M^{(l)}B^{(l)}, \end{aligned} \tag{5.6}$$

*where $C^{(l)}$ and $M^{(l)}$ are the number of input channels and kernel size of the $l$-th layer, respectively, and $C'^{(l_e)}$ is the number of output channels of the $l_e$-th layer.*

*Proof.* Deferred to Appendix. □

If $\forall l > l_s, \sqrt{C^{(l)}}M^{(l)}B^{(l)} > 1$, the upper-bound will keep increasing when the output goes through multiple layers. This indicates that earlier ViP layers with more succeeding layers may have a higher impact on the final output of the network and hence higher accuracy drop without finetuning. This actually reflects the intuition that perturbations

**Algorithm 3** Virtual Pooling (ViP)

1: **Input:** model $Net$
2: **Output:** ViP model $ViPNET$, Accuracy $ViPA$, Runtime $ViPR$
3:
4: // Sensitivity analysis
5: $i = 0$
6: $ViPLayers$ =[]
7: **for** $c$ **in** $Net.ConvLayers$ **do**
8: $\quad A_c$ = evaluate($Net.ViP(c)$)
9: $\quad ViPLayers.append((c, A_c))$
10: **end for**
11: $ViPLayers.sorted(key = A_c, 'descending')$
12:
13: //Progressively interpolate and finetune
14: $ViPA$ = [], $ViPR$ = []
15: **for** $j = 0 : len(ViPLayers)$ **do**
16: $\quad Net = Finetune(Net.ViP(0 : j))$
17: $\quad ViPA.append(evaluate(Net))$
18: $\quad ViPR.append(time(Net))$
19: **end for**
20: Return $ViPNET = Net$, $ViPA$, $ViPR$

from early layers will lead to higher error on the output as they propagate through the network. We will see this effect in both VGG-16 (Figure 5.3) and ResNet-50 (Figure 5.6).

### 5.2.2 ViP algorithm

While speeding up CNNs can be achieved with ViP, it may also lead to some accuracy drop since interpolation is a method of approximation. Therefore, we propose the following procedure, as shown in Algorithm 3, as part of the ViP method to reduce the accuracy degradation while maximizing the speedup we can achieve via ViP. Though our major target is speeding up the inference, ViP accelerates training as well as a result of the larger stride convolution. In Algorithm 3, we first do sensitivity analysis to detect which layers are less sensitive, in terms of the accuracy of the network, to the ViP operation (Line 7-10). For each of the convolution layers $c$, we insert ViP immediately after it, and evaluate the network accuracy $A_c$ without fine-tuning. The sensitivity is

measured as the accuracy drop with respect to the original accuracy. A larger accuracy drop means that the layer is more sensitive to the ViP operation, while a lower accuracy drop means that the layer is robust to ViP. This is equivalent to sorting the sensitivity array $A_i$ in descending order as shown in Line 11. There is one implementation detail on where to insert the ViP layer, as it can either be inserted after the convolution layer and before the ReLU layer, or after ReLU layer following the convolution layer. Both our experiments and prior work [44] show that inserting after ReLU gives better results. Our intuition is that by applying ViP before ReLU, we obtain less activations than the original without ViP and the network becomes less likely to identify smaller activation regions. Therefore, throughout the chapter, whenever we mention inserting ViP after a certain convolution layer, we mean inserting it after the ReLU layer that immediately follows it.

Based on the sorted per-layer sensitivity $ViPLayers$, we insert ViP layers progressively, and fine-tune the network after each interpolation to achieve a set of CNN models with different speedup-accuracy trade-offs (Line 15-19). For example, in the case of adding one ViP layer at a time, we add ViP after the $ViPLayers[0]$, fine-tune the model and obtain the first model, and then we add ViP after both $ViPLayers[0]$ and $ViPLayers[1]$, fine-tune the model and obtain the second model, and so on so forth. In this fashion, we will eventually generate $len(ViPLayers)$ models ($len(ViPLayers)$ is the total number of convolution layers that we apply ViP to), all with different accuracy and runtime. However, repetitively fine-tuning the model $len(ViPLayers)$ times can be quite time-consuming, especially for large CNN models. To alleviate this problem, we conduct grouped fine-tuning, in which rather than progressively inserting ViP one layer at a time, we insert several ViP layers at a time (still based on sensitivity values). This results in fewer rounds of fine-tuning, and hence less time, and both per-layer and grouped fine-tuning methods can generate different accuracy-speedup trade-offs for the baseline CNN model. While we show in the experiments that this accuracy sensitivity-based heurisitic delivers good results, applying Reinforcement Learning (RL) [48] to learn after

which convolution layer to insert ViP may lead to high speedup with minimal accuracy drop. However, RL-based approaches take an extremely long time to learn the optimal solution, as they require fine-tuning of the entire CNN for each training sample of RL, which could offset the simplicity and ease-to-use of the method of virtual pooling.

Figure 5.2 further illustrates how the ViP method can be applied to applications, such as a face detector in a mobile camera system. In mobile phone cameras, a face detector cannot only show where the faces are, but also help camera auto-focus for taking better pictures with people in sharp imaging. Therefore, in this case lower accuracy can be tolerated as long as some of the faces, if not all, are detected, while there is also speed requirement for camera to focus as fast as possible. For such applications, we can generate a set of models using the ViP method. We start from a pre-defined or pre-trained model, *e.g.*, Faster-RCNN, and a pre-collected dataset for human faces. Then we perform Per-layer Sensitivity Analysis (PSA) to determine the sensitivity of each convolution layer in the network. Based on the sensitivity, we can perform either grouped or per-layer finetuning to generate new models with higher speedup until we obtain the model that best satisfy the requirement on speedup-accuracy trade-off.

As it will be shown in Section 5.3, through the ViP method, we can speed up CNNs with minor accuracy drop. Furthermore, we can obtain a set of models with different speedup-accuracy trade-offs by applying ViP to various number of layers. As a result, our ViP method provides a knob for CNN practitioners to make trade-offs between accuracy and speedup based on the need of their applications. In addition, one can apply ViP on top of existing model acceleration methods, *e.g.*, model compression [6, 49], CNN binarization [50, 51], low rank approximation [52, 5], *etc*., to squeeze more performance out of the CNN models.

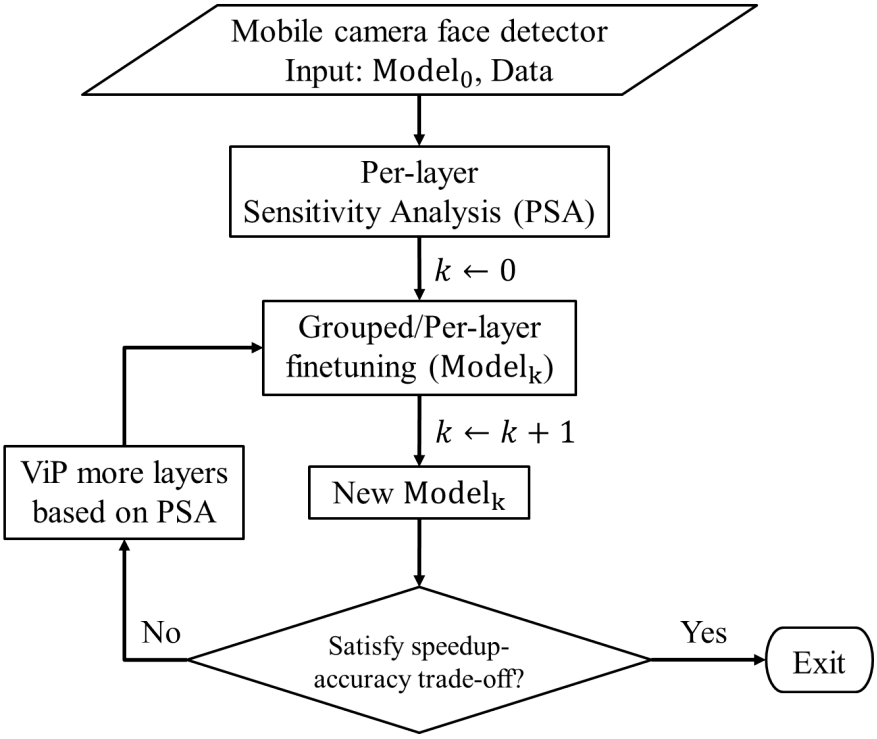



Figure 5.2: An example of applying ViP to the mobile phone camera face detector. ViP progressively generates new models with higher speedup until we obtain the model that best satisfy the requirement on speedup-accuracy trade-off.

## 5.3 Experimental results

In this section, we first describe the hardware and software setup of our experiments, and then present results to show the effectiveness of ViP method under:

1. Four CNN models: VGG-16 [45], ResNet-50 [46], All-CNN [53], Faster-RCNN with VGG-16 backbone [54] .

2. Three datasets: ImageNet [55], CIFAR-10 [56], PASCAL-VOC [57].

3. Two hardware platforms: Desktop and Mobile.

4. Two visual learning tasks: Image classification and object detection.

Table 5.1: System Configurations for desktop and mobile platforms.

| Desktop | |
|---|---|
| CPU/Main memory | Intel Core-i7 / 32GB |
| GPU/Memory | Nvidia Titan X / 12GB |
| DL platform | Caffe on Ubuntu 14 |
| **Mobile (Nvidia Jetson TX1)** | |
| CPU/Main memory | Quad ARM A57 / 4GB |
| GPU | Nvidia Maxwell Arch |
| DL platform | Caffe on Ubuntu 14 |

### 5.3.1 Experimental setup

Throughout the experiments, we use Caffe [58] as our deep learning platform since its correctness has been validated by numerous research works. For fast training and inference, we implement a self-contained custom ViP layer in CUDA and integrate it into Caffe. The ViP layer inserts interpolated points between both columns and rows. The row and column size is doubled after interpolation and the resultant image size is enlarged four times. Interpolation is performed independently on points. This process is therefore embarrassingly parallel and can be easily accelerated by GPU. Each thread launched by the CUDA kernel processes one interpolated element. The thread block dimension order from fastest- to slowest-changing are column, row, channel, and batch to match the data layout in Caffe. Based on their position in the interpolated image, the points to be interpolated are classified into four types and estimated using Equation 5.5.

CNNs are now widely deployed and used in both cloud services and mobile phones, therefore we experiment with both a high-end desktop machine and a mobile platform with low power and energy profile. The detailed configurations are shown in Table 5.1. The desktop computer is equipped with high-end Intel Core-i7 CPU and Nvidia Titan X GPU, while the mobile platform is the Jetson TX1 comprised of efficient Quad-core ARM A57 CPU and Nvidia GPU with Maxwell architecture and 256 CUDA cores.

### 5.3.2 Image classification

We first apply the ViP method to speedup and reduce the energy consumption of the image classification task.

#### Accuracy and speed

We experiment with state-of-the-art VGG-16 and ResNet-50 models using the ImageNet dataset. We first apply the ViP method on VGG-16 as described in Algorithm 3 in Section 5.2. We conduct sensitivity analysis to determine the per-layer sensitivity as shown in Figure 5.3. The $x$-axis labels provide the names of the layers being interpolated and we explicitly append "pool" in the name of the layers that are immediately preceding a pooling layer. As we can see, (1) after ViP insertion, different accuracy degradations without fine-tuning are obtained (shown on $y$-axis in Figure 5.3), (2) all layers immediately preceding a pooling layer exhibit the least sensitivity to ViP operation, which was also discovered by [44]. The reason for this is that, although ViP loses information due to interpolation, many of those interpolated values are discarded by the pooling layer, and as a result, ViP has less impact on the final output of the network. And (3) besides the pooling layers, we can see a general trend of decreasing sensitivity when we insert ViP in later-stage layers. This follows the intuition that early perturbations lead to high error on the output when propagating through multiple layers, which is mathematically shown in Equation 5.6.

The next step is to do model fine-tuning with progressively inserted ViP layers. We use grouped fine-tuning in the case of VGG-16 to save training time. Specifically, we have four rounds of fine-tuning according to the sensitivity of the layers: (1) in round one, we insert ViP after convolution layers 13, 12, 10, 7, 2 and 4; (2) in round two, we further insert ViP after convolution layers 11, 9 and 8; (3) in round three, we further insert ViP after convolution layer 1; (4) in the final round four, we insert ViP layers after the remaining convolution layers. Each round is initialized with the trained model from

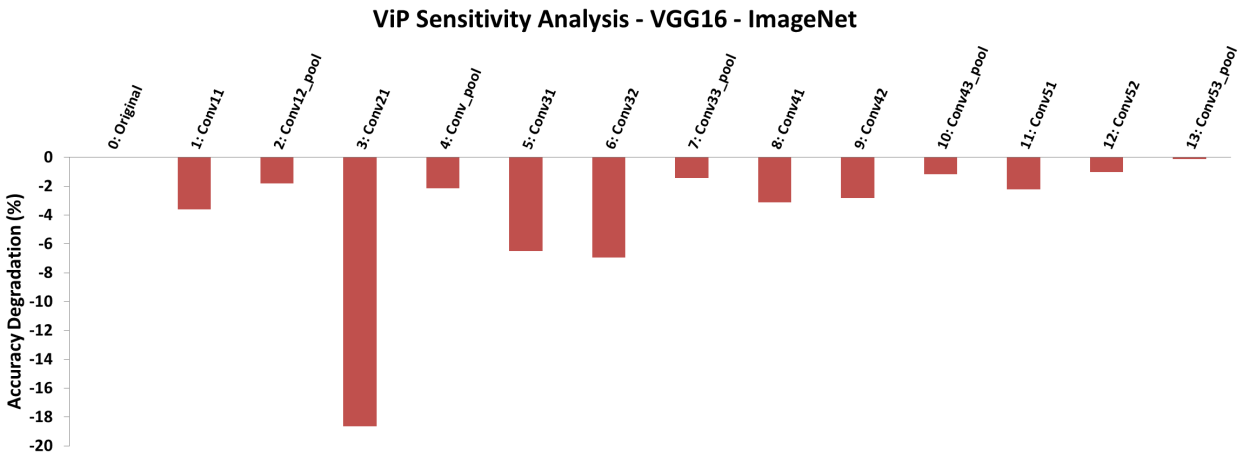



Figure 5.3: ViP sensitivity analysis of VGG-16 model under ImageNet dataset. For each of the convolution layers, we insert ViP immediately after it, and evaluate the network accuracy without fine-tuning. The sensitivity is measured as the accuracy drop with respect to the original accuracy.

the previous round, because such an approach (1) gives slightly higher accuracy than fine-tuning from the baseline model and (2) saves training time. The number of rounds is determined by the number of convolution layers one wants to accelerate and the granularity of the speedup-accuracy trade-off the application may require. It is possible to have as many fine-tuning rounds as the number of convolution layers, if the machine learning practitioner is looking for a model with very specific speedup-accuracy trade-off due to the nature of the target application.

Furthermore, we plot the training curve to illustrate how test accuracy recovers during grouped fine-tuning across four rounds, as shown in Figure 5.4. The zero line indicates the accuracy of the baseline network, and the $y$-axis is the accuracy improvement (degradation if negative) during fine-tuning. For fair comparison, we use top-5 accuracy for ImageNet throughout the chapter as also reported in [44]. The $x$-axis is the number of training iterations. We can see that after the initial insertion of ViP layers, there is a huge drop in accuracy. However, this gradually recovers during the fine-tuning step and even surpasses the original accuracy in round one. We conjecture that this is similar to the effect observed in [59], where linear interpolation serves as a type of regularization that improves network generalization.

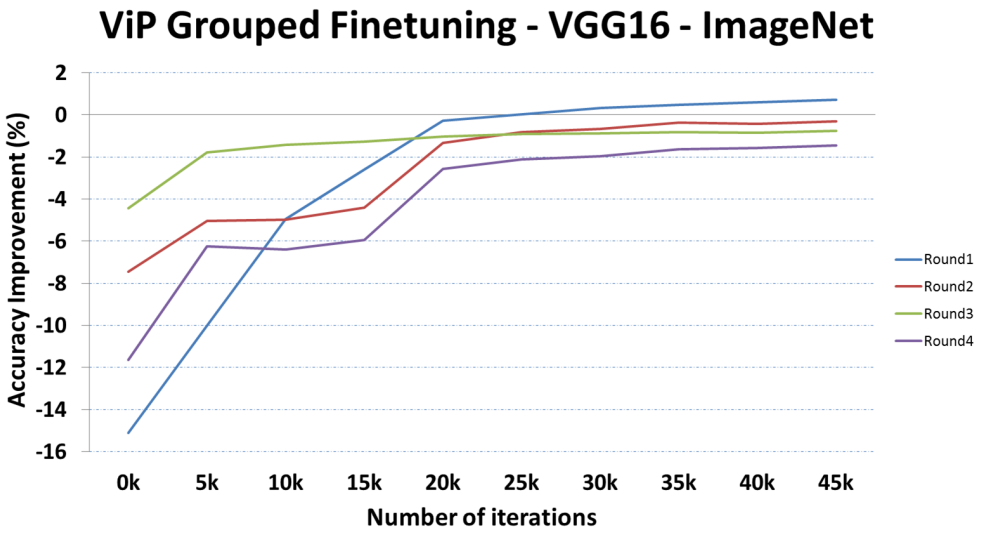



Figure 5.4: Four rounds of grouped finetuning of VGG-16 network using ImageNet dataset.

After four rounds of grouped fine-tuning, we obtain four models of different speedup-accuracy trade-offs. A positive value for accuracy change means improvement, while a negative value means accuracy drop. Speedup is measured as the ratio of the inference time of the original model over the inference time of the model with ViP. We fine-tune the model with ViP on the desktop machine, because (1) storage of the mobile platform is insufficient for holding the entire ImageNet dataset, (2) training on desktop machine is significantly faster and the trained model can be evaluated on both desktop and mobile platforms for runtime analysis, and (3) model accuracy is platform-independent, which means once a model is obtained, its test accuracy remains the same on any platform. Accordingly, we can report accuracy and speedup on both desktop and mobile platforms, while we only train the model on the desktop machine once.

We plot the results in Figure 5.5 along with the result of the previous state-of-the-art PerforatedCNNs [44]. Our method can achieve **2.1x** speedup with less than 1.5% accuracy degradation, while PerforatedCNNs can theoretically achieve 1.9x speedup with 2.5% accuracy degradation. The measured speedup of PerforatedCNNs is 2x when considering the reduced memory cost through implicit interpolation under Matlab im-

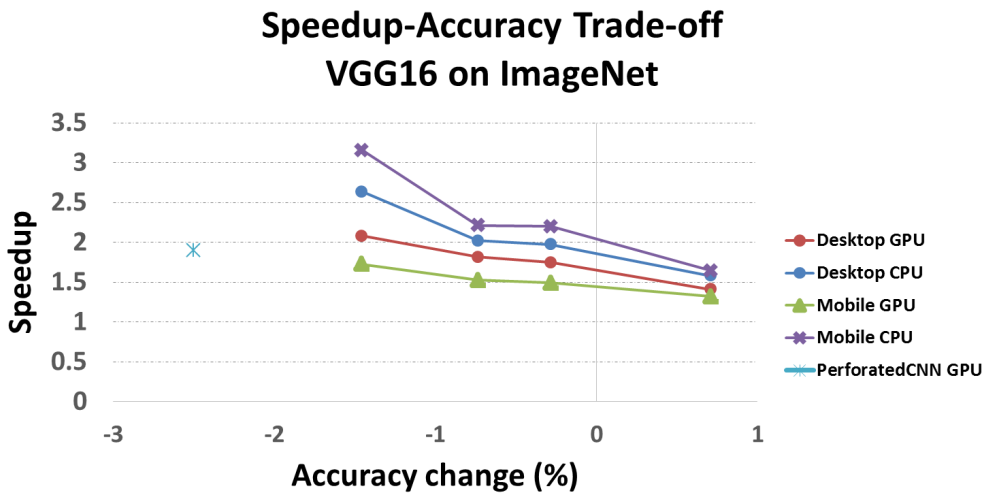



Figure 5.5: Speedup-Accuracy trade-off obtained by applying ViP on VGG-16 model with ImageNet dataset.

plementation [44]. In the same way, ViP can also reduce memory transfer cost between layers thanks to the smaller-sized intermediate outputs by using larger-stride convolution. Unfortunately, Caffe does not support implicit interpolation and hence no memory saving of intermediate outputs as pointed out by PerforatedCNNs [60]. Therefore, for fair comparison, we eliminate the effect of memory saving in both implementations and use the theoretical upper-limit for PerforatedCNNs speedup since they did not report speedup on Caffe implementation. We expect ViP method to achieve even higher speedup in implementations that support implicit interpolation which saves memory transfer cost. In the case of mobile CPU, ViP is able to speed up the CNN by **3.16x** with less than 1.5% accuracy drop. Besides, what ViP can obtain is a set of models with different speedup-accuracy trade-offs rather than a single configuration, CNN practitioners can pick any of the models in Figure 5.5 that meets their need.

Similarly, we apply ViP on ResNet-50 under ImageNet dataset. Figure 5.6 shows the results on sensitivity analysis and again we see the trend of decreasing sensitivity in later-stage layers. We have in total 53 convolution layers because there are 49 convolution layers on the primary branch and four on the bypass branches. Initially, we apply three rounds of grouped fine-tuning on ResNet-50. However, the final round, consist-

ing of layers with the highest sensitivity, results in a steep accuracy drop, from $-0.7\%$ to $-3.94\%$, we decide to use per-layer fine-tuning for the 12 layers in the last round to demonstrate the fine-grained progressive change in both accuracy and speed. Figure 5.7 shows the results. As expected, there is a clear trend of increasing speedup with higher accuracy drop when we insert more ViP layers. The speedup of mobile GPU and desktop GPU almost overlaps, and they both achieve **1.53x** speedup with less than 4% accuracy degradation. Meanwhile, mobile CPU obtains **2.3x** speedup at the same level of accuracy.

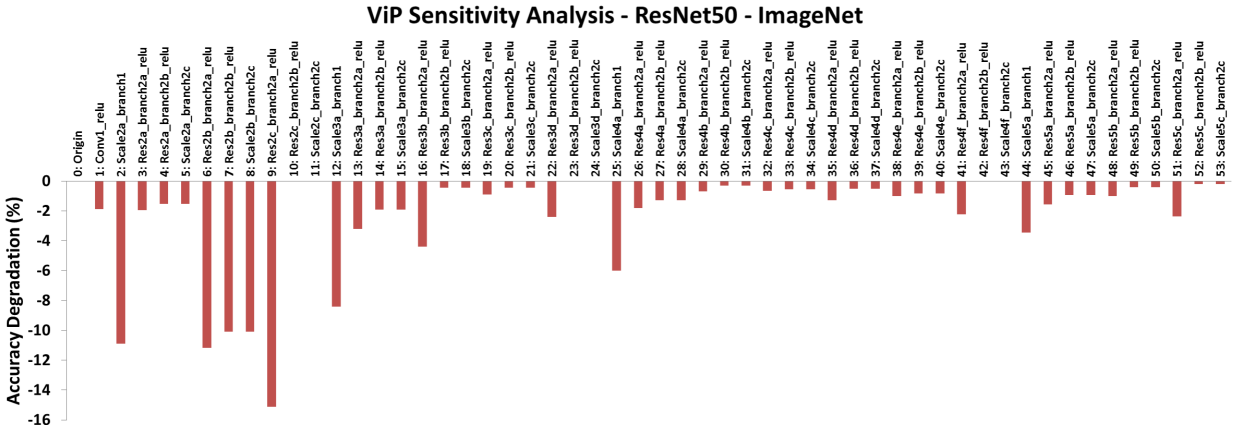



Figure 5.6: ViP sensitivity analysis of ResNet-50 model under ImageNet dataset. For each of the convolution layers, we insert ViP immediately after it, and evaluate the network accuracy without fine-tuning. The sensitivity is measured as the accuracy drop with respect to the original accuracy.

We also tried out our ViP method on the All-CNN network (Network All-CNN-C from [53]) which consists of nine convolution layers without a fully-connected layer and delivers high accuracy on CIFAR-10 dataset. With three rounds of ViP insertion and fine-tuning, we obtain three models with different speedup-accuracy trade-offs, as illustrated in Figure 5.8. We are able to achieve **1.77x** speedup on the Titan X GPU and up to **3.03x** speedup on the mobile CPU, with desktop CPU and mobile GPU in between, while the top-1 accuracy drop is within 4%. Also, with less than 1% accuracy degradation, we obtain a 1.36x speedup on Titan X GPU and 1.54x speedup on mobile CPU.

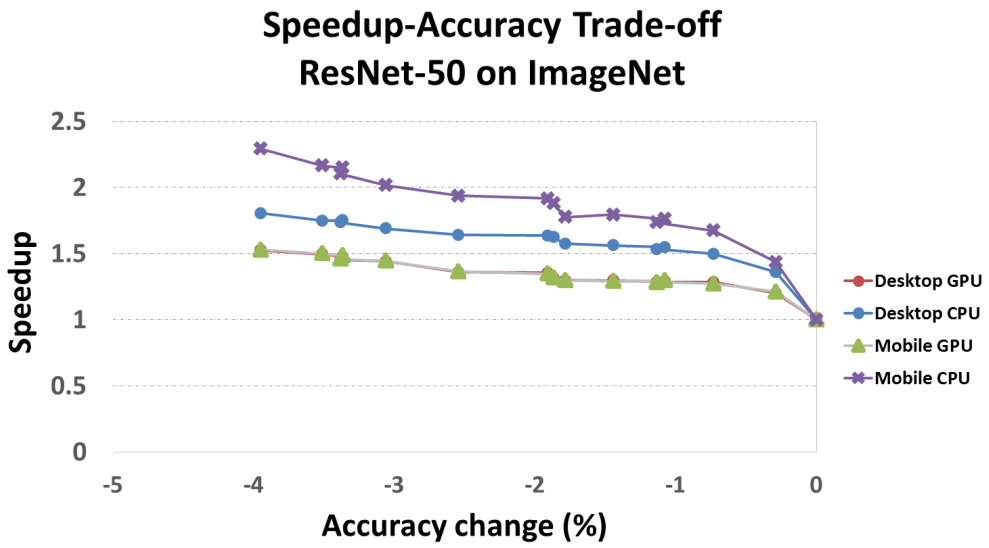



Figure 5.7: Speedup-Accuracy trade-off obtained by applying ViP on ResNet-50 model with ImageNet dataset.

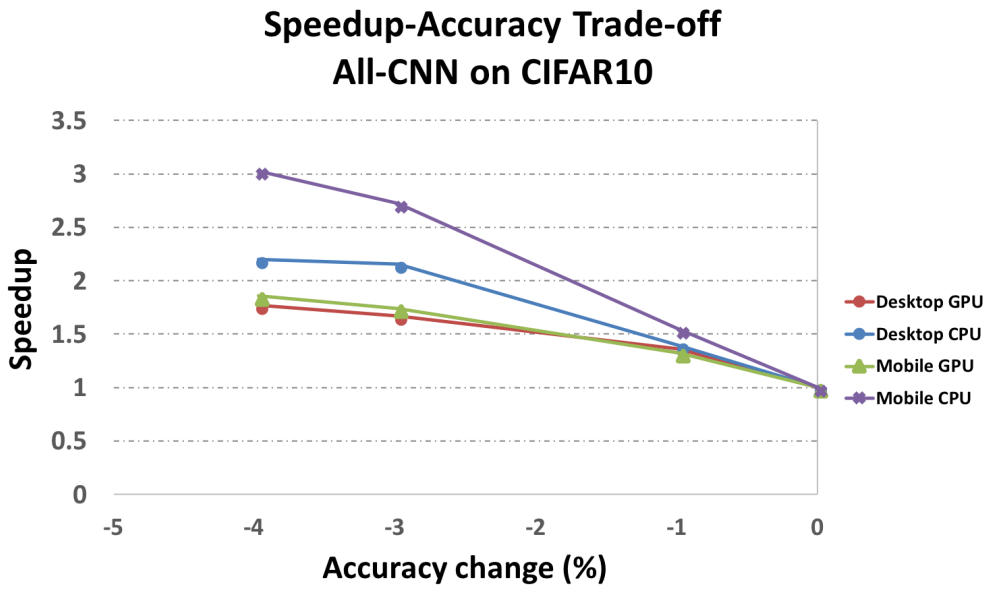



Figure 5.8: Speedup-Accuracy trade-off obtained by applying ViP on All-CNN model with CIFAR-10 dataset.

### Power and energy

Mobile smart phones have dramatically changed human lives in the recent decade, and with the advent of convolutional neural networks, more and more mobile apps start to integrate visual tasks like image classification and object detection that heavily rely on

CNNs. Other than speed, power and energy are the most critical constraints on mobile platforms. Therefore, we further conduct experiments to see how ViP improves the power and energy profile on the mobile platform running CNN.

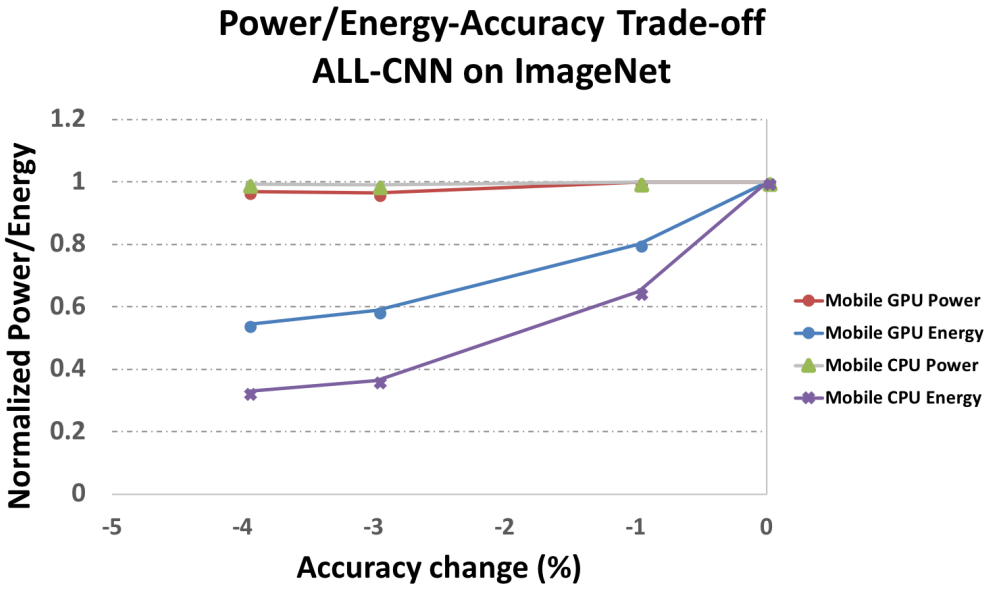



Figure 5.9: Powe/Energy-Accuracy trade-off obtained by applying ViP on All-CNN model with CIFAR-10 dataset.

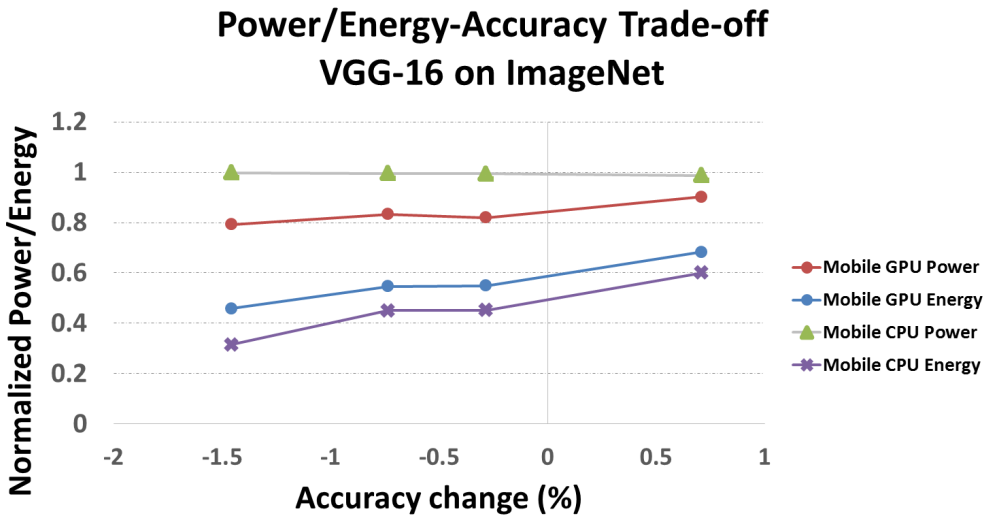



Figure 5.10: Powe/Energy-Accuracy trade-off obtained by applying ViP on VGG-16 model with ImageNet dataset.

We first port both Caffe and our custom ViP layer to Jetson TX1. We use the on-board sensor to measure the power consumption of CNNs with and without ViP technique,

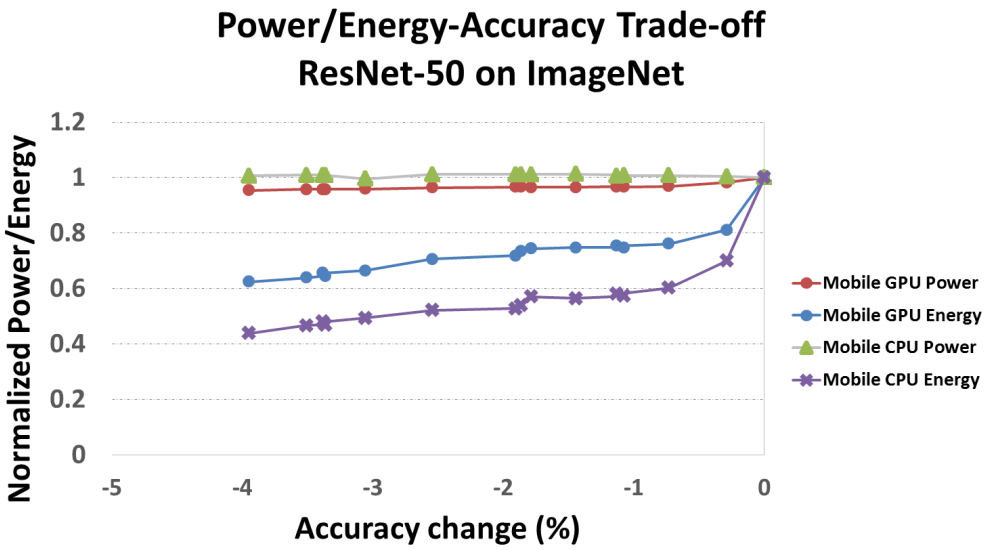



Figure 5.11: Powe/Energy-Accuracy trade-off obtained by applying ViP on ResNet-50 model with ImageNet dataset.

and obtain the energy consumption by multiplying power by CNN latency. We test on all CNNs used previously, *i.e.*, ALL-CNN, VGG-16 and ResNet-50, and report their power/energy-accuracy trade-off curves in Figures 5.9, 5.10 and 5.11, respectively. In each of the figures, we show four curves for power and energy consumption of either running CNN on mobile CPU or mobile GPU. We can see that, in terms of power, ALL-CNN has almost the same power on both CPU and GPU across all different ViP configurations, ResNet-50 has slightly lower power consumption on mobile GPU when using ViP and VGG-16 shows the highest power reduction of **21%** in mobile GPU power with ViP layers. In terms of energy consumption, VGG-16, ALL-CNN and ResNet-50 achieve up to **55%**, **46%** and **38%** mobile GPU energy reduction, respectively. Furthermore, All-CNN and VGG-16 can achieve up to **70%** CPU energy reduction while ResNet-50 tops at around **60%**.

### 5.3.3 Object detection

In real-world applications, people may more often seek the functionality of object detection than image classification. However, much of the prior work on CNN model

acceleration and compression only shows results on image classification [49, 50, 5]. Although image classification and object detection tasks share some common features like the early-stage convolutional layers, object detection has its unique components and challenges, *e.g.*, region proposal, bounding box regression, *etc*. Therefore, without experimental results, it is hardly convincing to infer that methods excel on image classification can also work well on object detection tasks. Accordingly, in this section, we further test the ViP method on object detection task to show that it works across both important tasks.

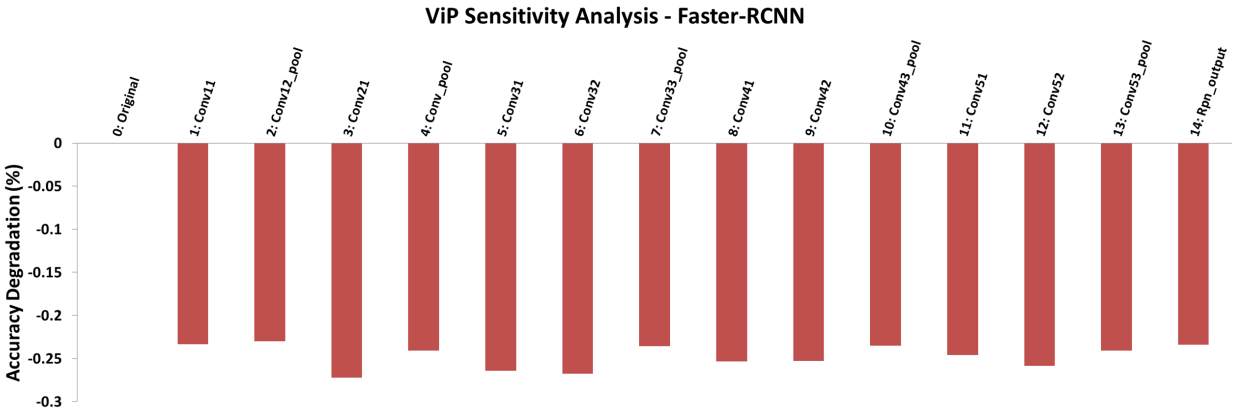



Figure 5.12: ViP sensitivity analysis of faster-rcnn with VGG-16 backbone under PASCAL VOC 2007 dataset. For each of the convolution layers, we insert ViP immediately after it, and evaluate the network accuracy without fine-tuning. The sensitivity is measured as the accuracy drop with respect to the original accuracy.

We use the Caffe implementation of the state-of-the-art object detection framework faster-rcnn [54] with PASCAL VOC 2007 dataset, and integrate it with our custom ViP layer. We use VGG-16 as the backbone network of faster-rcnn as in [54]. Similar to image classification task, we first analyze the sensitivity of each layers to ViP and the results are shown in Figure 5.12. Notice that, other than the convolution layers from VGG-16, we also have one layer from the region proposal network, and this layer turns out to be among the least sensitive layers that we need to insert ViP in early fine-tuning rounds. The mAP degradation from per-layer ViP ranges from $-0.23 \sim -0.27$, however, the recovered mAP degradation after fine-tuning is only down to $-0.024$. Besides, we

again observe that layers immediately followed by pooling are the most robust to ViP operation, as already discussed in the section of image classication. In the order of per-layer sensitivity, we conduct four rounds of grouped fine-tuning. As expected, with more layers followed by ViP operation, we are able to achieve higher speedup but with higher mAP degradation. In the end, we apply ViP to all convolution layers and achieve **1.8x** speedup with 0.025 mAP degradation.

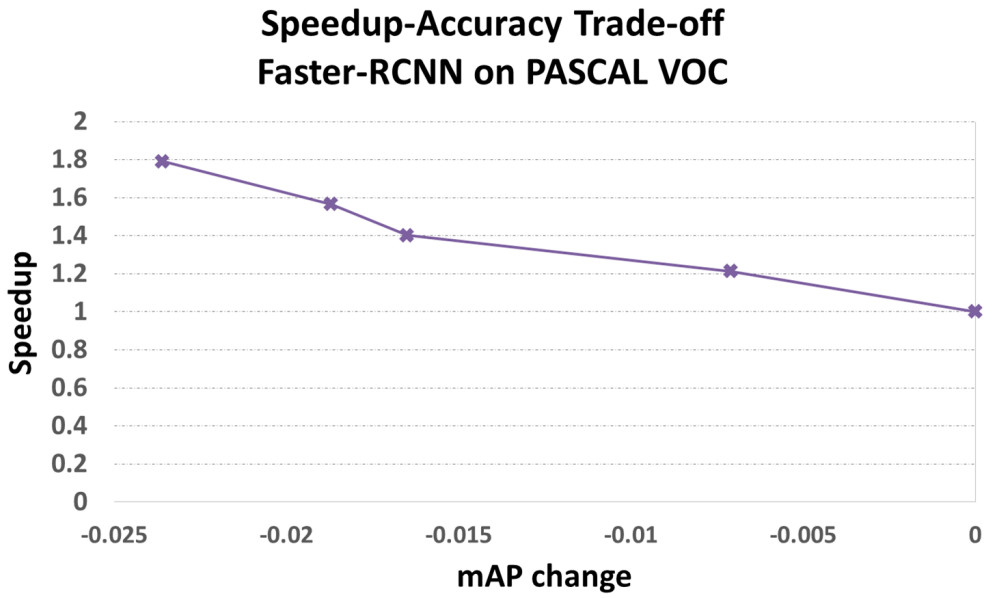



Figure 5.13: Speedup-Accuracy trade-off obtained by applying ViP on faster-rcnn with VGG-16 backbone under PASCAL VOC 2007 dataset.

## 5.4 Discussion

In this chapter, we propose the Virtual Pooling(ViP) method that exploits the spatial redundancy of the input and reduces CNN computation by using a larger stride convolution and then recovering the output with linear interpolation (which is very efficient). We test and validate our method extensively on four CNN models, three representative datasets, both desktop and mobile platforms, and two primary learning tasks, *i.e.*, image classification and object detection. When running on Nvidia Titan X GPU, ViP is able to speedup VGG-16 by **2.1x** with less than 1.5% accuracy degradation in the image classifi-

cation task, and speedup faster-rcnn by **1.8x** with 0.025 mAP degradation. Furthermore, we show that ViP is able to generate a set of models with different speedup-accuracy trade-offs. This provides CNN practitioners a tool for finding the model best suiting their needs.

# Chapter 6

# Convolutional neural networks: efficient training

## 6.1 Chapter overview

We have witnessed tremendous improvement in image classification tasks in recent years thanks to the use of supervised learning combined with the powerful model of Convolutional Neural Networks (CNNs) [61]. At the same time, the use of large-scale labeled datasets is one of the key elements that has led to this breakthrough [62] [63]. However, the definition of a label varies from application to application, and there is hardly a universal definition of what a “correct" label is for an image. One such example is how detailed the label should be, *i.e.*, label granularity (or label hierarchy), as illustrated in Figure 6.1. For example, in the case of animal image classification, it may be sufficient to label all images of carnivores as “carnivore", while in an application of carnivore classification, we may label different images as “dog", “cat", *etc.*, which are fine-grain labels of the coarse-grain label “carnivore". Therefore, it is equally correct to label the image of a dog as “carnivore" or “dog", yet deciding on which label of the two should be used depends on the task. We denote a fine-grain (coarse-grain) class as a class of images that are labeled with the respective fine-grain (coarse-grain) label, and fine-grain

(coarse-grain) training as the training process of CNNs using fine-grain (coarse-grain) labels. If the task at hand is classifying coarse-grain classes, *e.g.*, "carnivore" *vs.* "herbivore", the following question arises: should we directly train and test a CNN using coarse-grain labels as it has usually been done, or would it be beneficial if we trained a CNN with fine-grain labels, *e.g.*, "dog", "cat", "horse", "deer", *etc.* and map them back to coarse-grain labels during testing phase? The first approach is a method commonly used in image classification tasks [64], however, in our experiments, we find that training CNNs with fine-grain labels can achieve higher accuracy than using coarse-grain labels in most of the datasets considered. Table 6.1 shows both training and testing accuracy of coarse-grain classification using either coarse-grain or fine-grain labeling. We can see that fine-grain labeling helps improve both training accuracy (network optimization), and testing accuracy (network generalization) across representative image datasets: CIFAR-10 [63], CIFAR-100 [63], and ImageNet [62]. Moreover, helped by fine-grain labeling, the training process converges faster and requires less amount of training data to achieve the same level of testing accuracy, *i.e.*, becomes more data efficient. More specifically, for the CIFAR-10 dataset and two ImageNet subsets, a CNN trained with fine-grain labels and only 40% of the total training data can achieve even higher accuracy than a CNN trained with full training dataset but coarse-grain labels.

In this chapter, we design and conduct extensive experiments on various datasets to investigate this interesting phenomenon, and analyze and shed some light on how and why fine-grain label helps enhance coarse-grain image classification [65]. Our results show two potential practical use of this work: (i) when human resources are abundant, we can increase CNN accuracy by re-labeling the dataset with fine-grain labels and train the CNN using these new labels, and (ii) when human resources are limited and training data is hard to obtain, rather than relying on collecting more training data to improve CNN accuracy, we may instead re-label the dataset with fine-grain labels.

Table 6.1: Training and testing accuracy of five datasets when trained with fine-grain labeling (bottom row for each dataset) *vs.* coarse-grain labeling (top row for each dataset), and tested on coarse-grain labels.

| Dataset | # of training classes | # of testing classes | Training accuracy (%) | Testing accuracy (%) |
|---|---|---|---|---|
| CIFAR-10 | 2 | 2 | 99.9 | 98.42 |
| | 10 | 2 | 100.0 | 99.20 |
| CIFAR-100 | 20 | 20 | 100.0 | 85.04 |
| | 100 | 20 | 100.0 | 85.05 |
| CIFAR-100 animals | 10 | 10 | 100.0 | 81.42 |
| | 50 | 10 | 100.0 | 83.44 |
| ImageNet dog *vs.* cat | 2 | 2 | 94.1 | 92.68 |
| | 10 | 2 | 95.3 | 94.67 |
| ImageNet fruit *vs.* vege. | 2 | 2 | 91.8 | 89.65 |
| | 17 | 2 | 95.4 | 93.15 |

### 6.1.1 Chapter contributions

To the best of our knowledge, the work presented in this chapter makes the following contributions:

- This is the first work to analyze the use of finer-gain labeling for improving accuracy and training data efficiency for CNN-based image classification tasks.
- We show that through fine-grain labels, we can almost always improve CNN classification accuracy without changing network architecture.
- By using fine-grain labels, we can improve training data efficiency by a large margin, *e.g.*, only 40% of the total training data is needed to achieve even higher testing accuracy.
- We design and conduct experiments to understand why fine-grain labels help in this scenario.

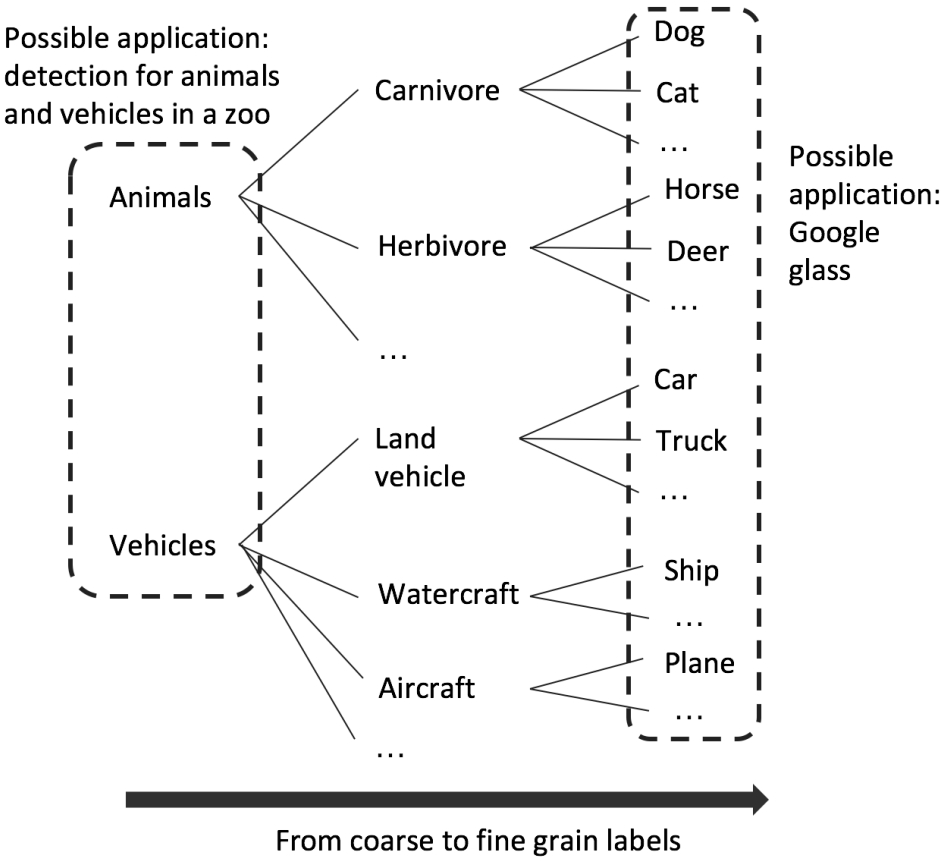



Figure 6.1: An example of label granularity (label hierarchy). For example, an image of a dog can be labeled “animal" or “carnivore" or “dog", and it is the target application that determines which label to use. This paper explores whether one should use the targeted coarse-grain labels or finer-grain labels for CNN training.

## 6.2 Label granularity and training data

In this section, we demonstrate the effects of fine-grain labels on improving image classification accuracy and further show its capability of enhancing training efficiency.

We define $A_{FC}^{train}$ and $A_{FC}^{test}$ as the training and testing accuracy of a CNN trained on fine-grain labels and evaluated on coarse-grain labels, respectively. In detail, we first train a network with fine-grain labels and output the predicted fine-grain labels of all input images. Then we map the predicted fine-grain labels to their respective coarse-grain labels via the predefined mapping as shown in table 6.2. This mapping follows the WordNet hierarchy [66] that is grouped by means of conceptual-semantic and lexical relations. Finally, the accuracy is computed by comparing the predicted coarse-grain

labels with the ground-truth labels. Similarly, we define $A_{CC}^{train}$ and $A_{CC}^{test}$ as the training and testing accuracy of a CNN trained on coarse-grain labels and evaluated on the same labels. To do this, we directly train a network with coarse-grain labels and compute accuracy by comparing the predicted labels, which are already coarse-grain labels, of the input images with their ground-truth labels.

We design and conduct experiments on well-known image classification datasets: CIFAR-10 [63], CIFAR-100 [63] and ImageNet [62], and we list their coarse- and fine-grain classes in Table 6.2. CIFAR-10 dataset is a great fit for applications similar to the one shown in Figure 6.1, *i.e.*, classifying whether an image contains an animal or a vehicle. CIFAR-10 has six animals: “bird", “cat", “deer", “dog", “frog", “horse", and four vehicles: “plane",“car", “ship", “truck". CIFAR-100 provides 20 coarse-grain classes and five fine-grain classes per coarse-grain class, resulting in 100 fine-grain classes in total. We also select all ten animal coarse-grain classes from CIFAR-100 to form another dataset serving applications like animal classification, and we call this dataset: CIFAR-100 animals. ImageNet dataset is collected and organized according to the WordNet hierarchy [62, 66] and therefore it naturally follows the coarse-to-fine-grain label hierarchy. We use subsets of ImageNet dataset to better visualize and demonstrate the benefits of training CNN with fine-grain labels. The first ImageNet subset task is to classify dog *vs.* cat, with a total of ten fine-grain classes of random breeds of dogs and cats. The second task is classifying fruit *vs.* vegetable with a total of 17 fine-grain classes.

Table 6.2: Coarse-grain and fine-grain classes of five datasets.

| Dataset | Coarse-grain classes | Fine-grain classes |
|---|---|---|
| CIFAR-10 | animal | bird, cat, deer, dog, frog, horse |
| | vehicle | plane, car, ship, truck |
| CIFAR-100 | aquatic mammals* | beaver, dolphin, otter, seal, whale |
| | fish* | aquarium fish, flatfish, ray, shark, trout |
| | flowers | orchid, poppy, rose, sunflower, tulip |
| | food containers | bottle, bowl, can, cup, plate |
| | fruit and vegetables | apple, mushroom, orange, pear, sweet pepper |
| | household electrical devices | clock, keyboard, lamp, telephone, television |
| | household furniture | bed, chair, couch, table, wardrobe |
| | insects* | bee, beetle, butterfly, caterpillar, cockroach |
| | large carnivores* | bear, leopard, lion, tiger, wolf |
| | large man-made outdoor things | bridge, castle, house, road, skyscraper |
| | large natural outdoor scenes | cloud, forest, mountain, plain, sea |
| | large omnivores and herbivores* | camel, cattle, chimpanzee, elephant, kangaroo |
| | medium mammals* | fox, porcupine, possum, raccoon, skunk |
| | non-insect invertebrates* | crab, lobster, snail, spider, worm |
| | people* | baby, boy, girl, man, woman |
| | reptiles* | crocodile, dinosaur, lizard, snake, turtle |
| | small mammals* | hamster, mouse, rabbit, shrew, squirrel |
| | trees | maple tree, oak tree, palm tree, pine tree, willow tree |
| | vehicles 1 | bicycle, bus, motorcycle, pickup truck, train |
| | vehicles 2 | lawn mower, rocket, streetcar, tank, tractor |
| CIFAR-100 animals | (10 coarse-grain classes above marked with *) | (50 corresponding fine-grain classes) |
| ImageNet dog *vs.* cat | dog | basset, chihuahua, maltese, papillon, pekinese, |
| | cat | tabby, tiger cat, Persian, Siamese, Egyptian |
| ImageNet fruit *vs.* vege | fruit | strawberry, orange, lemon, fig, pineapple, banana, jackfruit, custard apple |
| | vege | head cabbage, broccoli, cauliflower, zucchini, butternut squash, cucumber, artichoke, pepper, mushroom |

Table 6.3 shows the network configurations used for different datasets. For CIFAR-10, we use the full pre-activation residual network with 512 filters for the widest layer similar to the one in [67]. We use wide residual network [68] for CIFAR-100 dataset, which achieves 81.15% accuracy on 100 classes. We use "thinner" networks for ImageNet subsets to avoid overfitting because ImageNet subsets have fewer training images (22K images for fruit *vs.* vegetable, and 13K images for dog *vs.* cat) than CIFAR-10 (50K images) and CIFAR-100 (50K images). The network configuration for ImageNet subsets is similar to CIFAR-10, but with 75% fewer filters per convolution layer. We use random cropping and random flipping data augmentation [69] for all datasets. For the training configuration, we use momentum 0.9 and weight decay 5e-4. The learning rate starts at 0.1 for CIFAR-10 and CIFAR-100, and 0.01 for ImageNet subsets, and decays when the loss plateaus. We train CIFAR-10 and CIFAR-100 for 200 epochs, and ImageNet subsets for 225 epochs. If the CNN models are too small, we may run into higher risk of underfitting the fine-grain classes, as we have more classes with fewer samples in each.

Table 6.3: Configuration of CNNs used in the experiments.

| Dataset | # of layers | # of filters in widest layer | # of parameters |
|---|---|---|---|
| CIFAR-10 | 18 | 512 | 11.1M |
| CIFAR-100<br>CIFAR-100 animals | 26 | 640 | 36.5M |
| ImageNet subsets | 18 | 128 | 0.7M |

Table 6.1 shows the results. The second column gives the number of classes CNN is trained on and the third column shows the number of classes the CNN is tested on. If these two numbers are the same, it means training and testing are both using the same coarse-grain labels. If they are different, it means that CNN is trained first with fine-grain labels and then tested on the coarse-grain labels. We can see that training using fine-grain labels almost always improves testing accuracy compared to training using coarse-grain labels. In the case of CIFAR-100, fine-grain training provides negligible improvement on testing accuracy. We conjecture that this is due to the diminishing

return when more coarse-grain labels are, and we verify this hypothesis in Section 6.4.3. We further experiment with more than two levels of hierarchy. For example, in the case of ImageNet Fruit vs. Vegetable, we have two coarse-grain, four fine-grain and 17 finer-grain classes as described in Table 6.4. We obtain testing accuracy on coarse-grain classes of 89.65%, 91.53% and 93.15%, respectively. This shows that with deeper hierarchy, we expect higher testing accuracy from fine-grain training. However, under limited amount of training data, when we use extremely deep levels of the hierarchy, the number of samples of that level may not be sufficient for training, and hence lead to overfitting. We also observe that CNN training accuracy gets better when using fine-grain labels. Above results indicate that fine-grain labels help improve both network optimization and generalization, and we will analyze the reasons in Sections 6.3.1 and 6.3.2, respectively.

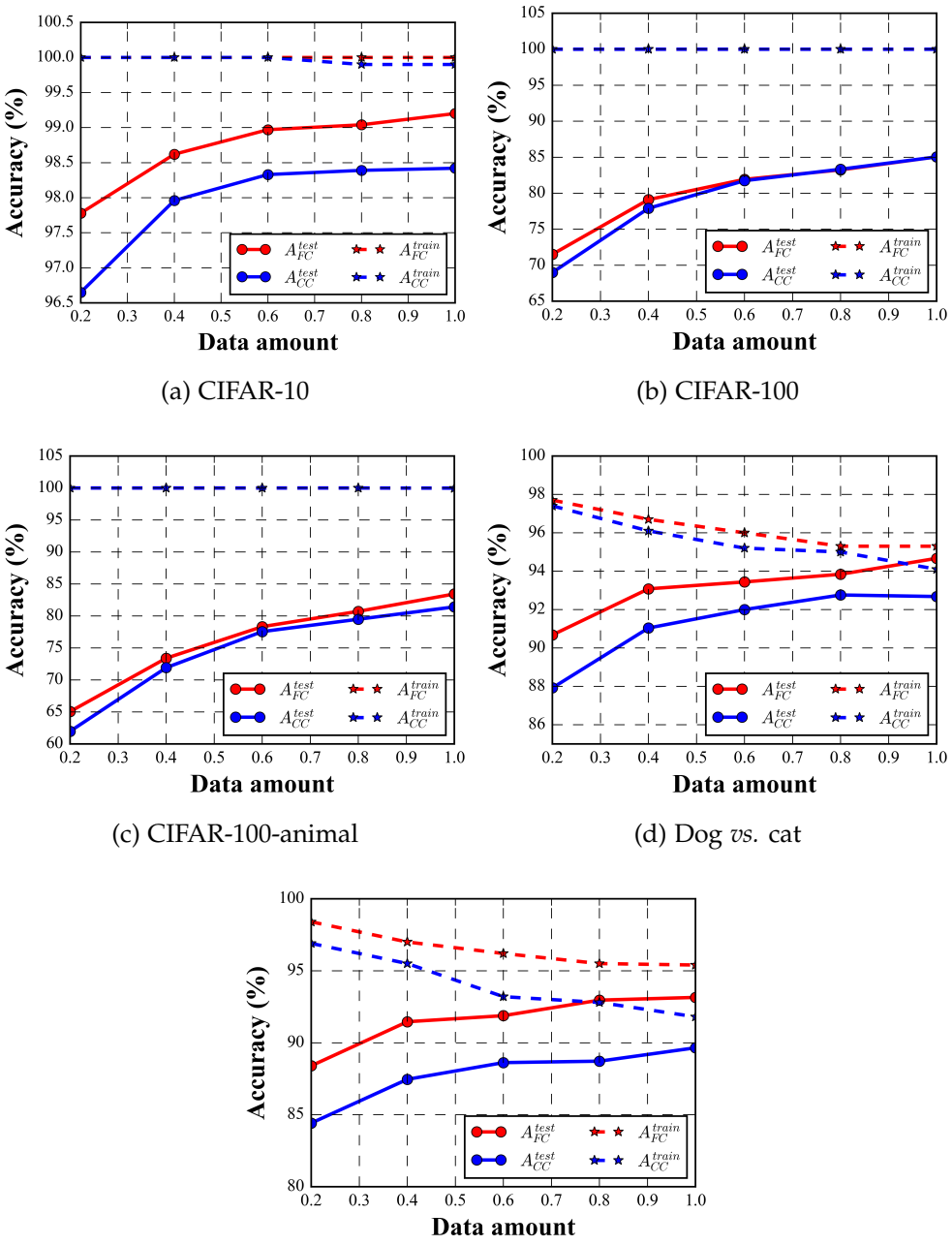
Table 6.4: Coarse-, fine- and finer-grain classes of ImageNet subset.

<table>
<tr><th></th><th>Coarse-grain classes</th><th>Fine-grain classes</th><th>Finer-grain classes</th></tr>
<tr><td rowspan="4">ImageNet fruit vs. vege</td><td rowspan="2">fruit</td><td>yellowish</td><td>orange, lemon, pineapple, banana, jackfruit</td></tr>
<tr><td>non-yellowish</td><td>strawberry, fig, custard apple</td></tr>
<tr><td rowspan="2">vege</td><td>round-shaped</td><td>head cabbage, broccoli, cauliflower,artichoke, pepper, mushroom</td></tr>
<tr><td>long-shaped</td><td>zucchini, butternut squash, cucumber</td></tr>
<tr><td>Testing Accuracy (%)</td><td>89.65</td><td>91.53</td><td>93.15</td></tr>
</table>

We further investigate how fine-grain labels affect training data efficiency. High training data efficiency means that (i) with the same amount of data, CNNs are able to learn and perform better, *i.e.*, achieve higher testing accuracy, and (ii) to achieve the same testing accuracy, CNNs require fewer training data.

To this end, we randomly chose 20%, 40%, 60% or 80% of the entire training dataset to form four new training sets with increasing data amount (same proportion in each class so that the number of images within each class is still balanced), use the full testing dataset for testing, and compare the accuracy of fine-grain and coarse-grain training. Since we find that keeping the same number of epochs for reduced data amounts leads to fewer weight updates, we use proportionally more training epochs for less training data to keep the number of weight updates the same. In other words, when 20% of training data is used, we train for 5X epochs.

The results are depicted in Fig 6.2, where we show training and testing accuracy for both fine-grain and coarse-grain training, *i.e.*, $A_{FC}^{train}$, $A_{FC}^{test}$, $A_{CC}^{train}$ and $A_{CC}^{test}$. We observe that training with fine-grain labels almost always improves testing accuracy. Especially in the case of (a), (d) and (e) of Figure 6.2 (CIFAR-10, ImageNet dog *vs.* cat, and ImageNet fruit *vs.* vegetable, respectively), we observe a significant improvement from the use of fine-grain labels: with less than 40% of the total training data, training with fine-grain labels is able to achieve even higher accuracy than using the full training dataset with coarse-grain labels. For CIFAR-100 animals (c), with only 80% of the total data amount, training with fine-grain labels is able to achieve comparable accuracy as using coarse-grain labels and full training dataset. Although fine-grain training has negligible improvement on testing accuracy with full dataset in case of CIFAR-100, when using fewer than 40% of the full dataset, fine-grain training still exhibits a clear advantage. This indicates that when availability of data is limited, having fine-grain labels can be helpful.




(a) CIFAR-10

(b) CIFAR-100

(c) CIFAR-100-animal

(d) Dog *vs.* cat

(e) Fruit *vs.* vegetable

Figure 6.2: Training (dotted) and testing (solid) accuracy curves with increasing amount of training data. CNNs trained with fine-grain labels are shown in red and those trained with coarse-grain labels are shown in blue. Experiments are conducted using five datasets: (a) CIFAR-10, (b) CIFAR-100, (c) CIFAR-100-animal, and two subsets of ImageNet datasets (d) dog *vs.* cat, (e) fruit *vs.* vegetable.

These experimental results show that training with fine-grain labels can help CNNs better utilize the available training data and can almost always improve CNN accuracy.

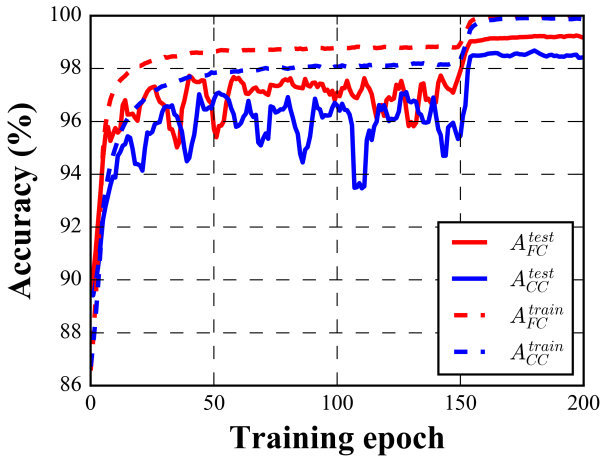
This indicates two potential practical usage of this work: i) if we have sufficient human resources, we can improve CNN performance by re-labeling data with fine-grain labels, and ii) if we have limited human resources, in order to improve CNN performance, it can be more advantageous to re-labeling images with fine-grain labels rather than collecting more data.

## 6.3 Optimization and generalization

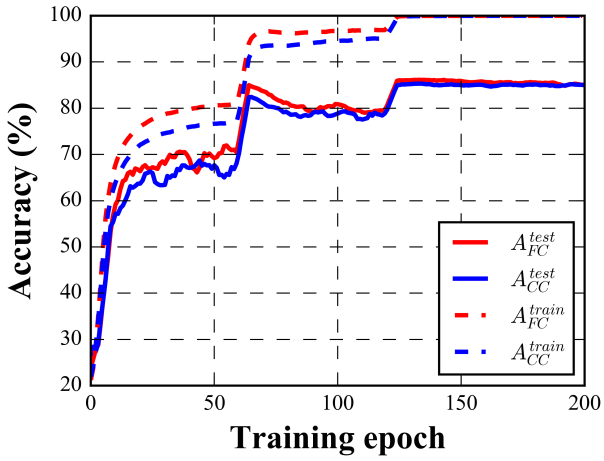
As we have discussed in Section 6.2, training with fine-grain labels can improve not only testing but also training accuracy. This means that fine-grain labels help with both network *optimization* and *generalization*. In this section, we design and conduct extensive experiments on all datasets showing how both optimization and generalization are improved.

### 6.3.1 Optimization

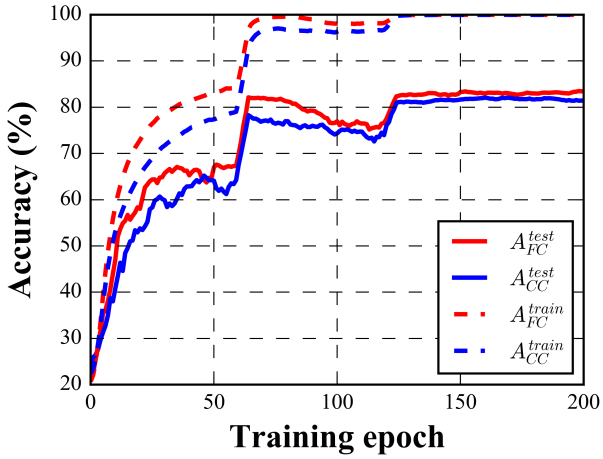
In Figure 6.3, the dotted curves show the training accuracy of both fine-grain training, $A_{FC}^{train}$, and coarse-grain training, $A_{CC}^{train}$, for all datasets. The training accuracy is evaluated at the end of every epoch during the training phase by using the training dataset. We can see that training with fine-grain labels not only achieves higher training accuracy, but also converges faster as the red curve is always above the blue curve. The accuracy jumps are the results of reduced learning rate and are common phenomena in training neural networks [69].



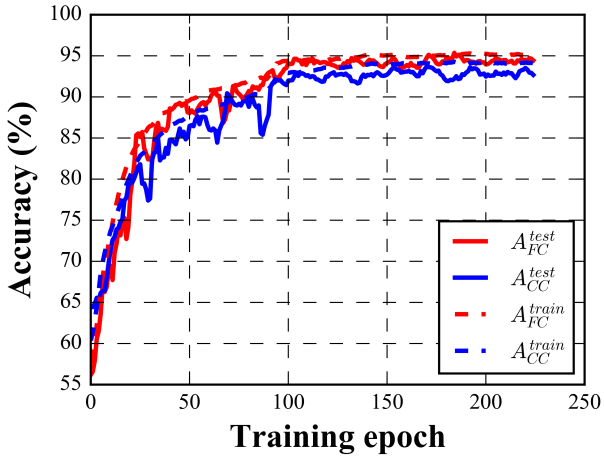

(a) CIFAR-10



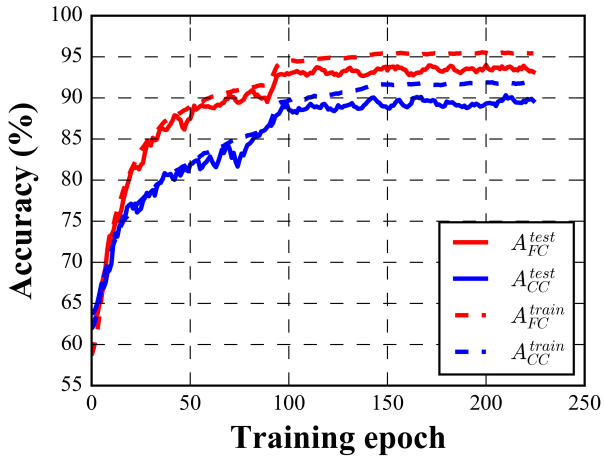

(b) CIFAR-100




(c) CIFAR-100 animals




(d) Dog *vs.* cat




(e) Fruit *vs.* vegetable

Figure 6.3: Training (dotted) and testing (solid) accuracy curves for five datasets. CNNs trained with fine-grain labels are shown in red and those trained with coarse-grain labels are shown in blue. Experiments are conducted using five datasets: (a) CIFAR-10, (b) CIFAR-100, (c) CIFAR-100 animals, and two subsets of ImageNet datasets (d) dog *vs.* cat and (e) fruit *vs.* vegetable.

Prior art investigating fine-grain labels on simple linear classifiers argues that the reason fine-grain labeling helps is the ability to learn piece-wise linear decision boundaries that can better approximate the true non-linear decision boundary [70]. That is, fine-grain training can have higher non-linearity compared to coarse-grain training due to increased parameters in the model. However, a further study [71] shows that in the case of non-linear classifiers, *e.g.*, RBF-kernel Support Vector Machine (SVM), fine-grain training no longer improves accuracy compared to using coarse-grain labels because the network itself has sufficient non-linearity to learn the non-linear decision boundary without the help of fine-grain labels. In the case of CNNs, we ask the question: is this piece-wise linear nature the reason for better training accuracy for fine-grain labels compared to coarse-grain training?

CNNs are already highly non-linear, so we conjecture that the answer is no. To evaluate this, we insert another fully-connected layer to a coarse-grain trained network right after the global pooling layer, so that compared to the original network, it can also achieve a piece-wise linear boundary on the high-level features. We train the new network end to end from scratch instead of pre-loading and freezing the weights of the preceeding layers, such that it fully utilizes all the degrees of freedoms of the model, possibly achieving higher training accuracy. We train this new network structure with coarse-grain labels, and compare the results with the baseline network trained with coarse- or fine-grain labels. We keep the training scheme for the slightly deeper network the same as the baseline network.

Table 6.5 shows our results. In the "CNN Arch" column, 'Extra layer' means that we add the fully-connected layer to the baseline CNN as described above. In the "Train Label" column, "F" and "C" indicate fine-grain and coarse-grain labels, respectively. The values in parentheses following each training and testing accuracy value are the improvement/degradation with respect to the training and testing accuracy of a baseline CNN trained with coarse-grain labels, respectively.

We can see that, compared to the baseline CNN trained with coarse-grain labels, adding one extra layer does not bring significant improvement in either optimization or generalization. In certain cases, the testing accuracy is degraded, in CIFAR-100 animals and ImageNet subset dog *vs.* cat, possibly due to the difficulty in optimizing a larger network.

This means that simply adding non-linearity to coarse-grain training cannot match the training accuracy brought by fine-grain training. That is, the slightly higher non-linearity brought by fine-grain training is not the only reason for achieving higher training accuracy. Rather, it is more likely that fine-grain labels give more hints to the network about which features to learn. This is also supported by the experimental results in Section 6.4.2, where we randomly generate fine-grain labels for each coarse-grain class, and find out that fine-grain training does not optimize better than coarse-grain training.

Table 6.5: Experiments on increasing CNN non-linearity and capacity under coarse-grain training. In "CNN Arch": 'Extra layer' means that we add the fully-connected layer to the baseline CNN to increase network non-linearity and capacity as described in Section 6.3.1. In "Train Label": "F" and "C" indicate fine-grain and coarse-grain labels, respectively. In the training and testing accuracy columns, the values indicated in the parentheses are the improvement/degradation with respect to the training and testing accuracy of a baseline CNN trained with coarse-grain labels, respectively.

| Dataset | CNN Arch | Train Label | Training accuracy (%) | Testing accuracy (%) |
|---|---|---|---|---|
| CIFAR-10 | Baseline CNN | F | 100.0 (+0.1) | 99.20 (+0.78) |
| | Extra layer | C | 99.9 (+0.0) | 98.50 (+0.08) |
| CIFAR-100 | Baseline CNN | F | 100.0 (+0.0) | 85.05 (+0.01) |
| | Extra layer | C | 100.0 (+0.0) | 86.33 (+1.29) |
| CIFAR-100 animals | Baseline CNN | F | 100.0 (+0.0) | 83.44 (+2.02) |
| | Extra layer | C | 100.0 (+0.0) | 80.73 (-0.69) |
| ImageNet dog *vs.* cat | Baseline CNN | F | 95.3 (+1.2) | 94.87 (+2.19) |
| | Extra layer | C | 93.8 (-0.3) | 92.2 (-0.48) |
| ImageNet fruit *vs.* vege | Baseline CNN | F | 95.4 (+3.6) | 93.15 (+3.5) |
| | Extra layer | C | 91.7 (-0.1) | 89.67 (+0.02) |

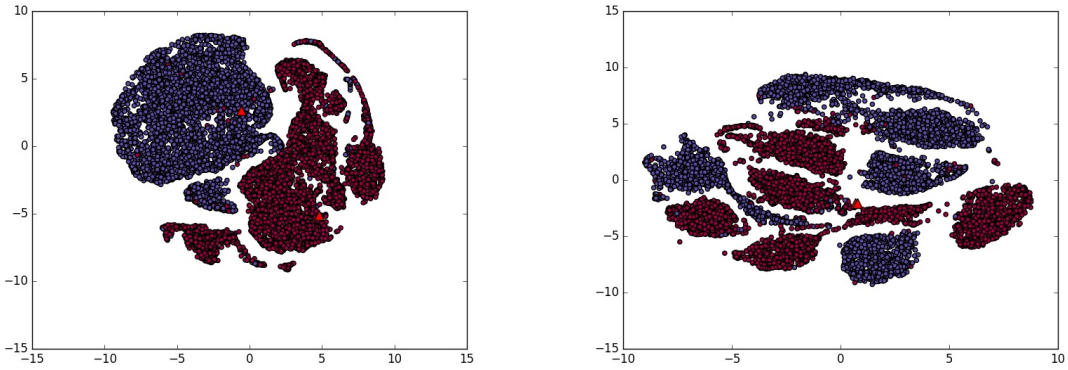
### 6.3.2 Generalization

As shown in both Table 6.1 and Figure 6.3, training with fine-grain labels (*vs.* coarse-grain) achieves higher testing accuracy. This may partially be due to better network optimization, because under ImageNet subsets, fine-grain training improves both training and testing accuracy. However, in the cases of CIFAR-10, CIFAR-100 animals and CIFAR-100, even for the same training accuracy, testing accuracy for fine-grain trained CNNs is still higher than coarse-grain training. This indicates that fine-grain training delivers higher generalization capability.

Our intuition is that with fine-grain labels, the CNN is able to learn more features than training with coarse-grain labels. For example, suppose that all cat images in the training set have whiskers, while none of the dogs has whiskers. Then, as long as the network trained with coarse-grain labels learns this feature, it can produce 100% training accuracy with no need to learn any other features. This is a well known phenomenon in weakly-supervised learning, in which the network only learns the most discriminative features [72]. Then, in the testing set, if a cat image does not include whiskers, the network will make an incorrect prediction. However, with fine-grain labels, the network needs to learn more features (*e.g.*, ears, tails, *etc.*) to distinguish among different breeds of dogs and cats. These extra features learned through fine-grain labeling may help the network's performance on coarse-grain class classification on the testing set, *e.g.*, it now can tell if it is a cat through ears, tails, *etc*, even though it does not have whiskers.

Figure 6.4 shows the t-distributed Stochastic Neighbor Embedding (t-SNE) visualization [73] of all CIFAR-10 testing images with coarse-grain (a) and fine-grain training (b).

Image features used for t-SNE visualization are the outputs of the second-to-last fully-connected layer, which is a technique commonly used to extract compact semantic representation of the raw input images [73]. These feature vectors are then transformed by the t-SNE technique [73] to a two-dimensional space for visualization. All data points are



(a) t-SNE visualization of two coarse-grain classes in CIFAR-10 trained with two coarse-grain labels

(b) t-SNE visualization of two coarse-grain classes in CIFAR-10 trained with ten fine-grain labels

Figure 6.4: t-SNE visualization of CIFAR-10 test set trained with coarse-grain labels *vs.* fine-grain labels. Data points shown in the same color belong to the same coarse-grain class.

colored according to their ground-truth coarse-grain labels. We also show the position of means of each coarse-grain class as red triangles in the figures. We can see that, for both coarse-grain training, Figure 6.4a, and fine-grain training, Figure 6.4b, there is a noticeable margin between coarse-grain classes, and a decision boundary can be drawn to separate them. However, the network has to learn extra features to further separate the fine-grain classes within each coarse-grain class when trained with fine-grain labels (as shown in Figure 6.4b), while when trained with only coarse-grain labels, the data points are merged together as there is no need to separate them (as being visualized in Figure 6.4a). This also explains why a similar phenomenon is not present in non-linear shallow classifiers, *e.g.,* kernel SVM and kernel logistic regression [70]: traditional classifiers use human-defined features that are always fixed, while CNNs are able to learn these features during training [74] (feature points moved in Figure 6.4a and Figure 6.4b). The learning of features turns out to be critical in improving the accuracy under fine-grain training as described above.

An orthogonal method used for enhancing the variety of learned features and thereby increasing generalization ability is dropout [75]. Dropout randomly drops some of the

features to encourage CNN to learn more various features. A possible question arises: by adding dropout to the network, will coarse-grain training reach the same testing accuracy as fine-grain training?

In cases of CIFAR-100 and CIFAR-100 animals, the original network already has dropout layers within each residual block with the optimal dropout rate 0.3 determined by experiments [68]. Therefore, Table 6.1 has already shown that adding dropout in these two datasets is still unable to reach the accuracy trained with fine-grain labels. We further conduct experiments on CIFAR-10 and ImageNet subsets, by adding a dropout layer between the global pooling layer and the fully-connected layer. The dropout rate is set as 0.3 in our following experiments.

Table 6.6 shows the experimental results of using the dropout technique. We observe that adding the dropout layer provides limited improvement in testing accuracy for coarse-grain training, and dropout for coarse-grain training still generates a noticeable margin when compared to the fine-grain training with or without dropout. This indicates that fine-grain labels can further improve CNN learning beyond what the traditional dropout technique can do. Actually, since fine-grain training and dropout are two orthogonal techniques, one can use both to further improve CNN performance. For example, in ImageNet subsets dog *vs.*cat and fruit *vs.* vegetable, combining the two techniques can push the testing accuracy to 95% and 93.86% from 92.68% and 89.65%, respectively.

## 6.4 Exploration

In this section, we further explore several scenarios in which the setting of coarse-grain and fine-grain labels change. More specifically, coarse-grain classes may vary due to the requirement of the application and the fine-grain labels may be noisy if they are generated via automatic unsupervised clustering algorithms. We also investigate how

Table 6.6: Experiments on increasing CNN dropout rate. Values in “Dropout" column indicates dropout rates used. In “Train Label" column: “F" and “C" indicate fine-grain and coarse-grain labels, respectively.

| Dataset | Dropout | Train Label | Training accuracy (%) | Testing accuracy (%) |
|---|---|---|---|---|
| CIFAR-10 | 0.3 | F | 100.0 | 99.10 |
| | 0.3 | C | 99.9 | 98.48 |
| | 0 | F | 100.0 | 99.20 |
| | 0 | C | 99.9 | 98.42 |
| ImageNet dog *vs.* cat | 0.3 | F | 94.8 | 95.00 |
| | 0.3 | C | 94.3 | 92.80 |
| | 0 | F | 95.3 | 94.87 |
| | 0 | C | 94.1 | 92.68 |
| ImageNet fruit *vs.* vege | 0.3 | F | 95.0 | 93.86 |
| | 0.3 | C | 91.7 | 89.93 |
| | 0 | F | 95.4 | 93.15 |
| | 0 | C | 91.8 | 89.65 |

increasing the number of coarse-grain classes impacts the improvement from using fine-grain labels, *i.e.*, $\Delta A^{test}$.

### 6.4.1 Customized coarse-grain classes

As mentioned, coarse-grain classes are the classification target, and as a result, the definition of coarse-grain classes is application dependent. For example, given an animal dataset, a task can be identifying cat *vs.* dog. vs horse, while another task can be separating standing animals from sitting and/or lying animals. Because of the diversity of applications, this mapping from fine-grain classes to coarse-grain classes can be drastically different. In this section, we conduct experiments to see how these customized coarse-grain classes affect the effectiveness of fine-grain labels.

A natural partition of CIFAR-10 dataset is the “animal" coarse-grain class *vs.* the “vehicle" coarse-grain class, where “animal" has six fine-grain classes and “vehicle" has four as depicted in Table 6.2. To simulate various applications, we keep the 6:4 ratio of the two coarse-grain classes and randomly switch their fine-grain classes to create

new coarse-grain classes. Rows (1) through (5) in Table 6.7 show five experiments with different coarse-grain class definitions. We use two coarse-grain classes in this case (denoted by 0 and 1), and values in the table indicate which coarse-grain class (0 *vs.* 1) each fine-grain class (plane, car, *etc*.) belongs to. The last three columns of Table 6.7 give the testing accuracy of the CNN trained with coarse-grain and fine-grain labels, respectively as well as the relative improvement of fine-grain training. We observe that fine-grain training achieves up to 2.78% improvement and always outperforms coarse-grain training under various customized coarse-grain classes.

We further experiment with balanced coarse-grain classes. In the previous experiments, we have a 6:4 ratio for the number of fine-grain classes within each coarse-grain class. Now, we balance it to a 5:5 ratio, and similarly, we randomly switch fine-grain classes across the two coarse-grain classes. Rows (6) through (10) in Table 6.7 show five experiments with different coarse-grain class definitions and a 5:5 ratio. Again, we can see that fine-grain training always produces higher testing accuracy than coarse-grain training.

Table 6.7: Testing accuracy, trained with coarse-grain *vs.* fine-grain labels, of customized coarse-grain classes of CIFAR-10 dataset. Zero and one indicates which coarse-grain class each fine-grain class belongs to.

| ID | Ratio | Classes | | | | | | | | | | $A_{CC}^{test}$ (%) | $A_{FC}^{test}$ (%) | $\Delta A^{test}$ (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | plane | car | bird | cat | deer | dog | frog | horse | ship | truck | | | |
| (1) | 6:4 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 98.42 | 99.20 | +0.78 |
| (2) | | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 97.68 | 98.64 | +0.96 |
| (3) | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 96.95 | 98.02 | +1.07 |
| (4) | | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 95.26 | 97.20 | +1.94 |
| (5) | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 93.44 | 96.22 | +2.78 |
| (6) | 5:5 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 97.60 | 98.51 | +0.91 |
| (7) | | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 95.90 | 97.59 | +1.69 |
| (8) | | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 96.17 | 97.54 | +1.37 |
| (9) | | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 94.15 | 96.28 | +2.13 |
| (10) | | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 94.19 | 96.16 | +1.97 |

### 6.4.2 Noisy fine-grain classes

By using fine-grain labels, we are able to improve CNN performance. To obtain fine-grain labels, we can either have humans label the images, or automatically cluster every

coarse-grain class into multiple fine-grain classes. The first approach is human-labor intensive but it is usually denoted as the ground-truth, while the second approach is relatively cheap, but error-prone. In this part, we investigate how a noisy fine-grain label, *e.g.*, generated from a coarse-grain class by using unsupervised clustering methods, may affect effectiveness of training with fine-grain labels.

To this end, we keep the coarse-grain labels fixed and randomly change the fine-grain labels within each coarse-grain class to simulate the effect of noisy labeling. We tune the probability of randomizing the fine-grain labels, *i.e.*, randomness factor, to control the amount of noise in the experiments. Table 6.8 shows the results under different randomness factors for CIFAR-10 dataset. We can see that with increased randomness factor, both $A_{FC}^{test}$ and the improvement brought by fine-grain training, $\Delta A^{test} = A_{FC}^{test} - A_{CC}^{test}$, keep decreasing. This means that training with highly incorrect fine-grain labels may actually hurt CNN performance. Therefore, how to automatically cluster each coarse-grain class into less-noisy fine-grain classes is an important direction to explore. We leave it for future work.

Table 6.8: Testing accuracy trained with noisy fine-grain labels of CIFAR-10 dataset.

| Randomness factor | $A_{FC}^{test}$ (%) | $\Delta A^{test}$ (%) |
|---|---|---|
| 0 | 99.20 | 0.78 |
| 0.01 | 98.94 | 0.52 |
| 0.03 | 98.55 | 0.13 |
| 0.1 | 98.12 | -0.30 |
| 0.3 | 97.72 | -0.70 |

### 6.4.3 Varying number of coarse-grain classes

We further investigate how the number of coarse-grain classes affects the effectiveness of fine-grain labels. With fine-grain labels, the neural network is encouraged to learn more features than it needs when trained with only coarse-grain labels, and these extra features help in network generalization, *i.e.*, improving the testing accuracy. We

conjecture that, to achieve high testing accuracy, a certain number of features needs to be learned by the network. Fine-grain labels help learn more features, however, with more coarse-grain classes, more features will be learned from only coarse-grain labels and hence it may be sufficient for classifying the test set, even without fine-grain labels. In other words, fine-grain labels bring diminishing returns when the number of coarse-grain classes increases.

To verify this, we experiment by varying the number of coarse-grain classes in the CIFAR-100 dataset and the results are shown in Table 6.9. We can see that with increasing number of coarse-grain classes, *i.e.*, from 5, 10, 15 to 20, the benefit from fine-grain training, *i.e.*, $\Delta A^{test}$, decreases, which is consistent with our expectation. In the case of CIFAR-100 dataset, when the number of coarse-grain classes goes beyond 15, the improvement brought by fine-grain labeling is negligible. However, this threshold is application and dataset dependent and should be determined experimentally in a case-by-case fashion.

Table 6.9: Testing accuracy, trained with coarse-grain *vs.* fine-grain labels, when varying number of coarse-grain classes in CIFAR-100 dataset. The coarse-grain class index follows the same order as in Table 6.2. The values inside the parenthesis in column $A_{FC}^{test}$ is $\Delta A^{test}$, the calculated improvement of fine-grain training over coarse-grain training.

| Coarse-grain class index | | | | | | | | | | | | | | | | | | | | Total | $A_{CC}^{test}$ (%) | $A_{FC}^{test}$ (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | | | |
| ✓ | | | | | | | | ✓ | | | ✓ | ✓ | | | | ✓ | | | | 5 | 80.53 | 83.22 (+2.69) |
| ✓ | ✓ | | | | | | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | 10 | 81.42 | 83.44 (+2.02) |
| ✓ | ✓ | ✓ | | ✓ | | | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 15 | 85.14 | 85.30 (+0.16) |
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 20 | 85.04 | 85.05 (+0.01) |

## 6.5 Discussion

In this chapter, we investigate the intriguing problem of how label granularity impacts CNN-based image classification. Our extensive experimentation shows that using fine-grain labels, rather than the target coarse-grain labels, can lead to higher accuracy and training data efficiency by improving both network optimization and generalization.

For example, trained with fine-grain labels, only 40% of the traininig data samples is needed to achieve even higher testing accuracy. Our results further suggest two practical applications: (i) with sufficient human resources, one can improve CNN accuracy by re-labeling the dataset with fine-grain labels, and (ii) with limited human resources, to improve CNN performance, rather than collecting more training data, one may instead collect fine-grain labels for the existing data.

# Chapter 7

# Related work

## 7.1 Learning for efficient many-core systems

DVFS control for multi-core systems has been widely studied [3, 18, 76, 77, 78, 79, 80, 81, 82, 83]. Reinforcement Learning (RL) [33, 84, 36, 85] and supervised learning methods [86, 87, 35, 34] have recently been applied for single core systems or systems with a modest numbers of cores. While both have shown to be effective, no scalability study exists. Targeting a limit study of DVFS scaling for performance improvement, Isci *et al.* [3] have proposed MaxBIPS which exhaustively searches for the best combination of V/F levels that maximizes performance under a given power constraint. However, MaxBIPS is not a scalable approach given that it relies on exhaustive space state search. Winter *et al.* [18] consider the previously proposed algorithm LinOpt [88] and determine its complexity to be polynomial in $N$ (number of cores), which is unsuitable for many-core systems [18]. They further propose the Steepest Drop method with a complexity of $O(\alpha \cdot N \cdot log(N))$ where $\alpha$ is the number of VF levels. [29] proposed a fast approach that uses a MaxBIPS-based [3] formulation, however, with a different objective: aggregated frequency rather than system throughput. Although aforementioned prior works have targeted power-constrained performance optimization, they do not consider TDP overshoot as an additional metric. Note that exceeding TDP is tied to thermal emergencies, which may

affect overall system reliability. Authors of [35] propose to perform multiple resource management, including power in addition to performance. Yet, their approach requires off-line profiling and training of an Artificial Neural Network. Reinforcement Learning (RL) [33, 84, 36] is able to adapt workload changes and deliver better energy/power savings or performance improvement under power constraints than conventional methods. Although a RL-based approach has the strength of learning the model of both the system and the workload, none of the aforementioned contributions address the scalability of RL in many-core systems. In order to mitigate the scalability problem, Juan *et al.* [36] propose to group cores into clusters, each managed by supervisors which moves each cluster back to nominal V/F level whenever the budget is exceeded. Nevertheless, the proposed recovery mechanism, the under-investigated importance of granularity of the clusters and the use of a uniform power budget across clusters renders this method suboptimal.

Prior work has also addressed thermal constrained performance optimization [89, 90, 79]. However, such approaches need separate procedures for model learning and action decision. In contrast, RL learns the optimal actions in a model-free manner. Furthermore, thermal constraints are usually local (per-core), different from the global power constraint we consider. Since the problem scope is different, we cannot provide a direct comparison with these approaches.

To be able to assess the benefit of any performance improvement approach, one needs to rely on simple, yet accurate enough models that can be used in taking online decisions. Prior art has proposed periodic migration of threads across different core types to sample the power and performance statistics [91]. Such an approach can be costly, given the migration cost for increased core count. Therefore, it becomes necessary to predict the power of a thread running on a given core based on performance counters customarily available in modern architectures. In this thesis, we employ the power-performance modeling methodology proposed by Juan *et al.* [34, 92, 21], to predict power as a function of the operating frequency under given workload behaviors.

## 7.2 Speeding up training of RL-based DVFS algorithms

Bayesian Optimization (BO) is a well-known sample-efficient optimization method for black-box functions [10, 12, 11], and has achieved successes in various domains, *e.g.*, robotics, vehicle control, model selection, *etc* [93, 37]. Recently, BO has been applied to hyperparameter tuning for deep neural networks [11, 94] and proven effective in finding better solutions with fewer training samples.

To the best of our knowledge, no prior art has applied BO to speeding up RL training, and in this thesis, we explore this in the context of RL-DVFS. One of the major challenges in BO is the curse of dimensionality [15, 37]. Several researchers have tried to alleviate this issue, but with strong assumptions, *e.g.*, the target function varies only along a low dimensional subspace [95, 40, 96] or it is an additive function of multiple subspaces [97, 15]. Unfortunately, the target function in DVFS cannot be modeled as an additive function of subspaces, nor can it be shrunk to one lower dimensional subspace.

## 7.3 Convolutional neural networks: efficient inference

There are several prior works targeting CNN acceleration [6, 49, 44, 50]. Model compression [6, 49, 98, 48] is a popular approach of reducing CNN memory requirement and runtime via weight pruning. [6] proposed to prune connections and fine-tune the network progressively which results in high compression rate. However, due to the non-structured sparsity generated by this method, it also needs specialized hardware to realize high speedup [99]. In light of this, [49] used group lasso to generate structured sparsity and speed up CNNs on general-purpose processors, *e.g.*, CPU and GPU.

CNN model binarization or quantization methods [100, 101, 50, 102, 103, 104] quantize CNN weights and/or activations into low-precision fewer-bit representations. Thereafter, they are able to both reduce memory cost and speedup computation by using efficient hardware units. [100] uses binary weights rather than continuous-valued weights

in CNN models, which is not only able to save memory space, but also greatly speedup convolution via replacing multiply-accumulate operations by simple accumulations. Ding *et al.,* [50] reduces the number of bits of CNN weights through its binary representation, which can be sped up by using shift-add operation rather than expensive multipliers on hardware. [101, 51] further quantize the CNN intermediate activations, resulting in both binary weight and input, which can be further accelerated via efficient XNOR operation.

Low rank approximation methods [52, 5, 105] speed up convolution computation by exploiting the redundancies of the convolutional kernel using low-rank tensor decompositions. The original convolution layer is then replaced by a sequence of convolution layers with low-rank filters, which have a much lower total computational cost. [52] exploit cross-channel or filter redundancy to construct rank-one basis of filters in the spatial domain. [5] use non-linear least squares to compute a low-rank CP-decomposition of the filters into fewer rank-one tensors and then fine-tune the entire network.

The closest work to ours is PerforatedCNNs [44] which, inspired by the idea of loop perforation [106], reduces the computation cost in convolution layers by exploiting the spatial redundancy. Nevertheless, PerforatedCNNs use a dataset dependent method to generate an irregular output mask that determines which neuron should be computed exactly. In addition, PerforatedCNNs need a mask, and hence loading overhead, at runtime to determine the value for interpolation, while ViP only depends on the intermediate activations of the CNN layer without extra parameters. Finally, PerforatedCNNs also considered the use of a pooling-structured mask, but it can only be applied to the layers immediately preceding a pooling layer and the associated interpolation method is nearest neighbor. In contrast, our method can be applied to any convolution layer in the network. Furthermore, we are able to show that the ViP method is able to achieve higher speedup with lower accuracy degradation.

## 7.4 Convolutional neural networks: efficient training

To the best of our knowledge, this is the first work to analyze the use of finer-gain labeling for improving accuracy and training data efficiency for CNN-based image classification tasks. Though there has been significant prior work looking into the hierarchy of classes/categories [107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118], our work has a distinct objective compared to prior art. Some of the prior work [115, 108, 113] aim to utilize the hierarchical label information to improve classification accuracy for the finest categories. On the theory side, Dekel *et al.* [108] propose a learning framework using large margin kernel methods and Bayesian analysis to deal with the classification problem with hierarchical label structures. Cesa-Bianchi *et al.* [115] propose a new loss function and use Support Vector Machine (SVM) as well as a probabilistic data model so that higher accuracy can be achieved with exponentially fast convergence speed. From a more practical viewpoint, Zhao *et al.* [113] leverage hierarchical information of the class structure and select different feature subsets for super-classes. Other work [116][109][112] aim to predict either coarse- or fine-grain labels conditioning on the confidence level. Deng *et al.* [116] optimize the trade-off between specificity (how fine-grain the predicted label is) and accuracy, while Bi *et al.* [109] develop a Bayes-optimal classifier to minimize the Bayesian risk. More recently, Wang *et al.* propose to stop the prediction process for a coarse-grain label so as to avoid an incorrect prediction. In addition, other prior work [110][117][111] focuses on the understanding of hierarchical labels. For example, Song *et al.* [110] study dataless hierarchical text classification with unsupervised methods. Hoyoux *et al.* [117] show some counter-examples where using hierarchical methods degrades the accuracy, and explore the reasons for such results. Oh [111] studies the combination of hierarchical classification and top-$k$ accuracy.

However, all these studies aim to increase the classification accuracy of fine-grain classes. Instead, we focus on the case of coarse-grain classes being the target of classification task, and we explore whether directly training with finer-grain labels can achieve

higher classification accuracy on coarse-grain classes than training with coarse-grain labels.

A work close to ours come from Mo *et al.* [119], who propose active over-labeling to generate finer-grain labels than the target coarse-grain labels, and demonstrate that fine-grained label data can improve precision of a classifier for the coarse-grained concept. Similar ideas were also studied by other prior work [70, 120, 71, 121, 122]. However, none of them explores deep learning models which are very different from conventional machine learning models, *e.g.*, Support Vector Machine (SVM), logistic regression, *etc*. Fradkin [71] performs experiments on linear and non-linear SVM, and finds that fine-grain training can improve accuracy for linear SVM since fine-grain labeling can learn a piece-wise linear decision boundary that better approximate the true non-linear boundary. However, fine-grain training does not help non-linear (RBF-kernel) SVM due to their inherent non-linearity. CNNs are highly non-linear models and relatively more difficult to optimize [123]. No results using CNNs have yet been shown on this topic.

# Chapter 8

# Conclusion and future work

In this thesis, we propose several approaches on both learning *for* and *with* efficient computing systems. The first angle we take is applying reinforcement learning to improve many-core system performance while satisfying both power and performance constraints. Our results show that OD-RL is **100x** faster on a 512-core system comparing to Steepest Drop. Our approach achieves up to **98%** saving on budget overshoot, **23%** higher energy efficiency, and is consistently better than Steepest Drop when the number of cores scales up. Considering both global power constraints and per application performance requirements, pa-OD-RL delivers, on average, (i) **17.8x** more epochs satisfying prescribed performance requirements, (ii) **5.6x** performance gain, and (iii) **20.0x** better performance-power trade-off (AOB metric) than the priority-unaware OD-RL, while both have similar runtime and scalability to thousand of cores. We then proceed to accelerate training of RL-based DVFS algorithoms by using Bayesian optimization. We introduce an hybrid method combining model complexity reduction and iterative-Bayesian Optimization method to overcome the curse of dimensionality of BO methods and speed up training of RL-based DVFS control algorithms. Based on deeper investigation of the iterative-BO method, we propose *iterative-BO with restart* which further boosts speedup to **37.4x**.

With the advent of deep learning, we focus on convolutional neural networks, which have achieved great success in visual tasks. We propose methods to improve the efficiency for both inference and training phases of CNN. (1) To improve CNN inference efficiency, we propose the Virtual Pooling (ViP) method that exploits the spatial redundancy of the input and reduces CNN computation by using a larger stride convolution and then recovering the output with linear interpolation (which is very efficient). We test and validate our method extensively on four CNN models, three representative datasets, both desktop and mobile platforms, and two primary learning tasks, *i.e.*, image classification and object detection. When running on Nvidia Titan X GPU, ViP is able to speedup VGG-16 by **2.1x** with less than 1.5% accuracy degradation in the image classification task, and speedup faster-rcnn by **1.8x** with 0.025 mAP degradation. Furthermore, we show that ViP is able to generate a set of models with different speedup-accuracy trade-offs. This provides CNN practitioners a tool for finding the model most suitable for their needs. (2) To improve CNN training efficiency, we introduce the intriguing property of how label granularity impacts CNN-based image classification. Our extensive experimentation shows that using fine-grain labels, rather than the target coarse-grain labels, can lead to higher accuracy and training data efficiency by improving both network optimization and generalization. For example, trained with fine-grain labels, only 40% of the traininig data samples is needed to achieve even higher testing accuracy. Our results further suggest two practical applications: (i) with sufficient human resources, one can improve CNN accuracy by re-labeling the dataset with fine-grain labels, and (ii) with limited human resources, to improve CNN performance, rather than collecting more training data, one may instead collect fine-grain labels for the existing data.

Future work in learning for efficient computing systems can explore advanced end-to-end learning frameworks, *e.g.*, convolutional neural networks, to directly learn coherent and efficient resource management and scheduling policies for operating systems and even new computer architectures. To further improve CNN inference efficiency, rather

than simple fixed-weight linear interpolation, we may extend ViP method to include trainable weight during interpolation and larger stride in convolution, which leads to weighted-average interpolation and may improve both efficiency and accuracy. In terms of CNN training efficiency, rather than human labeling, we may explore unsupervised clustering methods to automatically split existing coarse-grain classes into fine-grain classes, and we shall investigate and understand the theory behind this phenomenon and the limit of fine-grain labeling, *e.g.*, to which level shall we stop splitting the class due to diminishing returns.

# Bibliography


[1] "https://github.com/fmfn/bayesianoptimization." xi, 7, 43

[2] LinkViz, "https://www.mathworks.com/help/deeplearning /examples/visualize-activations-of-a-convolutional-neural-network.html." xiii, 47

[3] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *MICRO*. IEEE Computer Society, 2006, pp. 347–358. 1, 10, 12, 17, 18, 19, 23, 24, 27, 38, 90

[4] G. Liu, J. Park, and D. Marculescu, "Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems," in *ICCD*. IEEE, 2013, pp. 54–61. 2, 11, 38

[5] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," *arXiv preprint arXiv:1412.6553*, 2014. 2, 46, 54, 65, 93

[6] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143. 2, 46, 54, 92

[7] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu, "Neuralpower: Predict and deploy energy-efficient convolutional neural networks," in *Asian Conference on Machine Learning*, 2017, pp. 622–637. 2, 46

[8] A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 1998. 5, 6, 13, 23, 40

[9] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992. 6

[10] C. E. Rasmussen and C. K. Williams, *Gaussian processes for machine learning*. MIT press Cambridge, 2006, vol. 1. 7, 39, 43, 92

[11] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959. 7, 43, 92

[12] P. Hennig and C. J. Schuler, "Entropy search for information-efficient global optimization," *Journal of Machine Learning Research*, vol. 13, no. Jun, pp. 1809–1837, 2012. 7, 92

[13] J. M. Hernández-Lobato, M. W. Hoffman, and Z. Ghahramani, "Predictive entropy search for efficient global optimization of black-box functions," in *Advances in neural information processing systems*, 2014, pp. 918–926. 7

[14] Z. Wang and S. Jegelka, "Max-value entropy search for efficient bayesian optimization," *arXiv preprint arXiv:1703.01968*, 2017. 7

[15] K. Kandasamy, J. Schneider, and B. Póczos, "High dimensional bayesian optimisation and bandits via additive models," in *International Conference on Machine Learning*, 2015, pp. 295–304. 7, 41, 44, 92

[16] D. J. MacKay, "Probable networks and plausible predictionsâĂŤa review of practical bayesian methods for supervised neural networks," *Network: Computation in Neural Systems*, vol. 6, no. 3, pp. 469–505, 1995. 7

[17] Z. Chen, D. Stamoulis, and D. Marculescu, "Profit: priority and power/performance optimization for many-core systems," *IEEE Transactions on*

*Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 10, pp. 2064–2075, 2018. 11

[18] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *PACT*. ACM, 2010, pp. 29–40. 12, 23, 90

[19] Z. Chen and D. Marculescu, "Distributed reinforcement learning for power limited many-core system performance optimization," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 1521–1526. 13, 38, 39, 40

[20] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011. 16

[21] E. Cai, D. C. Juan, S. Garg, J. Park, and D. Marculescu, "Learning-based power/performance optimization for many-core systems with extended-range voltage/frequency scaling," *IEEE TCAD*, vol. 35, no. 8, pp. 1318–1331, Aug 2016. 19, 20, 21, 38, 91

[22] D. Stamoulis and D. Marculescu, "Can we guarantee performance requirements under workload and process variations?" in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. ISLPED '16, 2016, pp. 308–313. 19, 20

[23] D. Stamoulis, D. Rodopoulos, B. H. Meyer, D. Soudris, F. Catthoor, and Z. Zilic, "Efficient reliability analysis of processor datapath using atomistic bti variability models," in *GLSVSLI*. ACM, 2015, pp. 57–62. 20

[24] E. Cai, D. Stamoulis, and D. Marculescu, "Exploring aging deceleration in finfet-based multi-core systems," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–8. 20

[25] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO-42nd*. IEEE, 2009, pp. 469–480. 20, 21

[26] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011. 20, 21

[27] T. Ebi, D. Kramer, W. Karl, and J. Henkel, "Economic learning for thermal-aware power budgeting in many-core architectures," in *CODES+ ISSS, 2011 Proceedings of the 9th International Conference on*. IEEE, 2011, pp. 189–196. 26

[28] J. W. J. Williams, "Algorithm-232-heapsort," pp. 347–348, 1964. 32

[29] G.-Y. Pan, J. Yang, J.-Y. Jou, and B.-C. C. Lai, "Scalable global power management policy based on combinatorial optimization for multiprocessors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 4, p. 70, 2015. 34, 90

[30] P. P. Pande, R. G. Kim, W. Choi, Z. Chen, D. Marculescu, and R. Marculescu, "The (low) power of less wiring: Enabling energy efficiency in many-core platforms through wireless noc," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 165–169. 38

[31] R. G. Kim, W. Choi, Z. Chen, P. P. Pande, D. Marculescu, and R. Marculescu, "Wireless noc and dynamic vfi codesign: Energy efficiency without performance penalty," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 7, pp. 2488–2501, 2016. 38

[32] R. G. Kim, W. Choi, Z. Chen, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, "Imitation learning for dynamic vfi control in large-scale manycore systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017. 38

[33] W. Liu, Y. Tan, and Q. Qiu, "Enhanced q-learning algorithm for dynamic power management with performance constraint," in *DATE*. European Design and Automation Association, 2010, pp. 602–605. 39, 40, 90, 91

[34] D.-C. Juan, S. Garg, J. Park, and D. Marculescu, "Learning the optimal operating point for many-core systems with extended range voltage/frequency scaling," in *CODES+ ISSS*. IEEE, 2013, pp. 1–10. 39, 90, 91

[35] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO*. IEEE, 2008, pp. 318–329. 39, 90, 91

[36] D.-C. Juan and D. Marculescu, "Power-aware performance increase via core/uncore reinforcement control for chip-multiprocessors," in *ISLPED*. ACM, 2012, pp. 97–102. 39, 90, 91

[37] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010. 39, 44, 92

[38] S. Sadasivam, J. Lee, Z. Chen, and R. Jain, "Efficient reinforcement learning for automating human decision-making in soc design," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6. 39

[39] I. Szita and A. Lörincz, "Learning tetris using the noisy cross-entropy method," *Learning*, vol. 18, no. 12, 2006. 40, 43

[40] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, N. De Freitas *et al.*, "Bayesian optimization in high dimensions via random embeddings." in *IJCAI*, 2013, pp. 1778–1784. 41, 92

[41] "http://scikit-learn.org." 43

[42] D. Stamoulis, E. Cai, D.-C. Juan, and D. Marculescu, "Hyperpower: Power-and memory-constrained hyper-parameter optimization for neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 19–24. 46

[43] D. Marculescu, D. Stamoulis, and E. Cai, "Hardware-aware machine learning: modeling and optimization," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018, p. 137. 46

[44] M. Figurnov, A. Ibraimova, D. P. Vetrov, and P. Kohli, "Perforatedcnns: Acceleration through elimination of redundant convolutions," in *Advances in Neural Information Processing Systems*, 2016, pp. 947–955. 47, 53, 57, 58, 59, 60, 92, 93

[45] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014. 48, 50, 55

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778. 48, 50, 55

[47] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, "Training quantized nets: A deeper understanding," in *Advances in Neural Information Processing Systems*, 2017, pp. 5811–5821. 51

[48] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800. 53, 92

[49] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 2074–2082. 54, 65, 92

[50] R. Ding, Z. Liu, R. Shi, D. Marculescu, and R. Blanton, "Lightnn: Filling the gap between conventional deep neural networks and binarized networks," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM, 2017, pp. 35–40. 54, 65, 92, 93

[51] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542. 54, 93

[52] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," in *Proceedings of the British Machine Vision Conference. BMVA Press*, 2014. 54, 93

[53] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014. 55, 61

[54] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99. 55, 65

[55] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255. 55

[56] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014. 55

[57] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010. 55

[58] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in

*Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678. 56

[59] S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran *et al.*, "Dsd: Dense-sparse-dense training for deep neural networks," *arXiv preprint arXiv:1607.04381*, 2016. 58

[60] GithubPerf, "https://github.com/mfigurnov/perforated-cnn-caffe." 60

[61] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015. 68

[62] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255. 68, 69, 72

[63] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009. 68, 69, 72

[64] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014. 69

[65] Z. Chen, R. Ding, T.-W. Chin, and D. Marculescu, "Understanding the impact of label granularity on cnn-based image classification," in *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2018, pp. 895–904. 69

[66] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995. 71, 72

[67] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 630–645. 74

[68] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016. 74, 84

[69] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778. 74, 78

[70] A. Hoffmann, R. Kwok, and P. Compton, "Using subclasses to improve classification learning," in *European Conference on Machine Learning*. Springer, 2001, pp. 203–213. 80, 83, 95

[71] D. Fradkin, "Clustering inside classes improves performance of linear classifiers," in *Tools with Artificial Intelligence, 2008. ICTAI'08. 20th IEEE International Conference on*, vol. 2. IEEE, 2008, pp. 439–442. 80, 95

[72] H. Bilen and A. Vedaldi, "Weakly supervised deep detection networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2846–2854. 82

[73] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008. 82

[74] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833. 83

[75] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014. 83

[76] J. Li and J. F. Martinez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *HPCA*. IEEE, 2006, pp. 77–87. 90

[77] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: adaptive dvfs and thread packing under power caps," in *MICRO*. ACM, 2011, pp. 175–185. 90

[78] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *ISLPED*. IEEE, 2007, pp. 38–43. 90

[79] Y. Wang, K. Ma, and X. Wang, "Temperature-constrained power control for chip multiprocessors with online model estimation," in *ACM SIGARCH computer architecture news*, vol. 37, no. 3. ACM, 2009, pp. 314–324. 90, 91

[80] J. Sartori and R. Kumar, "Distributed peak power management for many-core architectures," in *DATE*. IEEE, 2009, pp. 1556–1559. 90

[81] J. M. Cebrián, J. L. Aragon, and S. Kaxiras, "Power token balancing: Adapting cmps to power constraints for parallel multithreaded workloads," in *IPDPS*. IEEE, 2011, pp. 431–442. 90

[82] E. Cai and D. Marculescu, "Tei-turbo: temperature effect inversion-aware turbo boost for finfet-based multi-core systems," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 500–507. 90

[83] ——, "Temperature effect inversion-aware power-performance optimization for finfet-based multicore systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1897–1910, 2017. 90

[84] Y. Tan, W. Liu, and Q. Qiu, "Adaptive power management using reinforcement learning," in *ICCAD*. ACM, 2009, pp. 461–467. 90, 91

[85] G.-Y. Pan, J.-Y. Jou, and B.-C. Lai, "Scalable power management using multilevel reinforcement learning for multiprocessors," *ACM TODAES*, vol. 19, no. 4, p. 33, 2014. 90

[86] G. Dhiman and T. S. Rosing, "Dynamic power management using machine learning," in *ICCAD*. ACM, 2006, pp. 747–754. 90

[87] H. Jung and P. M., "Improving the efficiency of power management techniques by using bayesian classification," in *ISQED*. IEEE, 2008, pp. 178 – 183. 90

[88] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 363–374. 90

[89] A. Bartolini, M. Cacciari, A. Tilli, L. Benini, and M. Gries, "A virtual platform environment for exploring power, thermal and reliability management control strategies in high-performance multicores," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*. ACM, 2010, pp. 311–316. 91

[90] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini, "A distributed and self-calibrating model-predictive controller for energy and thermal management of high-performance multicores," in *DATE'11*. IEEE, pp. 1–6. 91

[91] F. Wang et al., "Variation-aware task and communication mapping for mpsoc architecture," *IEEE TCAD*, vol. 30, no. 2, pp. 295–307, Feb 2011. 91

[92] D.-C. Juan, "A learning-based framework incorporating domain knowledge for performance modeling," 2014. 91

[93] G. Malkomes, C. Schaff, and R. Garnett, "Bayesian optimization for automated model selection," in *Advances in Neural Information Processing Systems*, 2016, pp. 2900–2908. 92

[94] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams, "Scalable bayesian optimization using deep neural networks," in *International Conference on Machine Learning*, 2015, pp. 2171–2180. 92

[95] B. Chen, R. Castro, and A. Krause, "Joint optimization and variable selection of high-dimensional gaussian processes," *arXiv preprint arXiv:1206.6396*, 2012. 92

[96] J. Djolonga, A. Krause, and V. Cevher, "High-dimensional gaussian process bandits," in *Advances in Neural Information Processing Systems*, 2013, pp. 1025–1033. 92

[97] D. K. Duvenaud, H. Nickisch, and C. E. Rasmussen, "Additive gaussian processes," in *Advances in neural information processing systems*, 2011, pp. 226–234. 92

[98] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," *arXiv preprint arXiv:1808.06866*, 2018. 92

[99] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 243–254. 92

[100] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131. 92

[101] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016. 92, 93

[102] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828. 92

[103] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *arXiv preprint arXiv:1702.03044*, 2017. 92

[104] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016. 92

[105] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in neural information processing systems*, 2014, pp. 1269–1277. 93

[106] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134. 93

[107] A. Sun and E.-P. Lim, "Hierarchical text classification and evaluation," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE, 2001, pp. 521–528. 94

[108] O. Dekel, J. Keshet, and Y. Singer, "Large margin hierarchical classification," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 27. 94

[109] W. Bi and J. T. Kwok, "Hierarchical multilabel classification with minimum bayes risk," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 101–110. 94

[110] Y. Song and D. Roth, "On dataless hierarchical text classification." in *AAAI*, vol. 7, 2014. 94

[111] S. Oh, "Top-k hierarchical classification." in *AAAI*, 2017, pp. 2450–2456. 94

[112] Y. Wang, Q. Hu, Y. Zhou, H. Zhao, Y. Qian, and J. Liang, "Local bayes risk minimization based stopping strategy for hierarchical classification," in *Data Mining (ICDM), 2017 IEEE International Conference on*. IEEE, 2017, pp. 515–524. 94

[113] H. Zhao, P. Zhu, P. Wang, and Q. Hu, "Hierarchical feature selection with recursive regularization," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 3483–3489. 94

[114] D. Wang, H. Huang, C. Lu, B.-S. Feng, L. Nie, G. Wen, and X.-L. Mao, "Supervised deep hashing for hierarchical labeled data," in *AAAI*, 2018. 94

[115] N. Cesa-Bianchi, C. Gentile, and L. Zaniboni, "Incremental algorithms for hierarchical classification," *Journal of Machine Learning Research*, vol. 7, no. Jan, pp. 31–54, 2006. 94

[116] J. Deng, J. Krause, A. C. Berg, and L. Fei-Fei, "Hedging your bets: Optimizing accuracy-specificity trade-offs in large scale visual recognition," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012, pp. 3450–3457. 94

[117] T. Hoyoux, A. J. Rodríguez-Sánchez, and J. H. Piater, "Can computer vision problems benefit from structured hierarchical classification?" *Machine Vision and Applications*, vol. 27, no. 8, pp. 1299–1312, 2016. 94

[118] R. Cerri, R. C. Barros, and A. C. de Carvalho, "Hierarchical classification of gene ontology-based protein functions with neural networks," in *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE, 2015, pp. 1–8. 94

[119] Y. Mo, S. D. Scott, and D. Downey, "Learning hierarchically decomposable concepts with active over-labeling," in *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE, 2016, pp. 340–349. 95

[120] Y. Luo, "Can subclasses help a multiclass learning problem?" in *Intelligent Vehicles Symposium, 2008 IEEE*. IEEE, 2008, pp. 214–219. 95

[121] N. Ahmed and M. Campbell, "On estimating simple probabilistic discriminative models with subclasses," *Expert Systems with Applications*, vol. 39, no. 7, pp. 6659–6664, 2012. 95

[122] M. Ristin, J. Gall, M. Guillaumin, and L. Van Gool, "From categories to subcategories: large-scale image classification with partial class label refinement," in

*Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on*. IEEE, 2015, pp. 231–239. 95

[123] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256. 95

# Appendix A

# Proof

Proof for Proposition 1 is presented here.

*Proof.* We prove this bound by first bounding the error of the ViP layer $\mathcal{O}^{(l_s)}$, and then bounding the error accumulated to higher layers.

For the ViP layer $\mathcal{O}^{(l_s)}$, $\forall c, h, w$, the ViP error of $\mathcal{O}^{(l_s)}_{c,h,w}$ can be bounded:

$$\begin{aligned} & |\mathcal{O}^{(l_s)ViP}_{c,h,w} - \mathcal{O}^{(l_s)Orig}_{c,h,w}| \\ =& |\frac{1}{K} \sum_{\substack{k=1 \\ (h_k,w_k)\in\mathcal{N}_{h,w}}}^{K} \mathcal{O}^{(l_s)ViP}_{c,h_k,w_k} - \mathcal{O}^{(l_s)Orig}_{c,h,w}| \\ \leq& \frac{1}{K} \sum_{\substack{k=1 \\ (h_k,w_k)\in\mathcal{N}_{h,w}}}^{K} |\mathcal{O}^{(l_s)ViP}_{c,h_k,w_k} - \mathcal{O}^{(l_s)Orig}_{c,h,w}| \\ \leq& L * d_{max} = \sqrt{2}L \end{aligned} \tag{A.1}$$

where $\mathcal{N}_{h,w}$ is the set of neighbor locations of $(h, w)$ that are averaged to compute the ViP value for pixel $(h, w)$, and $d_{max}$ is the maximum $l_2$-norm distance of the location $(c, h, w)$ and a neighbor location $(c, h_k, w_k)$, which is $\sqrt{2}$ in ViP.

Then, we bound the error accumulated from layer $l-1$ to $l$, *i.e.*, $|\mathcal{O}^{(l)ViP}_{c',h,w} - \mathcal{O}^{(l)Orig}_{c',h,w}|$. Due to Eq. (1),

$$\begin{aligned}\mathcal{O}^{(l)ViP}_{c',h,w} &= (\sum_{c=1}^{C^{(l)}} \sum_{m,n=-\lfloor M^{(l)}/2 \rfloor}^{\lfloor M^{(l)}/2 \rfloor} \mathcal{O}^{(l-1)ViP}_{c,s\cdot h-m,s\cdot w-n} * \mathcal{W}^{(l)}_{c',c,m,n})_+ \\ \mathcal{O}^{(l)Orig}_{c',h,w} &= (\sum_{c=1}^{C^{(l)}} \sum_{m,n=-\lfloor M^{(l)}/2 \rfloor}^{\lfloor M^{(l)}/2 \rfloor} \mathcal{O}^{(l-1)Orig}_{c,s\cdot h-m,s\cdot w-n} * \mathcal{W}^{(l)}_{c',c,m,n})_+\end{aligned} \tag{A.2}$$

where $(.)_+$ is the ReLU activation function. Due to the fact that $\forall x, y \in R, |(x)_+ - (y)_+| \leq |x-y|$, we have:

$$\begin{aligned}&|\mathcal{O}^{(l)ViP}_{c',h,w} - \mathcal{O}^{(l)Orig}_{c',h,w}| \\ \leq &|\sum_{c=1}^{C^{(l)}} \sum_{m,n=-\lfloor M^{(l)}/2 \rfloor}^{\lfloor M^{(l)}/2 \rfloor} (\mathcal{O}^{(l-1)ViP}_{c,s\cdot h-m,s\cdot w-n} - \mathcal{O}^{(l-1)Orig}_{c,s\cdot h-m,s\cdot w-n}) \mathcal{W}^{(l)}_{c',c,m,n}|\end{aligned} \tag{A.3}$$

Using Cauchy-Schwarz inequality, we have:

$$\begin{aligned}&|\mathcal{O}^{(l)ViP}_{c',h,w} - \mathcal{O}^{(l)Orig}_{c',h,w}| \\ \leq &\sqrt{\sum_{c=1}^{C^{(l)}} \sum_{m,n=-\lfloor M^{(l)}/2 \rfloor}^{\lfloor M^{(l)}/2 \rfloor} (\mathcal{O}^{(l-1)ViP}_{c,s\cdot h-m,s\cdot w-n} - \mathcal{O}^{(l-1)Orig}_{c,s\cdot h-m,s\cdot w-n})^2} \\ &* \sqrt{\sum_{c=1}^{C^{(l)}} \sum_{m,n=-\lfloor M^{(l)}/2 \rfloor}^{\lfloor M^{(l)}/2 \rfloor} \mathcal{W}^{(l)}_{c',c,m,n}} \\ \leq &\sqrt{C^{(l)}} M^{(l)} B^{(l)} \max_{c',h,w} |\mathcal{O}^{(l-1)ViP}_{c,h,w} - \mathcal{O}^{(l-1)Orig}_{c,h,w}|\end{aligned} \tag{A.4}$$

Therefore, accumulating the error from the ViP layer $l_s$ to layer $l_e$, we have:

$$|\mathcal{O}^{(l_e)ViP}_{c',h,w} - \mathcal{O}^{(l_e)Orig}_{c',h,w}| \leq \sqrt{2} L \prod_{l=l_s+1}^{l_e} \sqrt{C^{(l)}} M^{(l)} B^{(l)}, \tag{A.5}$$

Thus, the $l_2$-norm of the output error is bounded by:

$$\begin{aligned}&||\mathcal{O}^{(l_e)ViP} - \mathcal{O}^{(l_e)Orig}||_2 \\ \leq &\sqrt{2} L \sqrt{C'^{(l_e)} H^{(l_e)} W^{(l_e)}} \prod_{l=l_s+1}^{l_e} \sqrt{C^{(l)}} M^{(l)} B^{(l)}.\end{aligned} \tag{A.6}$$

□