The Dissertation Committee for Chirag Sakhuja
certifies that this is the approved version of the following dissertation:

Incorporating Prior Knowledge to Efficiently Design Deep Learning Accelerators

Committee:

Dr. Calvin Lin, Supervisor

Dr. Derek Chiou

Dr. Mattan Erez

Dr. Angshuman Parashar

Dr. Atlas Wang

Incorporating Prior Knowledge to Efficiently Design Deep Learning Accelerators

by

Chirag Sakhuja

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2024

Dedication

To my parents, Vivek and Monica, and my sisters, Shruti and Urvi.

Acknowledgments

The path to a PhD is mired with taxing twists and turns, and, more than the research, it was the people around me that kept me going. I owe deep thanks to many, but in this brief section I express my gratitude to those who had the most direct impact on me and my work.

First, I thank my advisor, Calvin Lin. I am grateful to have found an advisor whom I mesh so well with in terms of my research interests, work habits, and personality. Calvin taught me technical skills that I will carry with me for the rest of my life: how to find challenging and important problems to work on, how to ask the right questions to break down a problem into tractable pieces, how to use writing to gain clarity in my thinking, and how to articulate my thoughts clearly and concisely. Calvin also showed me how to be a well-rounded researcher. Throughout my PhD, he encouraged me to pursue my hobbies and to take ample breaks to balance work and life. It is with Calvin's patient guidance that I am able to complete the work I present in this dissertation.

I also thank my committee members for their continued feedback and guidance. Dr. Chiou is an inspiring researcher, teacher, and all-around person, and I am grateful to have met him and to have worked with him. His thoughtful advice shaped many of my decisions. Dr. Erez is a kind and extremely insightful advisor, and his poignant questions made my work stronger. I am also grateful for the impact he has had on the UT Austin ECE department, which I have called home for 12 years. Dr. Parashar has deep technical understanding, and I am thankful for his extensive feedback, which gave me a fresh perspective on my work. Finally, Dr. Wang, having a different background than the rest of my committee, patiently learned about my work and introduced me to new ideas, so I am thankful for his wisdom.

Next I thank my friends, who constantly helped me in one way or another throughout this journey:

- Minesh Patel is one of my closest friends and is a role model for what a researcher should be. He is incredibly hard-working, passionate, inspiring, and supportive, and I am grateful to have him as a friend.
- Alex Hsu has known me for 15 years, and I can always trust him to give me guidance, whether it be for life or for research. Speaking of, Alex is *always* willing and excited to dig deep into technical subjects, and we have brainstormed many ideas together.
- Nikhil Dixit was one of the earliest friends I made, and it is uncanny how many similarities we have: we both went to the same Sunday school, lived in the same neighborhood, have (almost) the same birthday, double-majored in ECE and CS, and ended up working full-time on the same team after we graduated with our MS. It is always a treat to spend time with him, no matter the purpose.
- Arjun Teh and I both started working full-time at the same time, and we both quickly decided we wanted to pursue a PhD instead. Arjun was already a close friend by that point, but since then he has been my go-to friend to talk shop with about the PhD experience or to look to for support.
- Elaine Lui and I hardly knew each other when we were both at UT, but we quickly became close friends the summer after she graduated to pursue her PhD. We have since supported each other through countless ups and downs, and her kind advice always puts a smile on my face.
- Jason Math quickly became a close friend—against all odds given our big age gap—after I TAed him and later reconnected with him in the social

dance class. He is a pleasure to be around and to talk with, and he keeps me young.

- Hansel Chiang and I connected through social dance and became very close friends, spending many days together. When times were tough during my PhD, she always lent a helping hand to make sure I was eating and sleeping properly, and she was always ready to dance with me to cheer me up.
- My lab mates and collaborators: Quang Duong, Charles Hong, Carson Molder, Molly O'Neal, and Zhan Shi. They have a broad range of expertises that culminated into an exciting and rewarding research environment.
- The students I mentored: Kunaal Jha, Aparna Kumari, Caroline Li, Anoop Rachakonda, Wendy Xie, Esther Yoon, and Jack Youstra. They not only have great insight and passion but have made me a better researcher and teacher. I am excited to see what they have in store.
- My oldest group of friends: Kevin Chen, Rohan Mutalik, Kevin Pham, and Thejas Prasad. They have been with me through thick and thin, and we are as still close as ever. We are all eagerly awaiting my graduation so that I can finally join in on their extravagant plans.
- The group of friends I made when pursuing my MS: Cassidy Burden, Barak Lidsky, Prakash Luu, and Ross McNulty. They are are a blast to be around and can always lighten up the mood.
- My close dance friends: Isaac Akintitan, JC Mayo, Rianna Godula, and Hannah Wang. They kept me going with a hobby that shaped my time as a PhD student.
- Nick and Melissa Enge, who pour their heart into teaching the social dance class and mentoring the course assistants. They have given me a lifelong hobby that has brought me immense joy.

Finally, I express my deepest thanks to my family. My parents always gave me boundless opportunity to pursue my passions, and, despite never fully understanding what a PhD student in computer architecture does, they always lent me their gracious support and encouragement. My older sister, who successfully juggles a thriving career in medicine with her many other passions and goals, inspires me to give my all in everything I do. And my younger sister, who boldly uprooted her life to live in the Czech Republic, reminds me that there is a fun, positive side to everything. I am who I am thanks to my family.

Abstract

Incorporating Prior Knowledge to Efficiently Design Deep Learning Accelerators

Chirag Sakhuja, PhD
The University of Texas at Austin, 2024

SUPERVISOR: Dr. Calvin Lin

Artificial intelligence (AI) has exploded in popularity over the past decade, and its computational demand has seen commensurate increase. AI models are getting bigger, and AI applications are becoming more widespread. A common strategy to mitigate the cost of this growth—which is estimated to consume 0.5% of the world's energy by 2027 [16]—is to develop domain-specific processors, called deep learning accelerators (DLAs), that are more area-efficient and energy-efficient at processing AI workloads than traditional processors, namely CPUs and GPUs.

DLAs are efficient because they are specialized. Each DLA is developed for specific applications that necessitate anything from high-power, high-performance environments, such as datacenters, to energy-constrained, low-performance environments, such as battery-operated sensors. Consequently, as AI applications continue to evolve, new DLAs must constantly be in development, which is costly and time-consuming.

It is advantageous to reduce the cost of DLA development so that DLAs remain a relevant strategy to combat the growth of AI. One approach is to automate the development of DLAs. However, this is challenging because DLA

development involves the careful selection of many design parameters that have complex interactions with each other, so automated tools, called design space exploration (DSE) tools, can struggle to produce DLAs that are more efficient than hand-designed DLAs.

In this dissertation, we present techniques that leverage *prior knowledge* to overcome this challenge. In particular, we show how (1) hand-crafted domain information and (2) pre-collected data can efficiently guide DSE tools to automatically find design parameters that result in efficient DLAs.

We package these techniques into three open-source tools. Spotlight and Polaris are automated DSE tools that reduce development effort in the early stage (architectural exploration) and middle stage (microarchitectural exploration) of the DLA development process, respectively. And Starlight is a tool used by Polaris to predict the performance and efficiency of a particular DLA design.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Over the past decade, artificial intelligence (AI) has become a household term. Chatbots [8] and smart devices [55] are just two of the many AI applications that have garnered significant mainstream attention, and the impact of AI extends well beyond the general public's eye [70]. Decades of research, paired with the exponential growth of computational power [94], have resulted in an AI revolution that has seemingly left no industry untouched. Even this dissertation is written with AI![1]

At the heart of these AI applications is a technique called deep learning (DL), which has only recently become computationally feasible despite having origins in the 1960s [38]. At a high level, DL loosely mimics the behavior of the human brain by tying together simple learning methods into a structure called a *DL model* that is, quite literally, more powerful than the sum of its parts.

Although DL models approach or exceed human performance on some tasks [23], they do so at a hefty computational cost: the development (i.e., *training*) of a modern DL model can produce the same amount of CO_2 emissions as 5 car lifetimes [100], and the deployment of the model (i.e., *inference*) can consume $9\times$ more energy over the model's lifetime than the development stage [17].[2] Moreover, with each new generation, DL models grow larger and consume more resources [104].

Extensive effort has gone into mitigating the impact of this growth [17, 82, 93], and one strategy that has had widespread success [10, 11, 12, 43, 41, 62, 78, 89] and shows promise to combat future growth [116] is to build

[1]This is a joke.

[2]These numbers reflect the state-of-the-art on quantifying the cost of deep learning, but accurately doing so has historically been challenging [69].

specialized processors, called deep learning accelerators (DLAs), that exhibit higher area-efficiency and energy-efficiency than other processors—namely CPUs and GPUs—when executing DL models [19].

DLAs, like CPUs and GPUs, must be designed to fit their specific use case. For example, chatbots are built on massive DL models that may require large DLAs in datacenters [41], and smart devices with simple learning mechanisms may require small, area-constrained and power-constrained DLAs [125]. Furthermore, DL use cases are constantly evolving—sometimes rendering existing DLAs inefficient or obsolete [33, 107]—, so DLAs must also evolve to maintain their efficiency benefits [12, 42].

Consequently, we are constantly designing new DLAs for new use cases and/or specifications. This is time-consuming and costly [43]. A solution to reduce development effort is to introduce automation. Fortunately, DLAs exhibit an exploitable property: despite differing in specific design parameters, such as the number of compute elements, many DLAs have similar high-level architectures [47, 59, 81]. So, it seems feasible for a tool to automatically determine optimal design parameters.

To briefly summarize: To wrangle the increasing computational demand for deep learning, it is advantageous to develop DLAs, and DLA development effort can be reduced by introducing automation. To this end, we present in this dissertation novel techniques to automatically design efficient DLAs. In particular, we demonstrate how to incorporate *prior knowledge*—e.g., domain expertise or offline datasets—to quickly find design parameters that optimize the efficiency of DLAs.

The process of searching for optimal design parameters is called design space exploration (DSE), and it can be applied to different facets of the deep learning stack, which comprises (1) the DL model, (2) the mapping of that model onto a DLA, and (3) the DLA architecture. Our work focuses on the latter two facets. This type of DSE, which simultaneously explores both the mapping design space and the DLA architecture design space, is called hardware/software (HW/SW) co-design, and it is an important application of DSE [95].

HW/SW co-design is challenging because the design space has complex constraints, and the performance function—i.e., the function that maps a point in the design space to a key metric such as delay or energy consumption—can vary drastically between nearby points [48, 76, 86], making it difficult to predict. We show, by designing three novel tools, how prior knowledge can be used to overcome these challenges.

First, we design Spotlight. Spotlight is a HW/SW co-design tool that explores a vast co-design space to find both (1) optimized software mappings for each layer of a DL model and (2) optimized DLA architecture parameters. Candidate designs in Spotlight are evaluated with a low-fidelity performance estimator called an analytical model that quickly provides first-order approximations of performance. The key design goal behind Spotlight is to reduce the number of evaluations—i.e, samples—necessary to find optimized designs, and our key contribution in this work is a technique to inject hand-crafted *prior knowledge*, in the form of domain information, to efficiently guide the exploration to promising regions of the design space. Spotlight produces designs that reduce delay by 153× over the best design produced by a state-of-the-art HW/SW co-design tool, ConfuciuX [45], and reduce energy-delay product (EDP) by 44× over a state-of-the-art hand-designed DLA, Eyeriss [11].

Second, we design Starlight. Starlight is a data-driven performance estimator that predicts the EDP, as measured by RTL simulation[3], of processing a DL model layer. We are motivated to design Starlight because we wish to extend Spotlight to explore the design space of real hardware—as opposed to the design

[3]Because of limitations with our RTL simulator, Firesim [50], energy consumption is modeled analytically.

space of an analytical representation of hardware—, but we observe that a core component of Spotlight—a proxy model that learns the shape of the performance function—is too inaccurate to learn the complex behavior of real hardware. Our key insight with Starlight, which is designed to replace the inaccurate proxy model, is that prior knowledge from an analytical model can be transferred to accurately predict the performance of real hardware. Consequently, Starlight is able to predict with 99% accuracy the EDP of DLA execution as measured by RTL simulation, which is a high-fidelity estimate for the behavior of real hardware. Moreover, Starlight's training data can be collected in under one day.

Third, we design Polaris, which is the natural progression of this line of work. Polaris is a HW/SW co-design tool that has similar inputs and outputs to Spotlight, but it evaluates designs with an RTL simulator. The key design goal behind Polaris is to leverage Starlight to efficiently perform co-design with real hardware evaluation. Polaris finds designs that, on average, reduce the EDP by 2.81$\times$ over a state-of-the-art HW/SW co-design tool, DOSA [35], and by 2$\times$ over Spotlight.

The contributions of this dissertation are:

- We present two novel techniques that incorporate prior knowledge to efficiently perform design space exploration (DSE) of the co-design space comprising (1) deep learning accelerator (DLA) architectural parameters and (2) the possible ways to map a layer of a deep learning model onto the DLA. These techniques result in DSE tools that produce DLA designs and software mappings that are more efficient than prior work.
  - We develop a novel method to inject domain information into a DSE tool to efficiently guide it to regions of the design space that a domain expert expects to contain optimized design parameters. In addition to resulting in better designs than prior work, our method is more expressive than prior work.

  - We are the first to transfer prior knowledge from a low-fidelity DLA performance estimator—namely an analytical model—to a high-fidelity DLA performance estimator—namely an RTL simulator—to accurately predict the performance of real hardware. The resulting performance estimator—a data-driven model called Starlight—predicts with 99% accuracy the energy-delay product (EDP) of DLA execution as measured by RTL simulation.

- We develop three open-source tools that reduce DLA development effort.

  - We develop Spotlight, which is a DSE tool that leverages our first method of incorporating prior knowledge. Spotlight automatically produces DLA designs and software mappings that result in 153× lower EDP than the best design produced by the ConfuciuX [45] DSE tool and 44× lower EDP than the hand-designed Eyeriss [11] DLA as measured by an analytical model. Due to its sample-efficiency, Spotlight produces these designs in shorter runtime than competing algorithms.
  - We develop Starlight, which is a data-driven performance estimator that leverages our second method of incorporating prior knowledge. Starlight is trained on just 820 evaluations from an RTL simulator—which can be collected in under one day—, and it predicts with 99% accuracy the EDP of DLA execution as measured by RTL simulation. Furthermore, we demonstrate that our transfer learning approach results in higher accuracy and more reliable training than traditional data-driven approaches.
  - We develop Polaris, which is a DSE tool that builds on Starlight to produce DLA designs and software mappings that result in 2.81× lower EDP than the best design produced by DOSA [35] and 2× lower

EDP than the best design produced by Spotlight as measured by RTL simulation. To produce these designs, Polaris evaluates just 48 samples, which can be discovered and measured in under 8 hours. We also demonstrate the benefit of Polaris' online DSE approach over the offline DSE approach typical of prior work.

The remainder of this dissertation is structured as follows. Chapter 2 provides background information that is useful for understanding this dissertation, and Chapter 3 contextualizes our contributions among the body of existing literature. Chapters 4, 5, and 6 present Spotlight, Starlight, and Polaris, respectively. Finally, in Chapter 7 we end with our closing remarks and suggested directions for future work.

Chapter 2: Background

This chapter introduces an assortment of topics that provide the necessary background to understand this dissertation. The first three topics are referenced heavily by the chapters on Spotlight (Chapter 4) and Polaris (Chapter 6), and the fourth topic covers techniques that are fundamental to Starlight. The following topics are introduced:

1. The convolution operation, which is a fundamental building block of deep learning workloads.
2. The high-level architecture of deep learning accelerators.
3. Bayesian optimization, which is the optimization algorithm used throughout this work, and Gaussian processes, which are a type of machine learning method typically used by Bayesian optimization frameworks.
4. Three selected machine learning techniques: transfer learning, variational autoencoders, and deep kernel learning.

2.1 Convolution Operation

Deep learning models are built using a variety of layer types, such as fully connected, attention, and convolutional layers. The computationally dominant layers can be represented, without loss of generality, in terms of a 3-D convolution operation, so it is a common target for acceleration and is the focus of this dissertation.

The 3-D convolution ($*$) operates on an input tensor of size $X \times Y \times C$ and a weight tensor of size $R \times S \times C$ to produce an output tensor of size $(X - R + 1) \times (Y - S + 1) \times 1$. In a convolutional layer, the 3-D convolution operation is

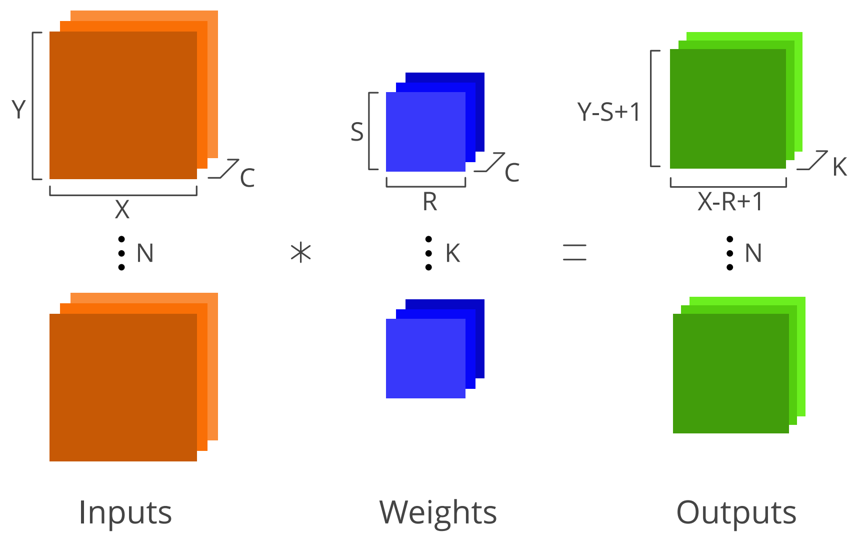


Figure 2.1: The operation performed by a convolutional layer in a deep learning model.

repeated for each of N input tensors and K weight tensors to produce $N \times K$ output tensors. The output tensors are reshaped into N tensors of size $(X - R + 1) \times (Y - S + 1) \times K$. Figure 2.1 depicts the operation of a convolutional layer.

At a high level, a convolutional layer is computed for each of the N input tensors and K weight tensors as follows: (1) the weight tensor is overlaid onto the top left of the input tensor, (2) the tensors are flattened into 1-D vectors and the dot-product is computed to produce a scalar value that is stored as one element of the output tensor, (3) the overlaid weight tensor is shifted by 1—first in the X dimension and then the Y dimension—across the input tensor, and (4) the process repeats until the weight tensor reaches the bottom right of the input tensor. Figure 2.2 shows the software algorithm used to compute a convolutional layer.

```
for n := 0 to N
  for k := 0 to K
    for c := 0 to C
      for y := 0 to Y-S+1
        for x := 0 to X-R+1
          for r := 0 to R
            for s := 0 to S
              Outputs[n][k][y][x] +=
                Inputs[n][c][y+s][x+r] * Weights[k][c][s][r]
```

Figure 2.2: The algorithm used to compute a convolutional layer.

2.2 Deep Learning Accelerators

A deep learning accelerator (DLA) is a specialized processor that is designed to efficiently execute DL models. Specifically, DLAs are optimized to process large tensor operations, such as matrix-multiplication and convolution, because these operations comprise the dominant computational elements of a DL model.

A DLA comprises two high-level components: (1) a compute core that performs the tensor operation, and (2) a memory hierarchy that is designed to exploit data reuse opportunities, of which there are especially many in a convolution operation [101].

Figure 2.3 shows, in more detail, the typical components of a DLA [101]. The compute core comprises a spatial array of processing elements (PEs) that perform one or more multiply or multiply-accumulate operations. If necessary, the outputs of the spatial array are accumulated before being stored in the memory hierarchy. The specific organization of the memory hierarchy varies, but for many edge-scale accelerators [10, 11, 60, 73, 78] there is (1) a software-managed scratchpad and (2) an L2 cache that is connected to DRAM.

The convolution operation shown in Figure 2.2 is mapped onto a DLA in the following three steps.

First, two of the seven dimensions (i.e., N, K, C, X, Y, R, S) are selected

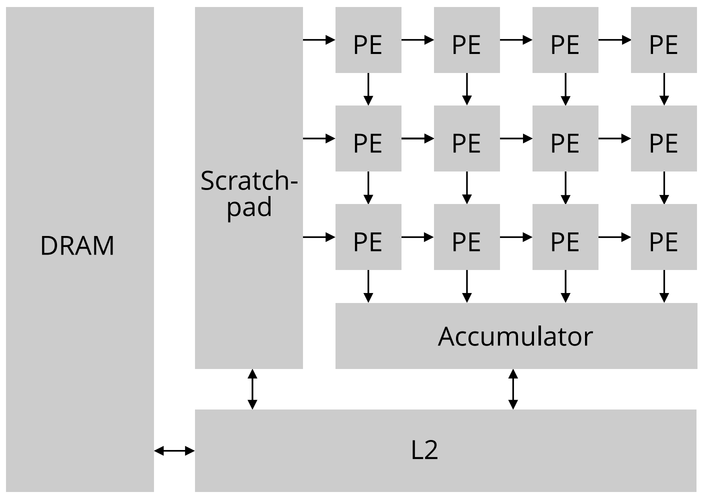


Figure 2.3: The architecture of a typical deep learning accelerator.

to be spatially unrolled—one vertically and one horizontally—across the spatial array. In some cases, this selection is fixed in hardware [47]. If a dimension is too large to be fully unrolled—as is typically the case—, then the data is both spatially and temporally multiplexed across the spatial array.

Second, the convolution operation is broken into pieces, called tiles, such that each tile fits, without overflow, in a targeted level of the memory hierarchy. For the DLA shown in Figure 2.3, which has two levels in the memory hierarchy, two tile sizes are selected: a larger tile size for the L2 and a smaller tile size for the scratchpad. The portion of the convolution that cannot fit in the L2 is temporally multiplexed—i.e., streamed to and from DRAM.

Third, the order of the loops is determined. Loop ordering affects the lifetime of data in the memory hierarchy and has implications on energy consumption. In some cases, the loop order is fixed in hardware [47].

2.3 Bayesian Optimization and Gaussian Processes

Broadly speaking, optimization algorithms aim to find a value or set of values that minimizes or maximizes a function. The task of finding a scalar maximum can be written mathematically as follows.

$$\mathbf{x}^* = \max_{\mathbf{x} \in \mathbb{X}} f(\mathbf{x}) \tag{2.1}$$

We call f the performance function and $\mathbb{X}$ the design space. In this work, we focus on the following subclasses of optimization algorithms.

- Single-objective optimization: When $f : \mathbb{X} \to \mathbb{R}$, f has a single, scalar global optimum. The task of finding $\mathbf{x}^*$ is called single-objective optimization. On the other hand, when $f : \mathbb{X} \to \mathbb{R}^n$ for $n > 1$, the task of finding the pareto-optimal set is called multi-objective optimization.
- Black-box optimization: When ∇f cannot be computed—as is often the case with performance functions—, an optimizer can only find $\mathbf{x}^*$ by evaluating $f(x)$ directly. This is called black-box optimization. On the other hand, white-box optimization moves candidates in the direction of ∇f to find local optima.

Bayesian optimization (BO) is a black-box optimization strategy that is commonly employed when the performance function is expensive to evaluate [5]. A BO framework comprises (1) a *surrogate model*, which predicts the value of the performance function and is cheap to evaluate, and (2) an *acquisition function*, which is used to select the next sample that should be evaluated.

The surrogate model is a learned model that must maintain a reliable measurement of uncertainty for its predictions—i.e., the output is a probability distribution rather than a scalar prediction. The most common type of surrogate model used in BO is a Gaussian process (GP) [28, 91]. At a high level, a GP learns a probabilistic approximation of the performance function by maintaining a Gaussian distribution for each point in the design space. Concretely, a GP takes as input a design, $\mathbf{x}$, and predicts a posterior distribution based on a prior

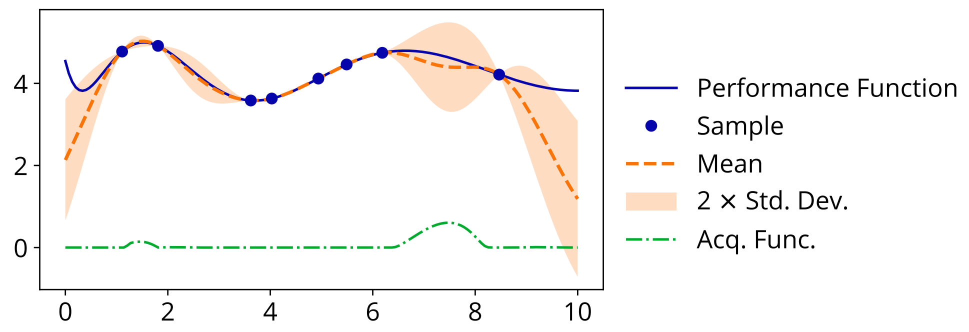


Figure 2.4: A Gaussian process that is modeling a ground truth function that has been sampled at 8 points. The acquisition function—in this case, Expected Improvement—is applied over the Gaussian process and maximized to determine the next sample to evaluate.

distribution over the space of functions comprised of a mean function, $m(\mathbf{x})$, and a covariance function, $k(\mathbf{x}, \mathbf{x}')$. If the covariance for every point in the design space is 0, then the GP exactly matches the performance function.

Figure 2.4 shows a GP (the orange shaded region and the orange dashed line) that is modeling a performance function (the solid line in blue) that has been sampled 8 times. The shaded region represents the uncertainty of the surrogate model at every input.

The acquisition function is a function that is applied over the surrogate model to balance both exploration of uncertain regions and exploitation of the regions that are likely to contain the optimum. A common acquisition function is Expected Improvement (EI) [40], which calculates the change in expected value of the surrogate model if a sample were to be evaluated. The acquisition function is maximized to select the next sample that should be evaluated. Figure 2.4 shows the EI acquisition function (the dot-dash line in green) applied to the GP. The acquisition function is maximized around $x = 7.75$, which is a region of high uncertainty. The other peak of the acquisition function is near $x = 1.75$, which is the maximum of the performance function and is likely to be selected after the point at $x = 7.75$.

Given a surrogate model and acquisition function, the steps that a Bayesian optimizer takes are: (1) select a sample by maximizing the acquisition function, (2) evaluate the sample on the performance function, (3) train the surrogate model with the new evaluation, and (4) repeat the process until either the evaluations converge or a fixed number of trials are complete.

2.4 Selected Machine Learning Techniques

This section presents three machine learning techniques that are fundamental to Starlight, and it assumes the reader has some familiarity with deep learning. If additional background is necessary, we refer the reader to free online resources [77, 79].

2.4.1 Transfer Learning

Transfer learning is a machine learning training technique that re-uses a model for a different task than it was originally trained for. There are many forms of transfer learning [129], but we focus on a straightforward form called hard weight sharing that directly transfers some trained weights from a *source model* to an untrained *target model*.

2.4.2 Variational Autoencoders

An autoencoder [85] is a type of DL model that learns to compress with minimal loss a high-dimensional input into a low-dimensional space called a *latent space*. The architecture of an autoencoder is shown in Figure 2.5. On the left side of the autoencoder, in what is called the encoder network, is a series of fully-connected layers that decrease in size until they reach the target dimension of the latent space. On the right side of the autoencoder, in what is called the decoder network, is a series of fully-connected layers that reverses the encoder network. The autoencoder is trained to minimize the loss between the input

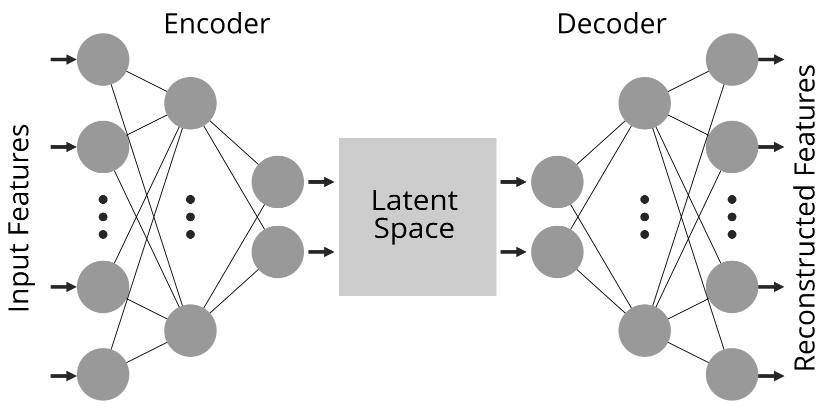


Figure 2.5: The architecture of an autoencoder.

of the encoder network and the output of the decoder network, which should precisely reconstruct the original input. Consequently, the autoencoder learns to encode inputs into unique representations in the low-dimensional latent space.

Autoencoders are susceptible to overfitting [3], and one solution is to inject randomness into the latent representations. Specifically, the last layer of the encoder network is modified to output a Gaussian distribution—as opposed to a scalar value—that non-deterministically encodes an input into the latent space. This type of autoencoder is called a variational autoencoder (VAE) [53]. VAEs are widely accepted be more robust than standard autoencoders.

2.4.3 Deep Kernel Learning

DL models, such as variational autoencoders, excel at learning low-level representations of complex, high-dimensional data [14]. They can be trained using a wide variety of methods [61], but they struggle to provide reliable uncertainty estimates [29], which are necessary for Bayesian optimization. On the other hand, Gaussian processes provide reliable uncertainty estimates, but they do not scale well to high dimensions [4]. Deep kernel learning (DKL) [113] is an emerging technique that combines the best of both worlds. At a high-level, DKL attaches a Gaussian process to the end of the encoder network of an autoencoder.

Recent studies show that DKL pairs well with both transfer learning and Bayesian optimization [2, 26, 63, 114].

Chapter 3: Related Work

Deep learning accelerators (DLAs) have been a hot area of research for the past several years, and thousands of papers have been published on accelerator designs and their design process [19, 123]. In this chapter, we present the prior work that is most relevant to this dissertation. Specifically, we focus on the following two topics in the context of ASIC-based accelerators for GEMM and convolution operations. First is evaluation frameworks, including analytical models and timing simulators. Second is design space exploration (DSE) tools, which automatically explore the values of design parameters to find a configuration that optimizes a performance function.

3.1 Evaluation Frameworks

At every stage of the development process—from architectural exploration to logic design to post-silicon—designers must ensure performance targets and constraints are being met. So, there are a variety of evaluation frameworks that are designed for use at every stage of the development process. In this section, we present these frameworks through the lens of design space exploration (DSE) tools. As such, we categorize frameworks by their wall clock time, which profoundly impacts the capabilities of a DSE tool; fast frameworks, which typically have lower fidelity, enable a DSE tool to explore many configurations, whereas slow frameworks, which typically have higher fidelity, limit the total number of configurations that a DSE tool can feasibly consider.

3.1.1 Fast Evaluation

The most common type of fast framework is an analytical model, which approximates performance to the first-order by abstracting away fine-grained details about execution behavior.

The two most flexible analytical models are MAESTRO [59] and Timeloop [81], which can both model a broad range of DLA architectures and workloads. MAESTRO provides an intuitive interface to specify the modeled software and hardware, but MAESTRO is not as widely applicable as Timeloop, which (1) provides finer control over design parameters, (2) integrates with other well-established performance estimators [75, 118] to provide higher accuracy, and (3) is regularly updated with new features [119].

There are many other analytical models in the literature, but they are typically either simple variations of the roofline model [112] or have limited flexibility [44]. Some examples include: (1) TENET [67], which introduces a more expressive representation for tensor operations than Timeloop but does not provide a means to specify a DLA architecture, (2) SCALE-Sim [88], which has a coarse-grained mode that predicts performance using the geometric properties of an abstract DL workload, and (3) the "Chip Predictor" in the AutoDNNChip framework [122], which estimates the performance of a chain of black-box IP blocks with known delays.

Although analytical models are invaluable tools for hardware design, they have limitations. Notably, analytical models do not track data or memory addresses, so they are oblivious to the nuances of runtime execution [74]. A recent type of fast performance model that can accurately predict the performance of r execution is a data-driven model.[1] Starlight (Chapter 5) is an example of a data-driven model.

Kaufman et al. [51] design the first general-purpose data-driven model, which is a graph neural network that estimates delay by consuming a tensor computation graph and DLA-specific opcodes. Esmaeilzadeh et al. [22] use a

[1]An additional benefit of this approach is that data-driven models are differentiable, which enables DSE tools to use white-box optimizers. We discuss this further in Section 3.2.

data-driven model to predict power, delay, and area, but their model does not account for the specific workload being executed. Ferianc et al. [24] design a Gaussian process that accepts as input (1) a convolutional layer and (2) DLA parameters that can be accessed publicly from a datasheet, and it outputs delay and energy consumption predictions. Other data-driven models are integrated into DSE frameworks, so we discuss them in Section 3.2.2.2 alongside related work in DSE.

3.1.2 Slow Evaluation

The two primary types of slow, pre-silicon evaluation frameworks for ASIC-based accelerators are cycle-accurate timing simulators, which use languages such as C++ or SystemC to model hardware, and RTL simulators, which simulate the gate-level behavior of hardware.

STONNE [74] and SCALE-Sim [88] are both cycle-accurate simulators, but neither models a full system or integrates with popular machine learning frontends, so they have limited usability. Bifrost [99] integrates STONNE with the machine learning compiler framework TVM [9] to add support for standard models. AccTLMSim [52] and SMAUG [120] are also cycle-accurate simulators, but they additionally model a full system.

There are several open-source, parameterizable, ASIC-based DLA implementations available [30, 68, 73, 78, 127] that can be measured with off-the-shelf RTL simulators. However, one RTL simulator that is particularly well-suited for research with ASIC-based DLAs is Firesim [50], which alleviates three key challenges with traditional hardware evaluation: (1) it accelerates RTL simulation using an FPGA, so simulation is orders of magnitude faster than it would be on a CPU, (2) it is built for use with FPGAs hosted by Amazon Web Services, and (3) it supports a highly parameterizable DLA, Gemmini [30].

3.2 DSE Tools

Because the DLA development process is a prime candidate for applying automation, DSE tools that explore the design space of DLAs have gained significant popularity in the past few years. In this section, we first present a brief overview of the facets of DLA development that DSE has been applied to. We then shift focus to the specific type of DSE known as HW/SW co-design that our work addresses.

3.2.1 Overview of Prior DSE Tools

Every level of the deep learning stack—ranging from DL models to place-and-route of DLAs—exposes a huge number of tunable design parameters, so there is ample opportunity to perform DSE.

Much of the work in this area performs DSE for a single level of the deep learning stack. The most notable work includes (1) (hardware-aware) neural architecture search [25, 117, 130], which performs DSE of the DL model architecture and is now standard practice for model design [111], and (2) software optimization (a.k.a. map-space exploration) [9, 34, 37, 46, 81], which performs DSE on the space of software optimizations that can be applied to the convolutional layer loop structure shown in Figure 2.2.

Some prior work ties together single-level DSE tools to build a convenient end-to-end framework that can be viewed as a subset of high-level synthesis [20, 54, 109, 110]. The input to these frameworks is a DL model in a high-level language, and the output is a specialized DLA that is typically implemented on an FPGA or CGRA. This direction of research is orthogonal to our work.

Other work performs DSE simultaneously across multiple levels of the deep learning stack. The two primary classes of this kind of DSE, which is referred to as co-design, are hardware/model co-design and hardware/software (HW/SW) co-design. The former designs a DL model that balances model accuracy and efficient execution [13, 64, 66, 83, 92, 105]. The latter, which is the focus of this dissertation, is discussed in depth in the following section.

3.2.2 HW/SW Co-Design

HW/SW co-design is a type of DSE that finds both (1) optimized layer mappings and (2) an optimized DLA configuration by simultaneously exploring both the software design space (a.k.a. the map-space) and the DLA architecture design space. It has been a popular area of research [102] because it can result in more efficient execution than what single-level DSE tools can achieve [95].

We organize this section by the subclass of optimization algorithm that is employed by the HW/SW co-design tool: either black-box optimization or white-box optimization. Black-box optimizers are typically performed in an online fashion, meaning with each evaluated design the optimizer more accurately learns the shape of the performance function. White-box optimizers are typically performed in an offline fashion using a differentiable proxy model, meaning they are susceptible to finding an optimum that may exist only in the proxy model and not in the performance function. As of now it is inconclusive if one approach is more favorable than the other [58], so both approaches are under active research. Spotlight and Polaris both employ black-box optimization.

3.2.2.1 Black-Box HW/SW Co-Design

The majority of DSE tools use black-box optimizers because traditional evaluation frameworks for DLAs are not differentiable.

Early work tackles the daunting problem of HW/SW co-design by exploring a small co-design space of design parameters. dMazeRunner [15] significantly prunes the software design space so that it is small enough to be explored randomly, and Interstellar [124] only explores in the software space the spatially unrolled dimension, which can be enumerated exhaustively. ZigZag [72] poses a large co-design space, but it uses heuristics to prune the space enough to explore it exhaustively.

A larger co-design space is more challenging to explore, so prior work employs sophisticated optimization algorithms. MAGNet [108] and HASCO [121] both employ off-the-shelf Bayesian optimization (BO) frameworks, but they still use heuristics to prune the co-design space. Apollo [126] and FAST [128] also use an off-the-shelf BO framework. Like Polaris (Chapter 6), the former uses a novel transfer learning technique to warm up the optimizer. Hypermapper [76] and Spotlight [86] (Chapter 4) are custom BO frameworks that consume hand-crafted domain information to guide the optimizer. Spotlight supports a significantly more expressive form of domain information. Other work employs reinforcement learning [45, 121] or genetic algorithm [45, 49], and Vaidya et al. [106] directly solve—i.e., they do not use an optimizer—a re-formulation of the problem.

3.2.2.2 White-Box HW/SW Co-Design

One advantage of data-driven evaluation frameworks is that data-driven models are differentiable, so HW/SW co-design tools can employ white-box optimizers.

VAESA [36], which is one of the earliest white-box HW/SW co-design tools, performs stochastic gradient (SGD) descent on a variational autoencoder that predicts energy and delay. Hong et al. [35] build a DL model that bridges the accuracy gap between analytical models and RTL simulation, and they incorporate the model into a tool called DOSA that performs SGD. Kumar et al. [58] collect an offline dataset of cycle-accurate simulations, use the data to build a model that can predict performance and infeasibilty of a design, and evaluate a multitude of white-box and black-box optimization algorithms to find candidate designs. AIrchitect [87] is a recommendation model that, given a target workload, automatically predicts optimized design parameters. ArchGym [56], although not a HW/SW co-design tool itself, is a modular framework that performs data collection that can be used by white-box optimizers. Esmaeilizadeh et al. [21] create a comprehensive framework for end-to-end DSE that includes a data-driven model to predict post-place-and-route performance, power, and area.

Chapter 4: Spotlight

The goal of a HW/SW co-design tool is to find an optimal design by exploring the co-design space comprising the hardware design space, which comprises architectural design parameters such as buffer sizes and processing element (PE) arrangement, and the software design space, which comprises loop optimization choices such as loop permutations and tiling factors.

Unfortunately, the co-design space exhibits unique characteristics that make it challenging to automatically explore: (1) the co-design space is massive, e.g. a single layer of the ResNet-50 [33] DL model on a spatial array of PEs has $O(10^{18})$ designs, (2) the co-design space is complex, as hardware and software parameters have complex interactions that render large and unpredictable parts of the co-design space infeasible or invalid, and (3) some parameters are ordinal (sortable but discontinuous values) or categorical (a set of arbitrary options), so performance and energy can vary wildly and unpredictably with minor changes to their values, i.e., there are performance cliffs.

To explore this vast co-design space, prior work employs intelligent optimization algorithms such as Bayesian optimization [21, 36, 76, 108, 121, 126, 128] or reinforcement learning [45, 121]. Unfortunately, these techniques largely rely on *off-the-shelf algorithms* which struggle with the complex portions of the design space, particularly with ordinal and categorical parameters [36, 76].

In this chapter we introduce a novel *customized* Bayesian optimization framework, daBO (domain-aware BO), that overcomes the challenges of exploring the HW/SW co-design space. Our key insight is that the optimization algorithm, which conventionally evaluates a large number of samples to learn the shape of the performance function—i.e., the function that maps a point in the co-design space to key metrics such as delay or energy consumption—, can be made more efficient by bootstrapping it with prior knowledge. For example, a domain expert knows that the degree of parallelism, which is derived from the spatially unrolled dimension, the shape of the DL model, and the arrangement of processing elements, is a more accurate predictor of delay than any of the constituent parts alone. In designing daBO, we introduce a flexible method of providing hand-crafted, high-level correlations, i.e. domain information, to the optimization algorithm. As a result, daBO is sample-efficient—i.e., it converges to a solution with fewer evaluations than prior approaches.

Because daBO is sample-efficient, it can be applied to massive HW/SW co-design spaces, enabling it to find—in the same amount of time—designs that are superior to those identified by other optimization techniques. Because it can leverage domain information, daBO can learn complex interactions between parameters. And because daBO can handle ordinal and categorical values, it can consider important design parameters that other techniques struggle with.

We use daBO as the basis for a new automated HW/SW co-design tool called Spotlight, which takes as input a set of DL models and a hardware budget. Spotlight then evaluates designs using the MAESTRO [59] analytical model, and Spotlight produces as output (1) optimized architectural parameters for a programmable DLA and (2) optimized software mappings for each layer of the DL model.

We make the following contributions:

- We present daBO (domain-aware BO), a novel Bayesian optimization framework that effectively deals with the ordinal and categorical design parameters that lead to discontinuities in the design space. In particular, daBO leverages domain information to efficiently learn correlations among categorical design parameters.

- We illustrate the benefits of daBO by presenting Spotlight, an open-source[1] automated HW/SW co-design tool that is built on daBO. We show that for the ResNet-50 DL model, Spotlight produces DLA designs with 44$\times$ lower energy-delay product (EDP) than an Eyeriss-like hand-designed DLA and 135$\times$ lower delay than a design created by ConfuciuX, a state-of-the-art HW/SW co-design tool. For the Transformer DL model, Spotlight achieves 902$\times$ lower EDP than an NVDLA-like hand-designed DLA and 52$\times$ lower delay than a cloud-scale Eyeriss-like DLA.

- We demonstrate that automated HW/SW co-design is critical for designing efficient DLAs. A significant part of Spotlight's benefit comes from the co-design of loop tile sizes with scratchpad sizes—a strategy that is made possible by daBO, which can efficiently explore the design space of tile sizes through the use of domain information.

- We empirically demonstrate that Spotlight exhibits several desirable properties.

  - It is extremely sample efficient. We show that it can effectively explore a co-design space of $O(10^{18})$ design points using just 100 hardware samples and 100 software samples per layer.
  - It can find designs that prior work completely ignores. Specifically, Spotlight considers both loop permutations and loop tiling factors for each dimension, while prior work in automated HW/SW co-design prunes this part of the co-design space.
  - It is highly flexible and can be used in diverse design settings that include both edge-scale and cloud-scale designs: (1) It supports single-model co-design of DLA architectural parameters and software

[1] `https://github.com/chiragsakhuja/spotlight`

mappings, which is useful for FPGA deployment, and (2) it produces programmable DLAs that are able to efficiently execute DL models that they were not co-designed for—a property that is useful for ASIC deployment.

The remainder of this chapter is organized as follows. In Section 4.1 we discuss the specific HW/SW co-design space used in this work and introduce our concept of a feature space. Section 4.2 introduces daBO, and Section 4.3 describes Spotlight, which is evaluated in Section 4.4 before we conclude in Section 4.5.

4.1 Co-Design Space

The co-design space used in this work is the Cartesian product of the hardware and software design space of DLAs, as described in Chapter 2. Specifically, we select a set of parameters that, as prior work [48, 47, 59, 39, 81] has shown, captures a wide variety of DLAs and software optimizations. This co-design space is massive: $O(10^{18})$ for a single layer of ResNet-50 running on a parameterizable DLA.

First, we present the precise values in the co-design space that Spotlight explores, which are categorized as cardinal, ordinal, or categorical. Then, we present the notion of a feature space, which is our technique for reducing the complexity of the co-design space by using domain information.

4.1.1 Parameter Space

The parameter space that Spotlight explores comprises (1) the architectural parameters for DLAs and (2) the full set of loop transformations that can be applied to the 7-level loop to compute a convolutional layer, as shown in Section 2.1.

The hardware design space comprises the following prominent

Parameter	Range
SIMD Lanes	2 to 16
Bandwidth	64 to 256
PEs	128 to 300

(a) Cardinal parameters

Parameter	Range	Stride
Scratchpad Size	64 to 256 KB	8
Register File Size	64 to 256 KB	8
PE Aspect Ratio	Divisors of PE Count	N/A
Tiling Factors†	Divisors of layer shape	N/A

(b) Ordinal parameters

Parameter	Values
Loop Order†	Permutations of loops
Unroll Dimension†	N, K, C, R, S, X, Y

(c) Categorical parameters

†Independent values per level of the memory hierarchy.

Table 4.1: The ranges of design parameters that Spotlight explores.

characteristics of DLAs: processing element (PE) count and arrangement (in a 2-D spatial array); the number of SIMD lanes in each PE; the size of the register files (RFs) that are in each PE; the size of a single global scratchpad; and the bandwidth of the simple interconnect, which supports unicast and multicast. To compare fairly against prior work, we use a fixed 8-bit precision. Table 4.1 shows the ranges of hardware design parameters that Spotlight explores when designing an edge-scale DLA.

The software design space, which is independent for each layer of the DL model, consists of all loop transformations that can be applied to the 7-level loop of a convolution. We consider three loop transformations: loop tiling, loop reordering, and spatial unrolling.

Loop tiling [115] is a common loop optimization that improves data locality by splitting large loops into smaller loops that fit into on-chip caches or scratchpads. Each of the 7 loops in the convolution computation can be independently tiled. Naively, for a DLA with two levels of memory hierarchy, there are $(N \times K \times C \times R \times S \times X \times Y)^2$ options for loop tiling, but many of these options are invalid or require either insertion of edge cases in the loops or padding in the memory footprint. Our design space only considers loop tiling options that perfectly divide the size of the layer.

After loop tiling is applied, the resulting 14 loops can be reordered in any of $(7!)^2$ permutations, and each permutation is a viable option.

Finally, one loop out of each level of loop tiling can be spatially unrolled along each of the vertical and horizontal dimensions of the 2-D spatial array. Spotlight considers all 7^2 options for spatial unrolling.

4.1.2 Cardinal, Ordinal, and Categorical Parameters

Cardinal parameters, which take on integral values within a specified range, are straightforward for optimization algorithms to explore because they tend to exhibit appreciable trends. For example, as on-chip bandwidth is increased, energy consumption and area increase, and delay decreases. Ordinal parameters, which take on ordered values, are more complex to explore—especially if they have inconsistent spacing—, but they can still exhibit appreciable trends. Categorical parameters, however, are problematic for optimization algorithms because they represent arbitrary values that have no correlation among them, so changes in their value have unpredictable implications. Table 4.1 organizes by type each parameter in Spotlight's design space.

4.1.3 Feature Space

The HW/SW co-design space of DLAs exhibits three unique challenges: (1) the co-design space is vast, (2) the co-design space is complex, with interactions among parameters rendering large portions of the space invalid, and

Feature	Calculation
Raw Cardinal Parameters	SIMD Lanes, On-Chip Bandwidth, Total # of PEs, Width of PE Array
Total Amount of On-Chip SRAM	Register File Size + Scratchpad Size
Parallelism Available in Kernel	$R_0 \times S_0$
Degree of Spatial Unrolling	Outer Loop Unrolled Tile Size$\times$ Inner Loop Unroll Tile Size
PE Utilization	$\frac{\text{DRAM Tile Size}}{\text{Outer Loop Unrolled Tile Size}\times\text{Height of PE Array}} \times \frac{\text{Outer Loop Unrolled Tile Size}}{\text{Inner Loop Unrolled Tile Size}\times\text{Width of PE Array}}$
Number of Loop Iterations to Completion	$\lceil\frac{\text{Outer Loop Unrolled Tile Size}}{\text{Height of PE Array}}\rceil \times \lceil\frac{\text{Inner Loop Unrolled Tile Size}}{\text{Width of PE Array}}\rceil$
Approximate Transfers from DRAM	$(X_0/X_2) \times (Y_0/Y_2)\times$ (Width of PE array $+$ Height of PE array)
Size of Commonly Unrolled Dimensions	$2 \times X_0 + 3 \times Y_0 + 5 \times K_0 + 7 \times K_1 + 11 \times K_2$

Table 4.2: Features used as domain information by Spotlight.

(3) changes to the numerous ordinal and categorical parameters can result in erratic changes in behavior of the resulting design. Our technique of injecting domain information into the optimization algorithm overcomes these challenges.

4.1.3.1 Overview

To understand how domain information can improve an optimization algorithm's learning process, consider an example: It is well known that end-to-end delay is directly proportional to PE count and utilization, and given enough sample points, an optimization algorithm can learn this correlation on its own. However, it is sample-efficient for an expert to explicitly highlight this correlation. Thus, domain information can be used (1) to guide the exploration toward profitable regions and away from invalid regions of the co-design space, and (2) to provide information on the behavior of parameters so that changes to these parameters are more predictable.

Typically, an optimization algorithm explores the parameter space directly, but we introduce the notion of a *feature space*, which comprises *features*, which are defined as an arbitrary transformation over the parameter space.

Concretely, let $\mathbb{X}$ be the set of HW/SW co-design parameters. The performance function, f, maps a design in $\mathbb{X}$ to its performance. The feature space is defined as any transformation $T : \mathbb{X} \rightarrow \mathbb{X}'$, where $\mathbb{X}'$ is the feature space and comprises individual features $x'_i : \mathbb{X} \rightarrow \mathbb{R}$. The transformed performance function, f', which is what is learned by Spotlight, maps the performance of a design, $\mathbf{x} \in \mathbb{X}$, as follows: $f'(T(\mathbf{x}))$.

It is easier for an optimization algorithm to find correlations in f' than f. For example, it is unreasonable for an optimization algorithm to learn much useful information about delay from just the spatially unrolled dimension, which is a categorical parameter that takes on 7^2 unrelated values. By contrast, it is much more apparent that there is an inverse relationship between delay and degree-of-parallelism, which is a feature derived from the spatially unrolled dimension, the tiling factors, and the PE arrangement.

4.1.3.2 Feature Selection

The quality of the features determines the quality of the exploration, so thorough feature selection is critical. The selection of relevant and meaningful features is domain-specific, so we propose four general guidelines. First, ensure that categorical parameters are incorporated into one or more features so that it is easier for the optimization algorithm to find correlations among them. Second, encode domain information, i.e. well-known complex interactions among hardware and software parameters, as features. Examples of domain information are: the cost of data transfer among parts of the memory hierarchy and knowledge about the infeasible regions of the co-design space. Third, design features that have linear trends so that the Bayesian optimization framework can quickly learn the simple correlations. Fourth, verify the usefulness of each feature by computing permutation importance [1].

We use these guidelines to brainstorm an initial set of 15 intuitive features including buffer utilization, reuse volume, PE perimeter, and those in Table 4.2. To ensure that the features are of high quality, we measure the correlation between each feature and the performance metrics by (1) computing each feature's value for millions of random HW/SW samples, and (2) visualizing a graph of performance vs individual feature values. We discard any features that do not exhibit a strong correlation. Furthermore, to ensure that removal of a feature does not affect exploration quality, we evaluate our automated HW/SW co-design tool, Spotlight, both with and without these weakly-correlated or uncorrelated features (see Section 4.3).

Table 4.2 shows the final results of our feature selection process, including the equations used to compute each feature. We validate each of these features by ensuring that the correlations learned by the surrogate model are the same as those that we observe with our offline samples. The first features are simply raw cardinal parameters, which our optimization algorithm is already able to correlate well with performance metrics. Next, the total amount of on-chip SRAM is directly correlated with power consumption. The next three features—parallelism available in kernel, degree of parallelism in the spatially unrolled dimension, and PE utilization—measure available parallelism, which is a property of both the hardware and software, and is strongly correlated with delay. Next, some designs can produce many edge cases that lead to a large tail latency, so we incorporate as features an approximation for the number of loop iterations for a layer to completely execute and the number of transfers of the input and kernel matrices from DRAM. Finally, we incorporate commonly unrolled spatial dimensions that are correlated with delay. We observe that each independent parameter—X_0, Y_0, K_0, etc.—has a weak, but notable, correlation with delay because the parameters generally take on fewer than 32 unique values, making it difficult to disambiguate them. For this feature, we spread out the number of unique values by using the prime numbers as the “basis vectors” to compute a linear combination of these parameters.

4.2 Domain-Aware BO

Our novel Bayesian optimization framework utilizes the notion of a feature space to efficiently explore the co-design space.

As an optimizer, Bayesian optimization consists of two major components: (1) a surrogate model that predicts a Bayesian posterior probability distribution over the values of a cost function, and (2) an acquisition function that leverages the posterior distribution to suggest a design point to evaluate.

4.2.1 Surrogate Model

Conventionally, the surrogate model predicts the cost function by learning the characteristics of the parameter space. With daBO, the surrogate model is trained on features instead of parameters. Candidate designs are randomly generated in the parameter space, and daBO transforms them into the feature space before evaluating them on the surrogate model.

As is common practice, daBO uses a Gaussian process (GP) as the surrogate model. Typically, a Matérn [71] or Radial Basis Function (RBF) [7] kernel is employed because these kernels can approximate a wide variety of cost functions [31], but both kernels have complexity $O(N^3)$, and we find that, in the context of Spotlight, they overfit to the evaluated samples. Instead, daBO employs a simple linear kernel, which has $O(N)$ complexity, that takes far fewer samples to accurately model the trends of the cost function. Furthermore, a linear kernel fits well with our feature selection methodology.

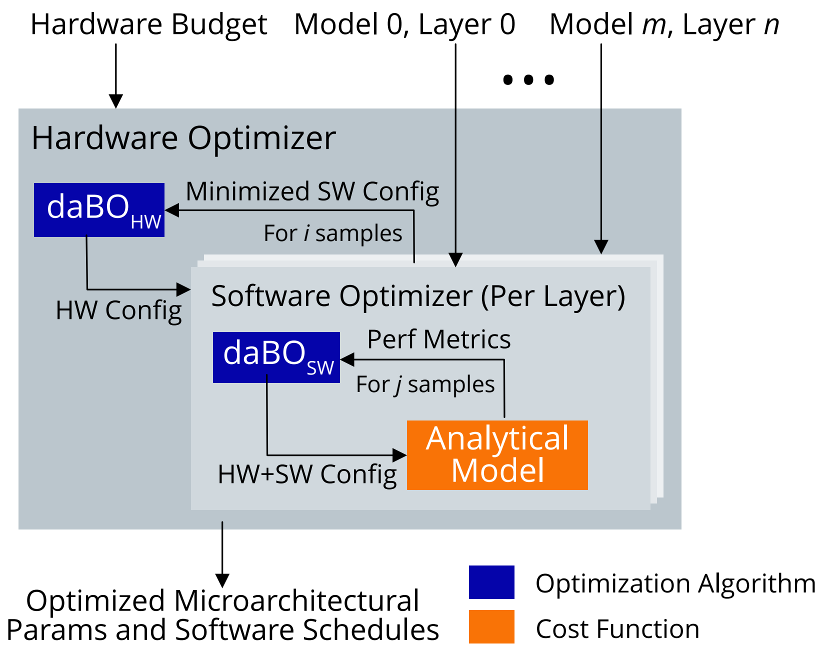


Figure 4.1: Spotlight takes as input a hardware budget and a DL model and performs a nested optimization using our novel Bayesian optimization framework, daBO, to produce optimized microarchitectural parameters and software mappings.

4.2.2 Acquisition Function

The acquisition function selects the next design to evaluate on the cost function. A common choice of acquisition function is Expected Improvement [40], but we find that, empirically, lower confidence bound [98] converges more quickly to a minimum.

4.3 Spotlight

Spotlight is a design space exploration tool that employs multiple instances of daBO to perform automated HW/SW co-design. At a high level, Spotlight accepts as input a hardware budget and a set of layers from one or more DL models; for each input layer, Spotlight produces as output architectural parameters for an optimized DLA, along with optimized software mappings.

Spotlight uses the MAESTRO [59] analytical model to evaluate designs. Spotlight does not perform code generation or hardware synthesis. Figure 4.1 provides an overview of Spotlight.

4.3.1 Layerwise Optimization

It is challenging to optimize multiple layers of a model simultaneously, so Spotlight iteratively optimizes the hardware design and software mappings using a layerwise approach. Independent instances of daBO are used to explore the hardware and software design spaces, so we denote the instances as daBO_{HW} and daBO_{SW}.

We use $\mathbf{x}_h$ and $\mathbf{x}_s$ to denote the set of hardware and software parameters in the parameter space. In Spotlight's layerwise approach, the hardware optimization is first performed by daBO_{HW} with the objective being to minimize $f(\mathbf{x}_h \mid \text{layers})$, which can be the energy-delay product (EDP) or delay of executing the DL model layers on the hardware design. Given the hardware design, Spotlight optimizes the software mapping by applying daBO_{SW} to each layer independently, with the objective being to minimize $f(\mathbf{x}_s \mid \mathbf{x}_h, \text{layer}_j)$, which is defined as the EDP or delay of executing the layer j on the hardware design. The software optimization produces the best software mapping for each layer on the hardware design. The layerwise energies and delays are then added together to compute aggregate EDP or delay, which is fed back to daBO_{HW} to generate the next hardware design. This concludes one iteration of optimization. The iterative optimization between hardware and software repeats for a user-defined number of trials.

4.3.2 Candidate Evaluation

To evaluate the cost of each design, we use MAESTRO [59] to report delay, energy, throughput, power, and area. MAESTRO has been validated against RTL simulation, and our hardware and software design spaces naturally translate into MAESTRO's data-centric loop representation. MAESTRO models primitives, such as interconnects and convolutional layers, that are building blocks for DLAs and DL models.

Spotlight performs single-objective optimization to minimize delay or EDP, which is a common metric for comparing DLAs [46]. From the pareto-optimal frontier, Spotlight selects the configuration that is closest to the inputted area and power budgets without exceeding them.

4.4 Evaluation

We evaluate Spotlight in a variety of settings and against a variety of baselines. Unless otherwise specified, we evaluate Spotlight with 100 hardware samples and, for each hardware design and each layer, 100 software samples.

DL Models We co-design separate DLAs with each of five DL models. Four models—VGG16 [96], ResNet-50 [33], MobileNetV2 [90], and MnasNet [103]—are popular for image processing and span nearly a decade of progress, including one model, MnasNet, that is automatically generated by neural architecture search (NAS). The fifth model is a single Transformer [107], which is a building block for the state-of-the-art natural language processing model, GPT-3 [8].

Hand-Designed DLAs We compare Spotlight's optimized DLA designs against three hand-designed DLAs: NVDLA-like [78], Eyeriss-like [11], and MAERI-like [60][2]. NVDLA and Eyeriss are popular edge-scale DLAs that have been fabricated. Both DLAs suffer from rigid dataflows that cannot always run modern

[2]We refer to the hand-designed DLAs as Eyeriss-like, NVDLA-like, and ShiDianNao-like because the MAESTRO model can only approximate their behavior.

DL models efficiently [12, 47], while MAERI, which is a more recent edge-scale DLA that has not been fabricated, is designed to be highly flexible. For fairness, we evaluate both Spotlight-generated DLAs and the hand-designed DLAs with our layerwise software optimizer, daBO$_{\text{SW}}$ and we scale all DLAs so that they fit in the same area.

HW/SW Co-Design Tools Where possible, we compare Spotlight against two state-of-the-art HW/SW co-design tools that also use the MAESTRO [59] ecosystem: ConfuciuX [45] and HASCO [121]. ConfuciuX optimizes with a combination of reinforcement learning and genetic algorithms, and HASCO optimizes with a combination of Bayesian optimization and reinforcement learning. Both tools explore limited software mappings: ConfuciuX selects one of Eyeriss-like, NVDLA-like, or ShiDianNao-like, and HASCO does not explore software mappings at all. We evaluate ConfuciuX and HASCO with their out-of-the-box configurations. We do not show comparisons against Hypermapper [76], which is a Bayesian optimization framework that consumes a simpler form of domain information, because most runs do not terminate within four days of runtime (far longer than the scale of our evaluated results), and those that do produce designs on par with Eyeriss-like.

DLA Size We generally use Spotlight to generate edge-scale DLAs with the parameters specified in Table 4.1. Additionally, we optimize for a cloud-scale setting and compare against scaled-up hand-designed DLAs. To explore cloud-scale DLAs, the only change to Spotlight is the range of the parameter values that Spotlight explores—Spotlight works out-of-the-box without any other change to configuration.

Performance Metrics Spotlight can minimize either delay or energy-delay product (EDP) under area and power constraints.

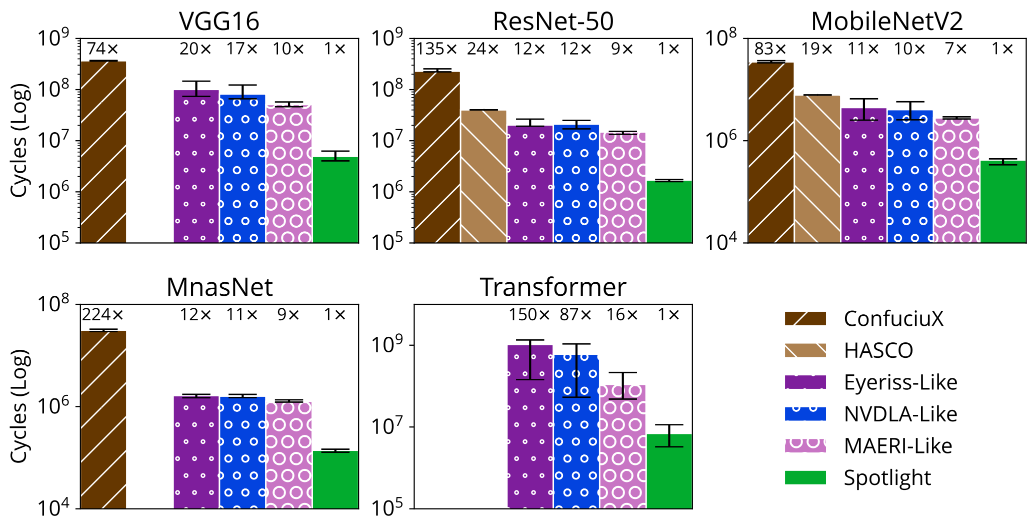


Figure 4.2: Comparison of Spotlight against edge-scale hand-designed DLAs and those designed by state-of-the-art HW/SW co-design tools [45, 121]. The missing data is due to limitations of HASCO—which does not accept VGG16, MnasNet, or Transformer as inputs—and ConfuciuX—which cannot optimize Transformer. Lower is better.

Design Scenarios We present results for two different scenarios, which are described in more detail in their respective sections: single-model co-design (Section 4.4.1) and multi-model co-design (Section 4.4.2).

We conclude the evaluation with a discussion of Spotlight's benefits (Section 4.4.3), a deeper dive into daBO's behavior (Section 4.4.4), and an ablation study (Section 4.4.5).

4.4.1 Single-Model Co-Design

One use case for Spotlight is to co-design a DLA with a full DL model. The generated DLA can be deployed on an FPGA, which can be reconfigured for each new model, or it can be deployed as a highly specialized ASIC, for example, in a low-power IoT device with a long lifetime and static workload.

The key takeaway from this first set of experiments: When co-designing

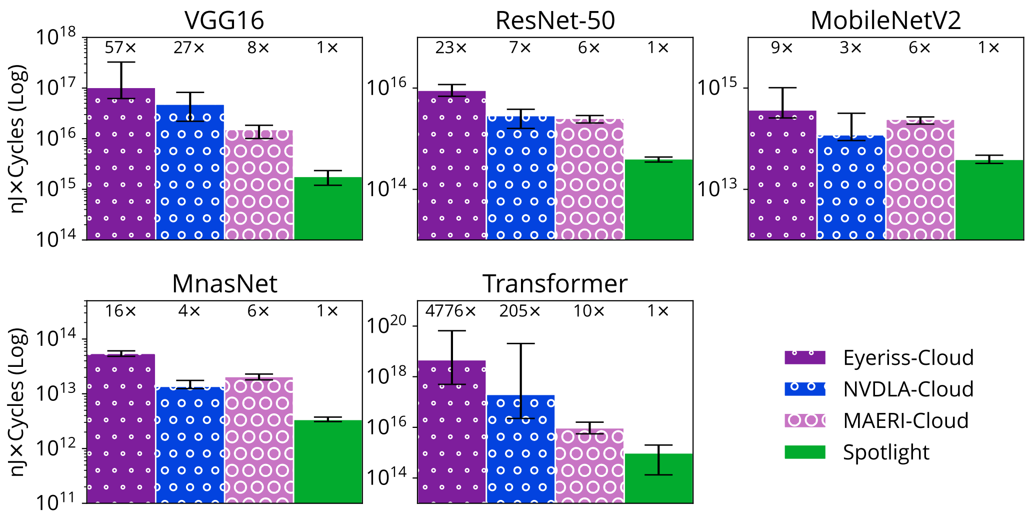
with a single DL model, Spotlight produces designs that achieve significantly lower delay than hand-designed DLAs and those produced by other co-design tools.

Figure 4.2 shows the results when Spotlight co-designs edge-scale DLAs. Each bar represents the median delay of 10 independent trials, and the error bars indicate min/max of the trials. The missing data is due to limitations of HASCO and ConfuciuX, which cannot run all the selected DL models. This figure focuses on delay because HASCO and ConfuciuX cannot minimize energy-delay product (EDP). Notably, the trends when minimizing EDP are identical.

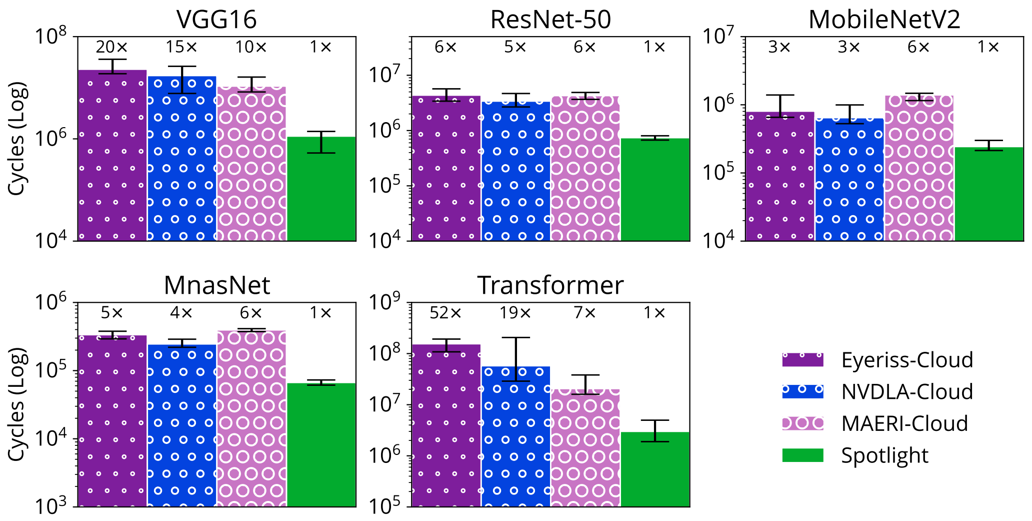
ConfuciuX and HASCO produce inefficient designs primarily because of their limited design spaces—neither aims to co-design loop tile sizes with scratchpad sizes, and we show in Section 4.4.3 that co-design of these parameters is the primary reason that Spotlight performs well. Additionally, ConfuciuX and HASCO explore a severely limited set of software mappings, but we show in Section 4.4.5 that this is not a crippling limitation.

Not surprisingly, of the hand-designed DLAs, MAERI generally achieves the lowest delay, followed by NVDLA and then Eyeriss. MAERI is highly flexible, so it can efficiently execute a wider variety of layer shapes than NVDLA and Eyeriss. NVDLA achieves lower delay than Eyeriss because it spatially unrolls the K and C dimensions, which exhibit higher parallelism in the mid and late layers of every evaluated model than the X and Y dimensions that Eyeriss unrolls. Eyeriss performs especially poorly on Transformer because we convert the GEMM operations that compose Transformer into convolution operations, which results in layer shapes that Eyeriss is not designed to efficiently execute.

Figure 4.3 presents results for cloud-scale DLAs when Spotlight minimizes EDP (top graphs) and delay (bottom graphs). We do not compare against HASCO or ConfuciuX because they do not support cloud-scale DLAs out-of-the-box. For this experiment, the only change we make to Spotlight is to change the range



(a) EDP



(b) Delay

Figure 4.3: Comparison of (a) EDP (nJ$\times$Cycles) and (b) delay (Cycles) of Spotlight against scaled-up versions of hand-designed DLAs. Lower is better.

of parameters; we do not change the feature space or otherwise tune BO for the cloud setting. These results follow the same trends as the edge-scale DLAs.

4.4.2 Multi-Model Co-Design

Spotlight can also be used to co-design one DLA with many DL models. Such a DLA might be deployed as an ASIC, so it must efficiently execute a variety of DL models and remain efficient as new DL models are developed.

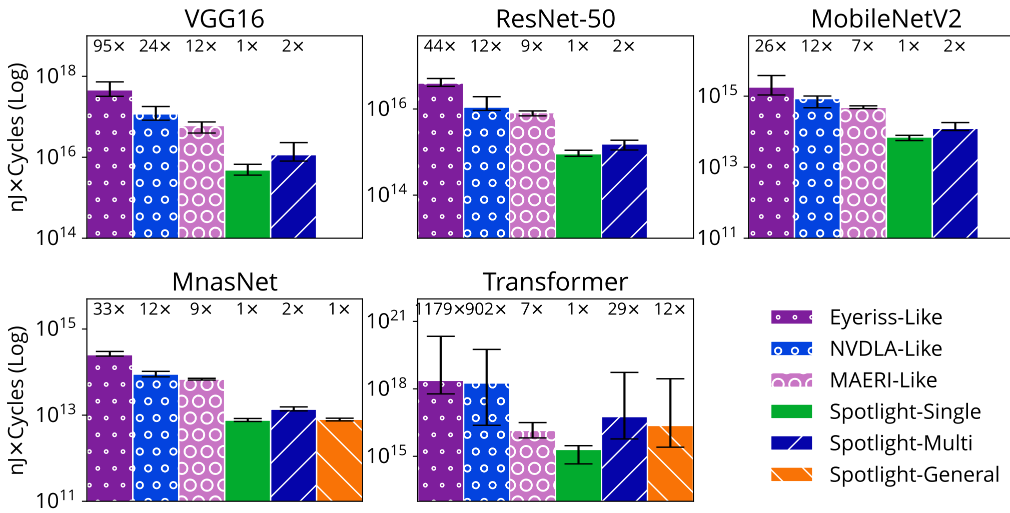
Specifically, we consider two realistic deployment scenarios: (1) We assume that all the DL models are known at design-time, which is common for dedicated IoT DLAs; and (2) we assume that only a limited set of models is known at design-time, and the hardware is expected to generalize to unseen models.

The key takeaway: Spotlight can automatically design programmable DLAs that frequently outperform programmable hand-designed DLAs.

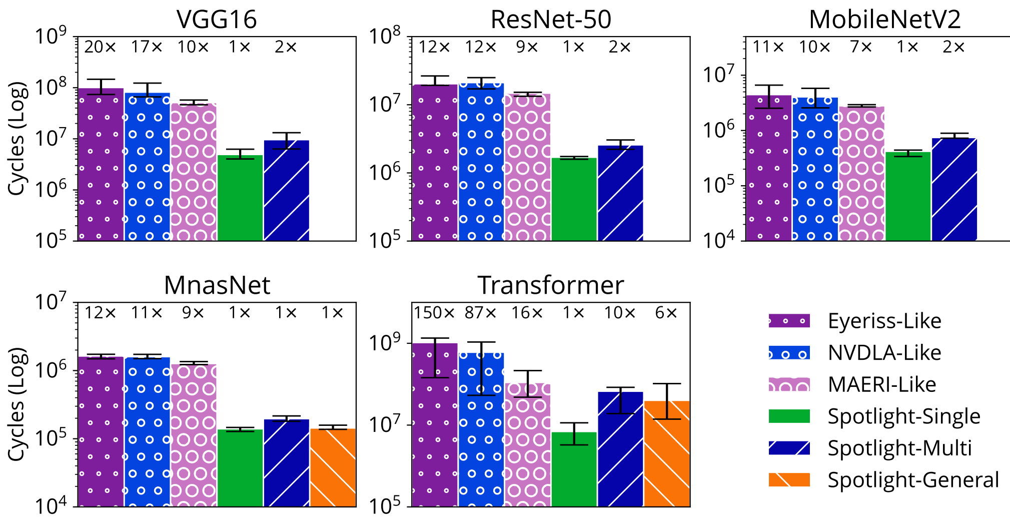
Figure 4.4 shows results for both EDP (top graphs) and delay (bottom graphs), comparing Spotlight's design against hand-designed DLAs that are designed to generalize. Spotlight-Single shows the results of single-model co-design, as described in Section 4.4.1. Spotlight-Multi shows the results of deployment scenario (1), and Spotlight-General shows the results of deployment scenario (2).

To emulate the first scenario, we co-design a DLA with all five DL models as input to Spotlight and then re-run Spotlight's layerwise optimizer (daBO_{SW}) for each model independently on the resulting DLA. Unsurprisingly, Spotlight-Multi has higher EDP and delay than Spotlight-Single because Spotlight-Single finely tunes each DLA for a single model. However, Spotlight-Multi still almost always outperforms each hand-designed DLA, highlighting the benefits of automated design.

To emulate the second scenario, we evaluate whether the hardware co-designed with a subset of DL models—VGG16, ResNet-50, and MobileNetV2—



(a) EDP



(b) Delay

Figure 4.4: The EDP (nJ×Cycles) (a) and delay (Cycles) (b) of the best designs found in the single-model co-design (green), the multi-model co-design (purple), and the generalization (yellow) scenarios. For the generalization scenario, we co-design the DLA with VGG16 ResNet-50 and MobileNetV2, and we evaluate it on MnasNet and Transformer. Thus, only MnasNet and Transformer have yellow bars. Lower is better.

generalizes well to other DL models—MnasNet and Transformer. We co-design a DLA by providing the first three models as input to Spotlight, and then given the resulting DLA we run daBO$_{\mathrm{SW}}$ independently for each of the last two models. We find that Spotlight-General has slightly higher EDP and delay than Spotlight-Single. Rather counterintuitively, we see that Spotlight-General has *lower* delay and EDP than Spotlight-Multi. We conjecture that when simultaneously co-designing for five models, daBO$_{\mathrm{HW}}$ is unable to learn correlations among the complex software space spanning hundreds of unique layers, so the resulting DLA is no longer as efficient for any single model.

4.4.3 Discussion

To understand the benefit of Spotlight, we compare its optimized designs with the behavior of the hand-designed DLAs and HW/SW co-design tools.

The single most significant benefit of using Spotlight is its ability to co-design scratchpad sizes with tile sizes and loop unrolling properties, which leads to improved data locality. For example, given the same area and power budget, when Spotlight's optimized configuration, called Spotlight-Opt, runs ResNet-50, it achieves 26× higher throughput per Joule than Eyeriss, 28× higher than NVDLA, and 8.3× higher than MAERI. The main source of this improvement is greater input and weight reuse, computed as reads per fill, in the scratchpad and register file within each PE. Eyeriss and NVDLA, which have rigid software mappings and fixed hardware, are unable to adjust the spatially unrolled dimension or on-chip memory sizes, so they cannot maintain high on-chip memory utilization for diverse layer shapes. MAERI supports flexible dataflows but still has fixed on-chip memory sizes, so it loses a degree of freedom compared to Spotlight, which finds a better balance between PE count and on-chip memory space than MAERI, so Spotlight-Opt has higher average utilization of both.

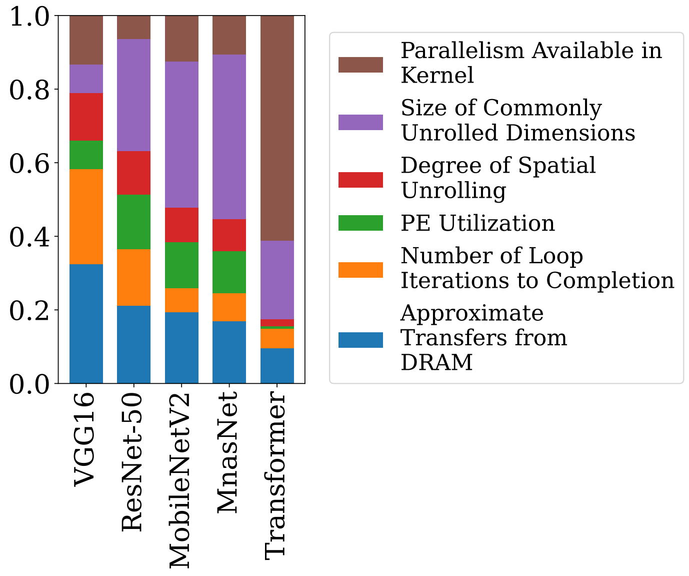


Figure 4.5: The relative importance of each feature in daBO$_{\text{SW}}$.

Qualitatively, the same reasoning explains Spotlight's improvement over HASCO and ConfuciuX. Neither HASCO nor ConfuciuX explores tile sizes nor spatial unroll dimension, so these tools struggle to produce designs that match the efficiency of Spotlight-Opt.

Additionally, Spotlight achieves good results through a series of small wins, which designers often do not consider, during the execution of each layer. For example, we find that Spotlight often produces DLAs with a long and narrow PE array, resulting in two benefits: (1) on the narrow side of the array, network latency is lower and there are fewer unicast operations, and (2) the layer edge cases, which result in low utilization and add tail latency, are smaller and thus have smaller impact on overall runtime. These results (1) illustrate the importance of co-design and (2) the benefits of automated co-design over manual co-design.

4.4.4 Feature Space Analysis

We have demonstrated that Spotlight can efficiently co-design DLAs and software mappings. We now peer into daBO to understand the source of Spotlight's benefits.

Specifically, we rank the importance of each feature. For each instance of daBO_{SW} in single-model configuration, we compute permutation importance [6]: After the GP is trained, we randomly perturb each feature in turn and measure the resulting change in the surrogate model's prediction. Features that cause large changes are considered to be more important.

Figure 4.5 shows the relative importance of each feature. Aside from Transformer, for which "parallelism available in the kernel" is dominant, no single feature is the sole indicator of performance. Parallelism is especially important for the Transformer model because Transformer is dominated by GEMM operations, which when converted to convolution operations result in large and uneven kernel sizes. In general, the most important feature varies.

We repeat this experiment with two modified configurations of Spotlight: (1) with only vanilla parameters instead of features (Spotlight-V) and (2) with the union of all features and raw parameters (Spotlight-A). We find the exact same result: There are typically a few features, which are different for each model, that are the most indicative of performance. We find that Spotlight-A produces DLAs that are on par with Spotlight, and both Spotlight and Spotlight-A produce better DLAs than Spotlight-V. This observation indicates that while good feature selection is still critical, Spotlight is somewhat resilient to the precise feature selection.

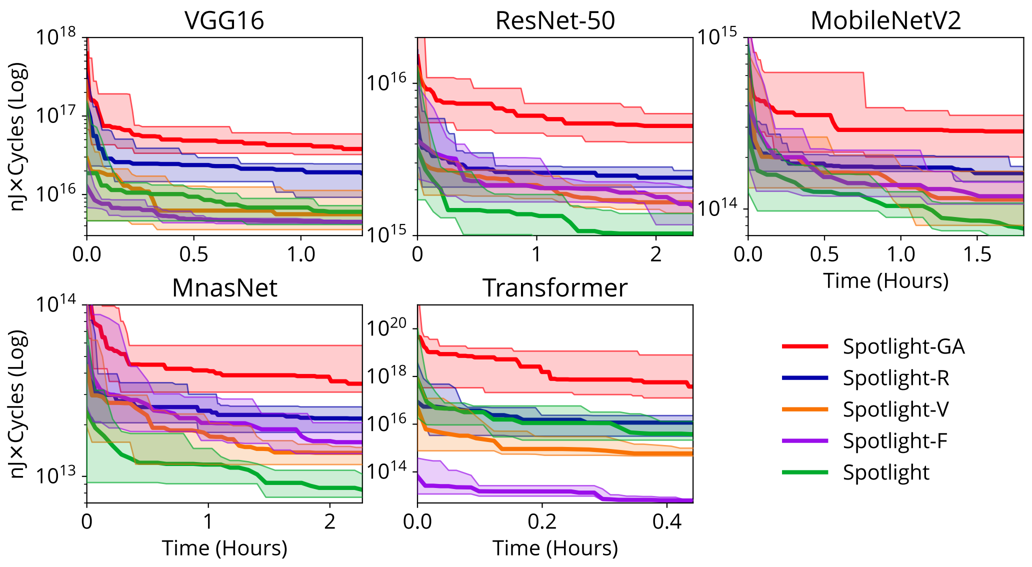
4.4.5 Ablation Study

To isolate the benefits of the daBO framework we compare sample convergence against ConfuciuX and four different optimization algorithms within the Spotlight tool—i.e., we replace daBO$_{\text{HW}}$ and daBO$_{\text{SW}}$ with each of the following five algorithms: genetic algorithm (Spotlight-GA), random search (Spotlight-R), vanilla BO (Spotlight-V), and BO with fixed software mapping options (Spotlight-F). More specifically, Spotlight-V is identical to off-the-shelf BO because it directly explores the parameter space instead of the feature space. Spotlight-F explores the Spotlight feature space, but it only explores the three software mappings supported by ConfuciuX—namely, Eyeriss-like, NVDLA-like, and ShiDianNao-like—and it only explores tiling factors in the K and C dimensions.

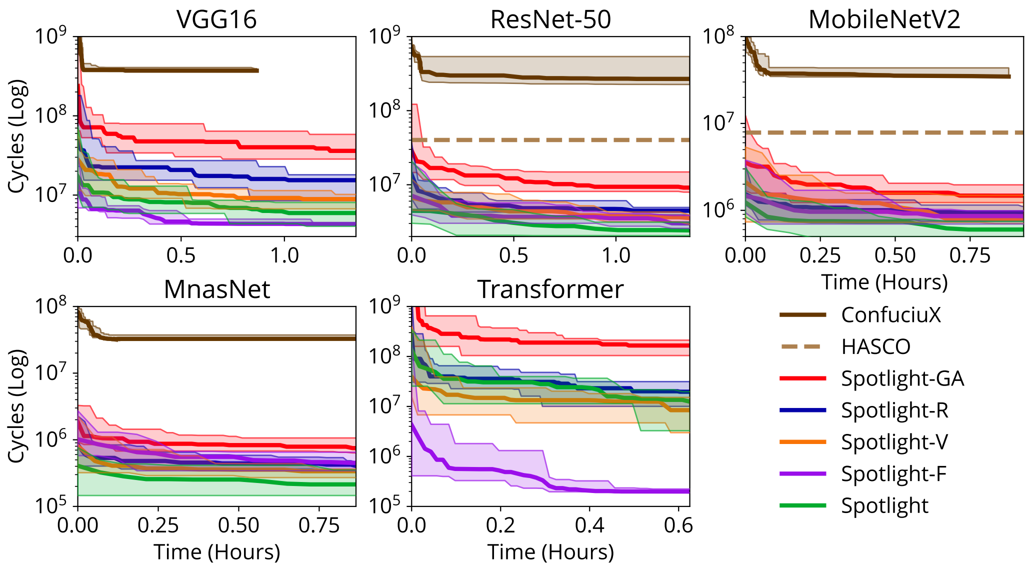
The key takeaway: Bayesian optimization is a strong starting point and is further enhanced by the introduction of the feature space. Moreover, most of the designs selected by Bayesian optimization are superior to the best configuration produced by competing algorithms.

Figure 4.6 shows how each optimization algorithm, including ConfuciuX, converges—as a function of wall clock time—to a minimized EDP and delay when co-designing a single model. The shaded region represents the minimum and maximum of 10 optimization trials, and the solid line represents the median. We are unable to collect per-sample data with HASCO, so we denote with a dashed line the best result of HASCO's 10 trials.

BO consistently achieves lower EDP and delay than random search, genetic algorithm, ConfuciuX, and HASCO. Furthermore, our results suggest that given unlimited runtime, ConfuciuX may never achieve the same quality of solutions that Spotlight can achieve in a few hours. Moreover, both Spotlight and Spotlight-F, which use domain information, outperform Spotlight-V, which does not use domain information, by up to $2\times$ in all cases except for Transformer. For Transformer, we compute permutation importance [1], as described in Section 4.4.4, on the parameter space of Spotlight-V and the feature space of Spotlight, and we find, unexpectedly, that the raw parameters have a larger impact



(a) EDP



(b) Delay

Figure 4.6: The EDP (a) and Delay (b) during single-model co-design for five optimization algorithms: Spotlight, three variations of Spotlight—random search (Spotlight-R), BO with fixed dataflow (Spotlight-F), and vanilla BO (Spotlight-V)—and two state-of-the-art co-design tools. For each layer of each hardware design, Spotlight and variations evaluate 100 sample points in the software design space. The solid line represents the median of 10 trials, and the shaded region represents the minimum and maximum. Lower is better.

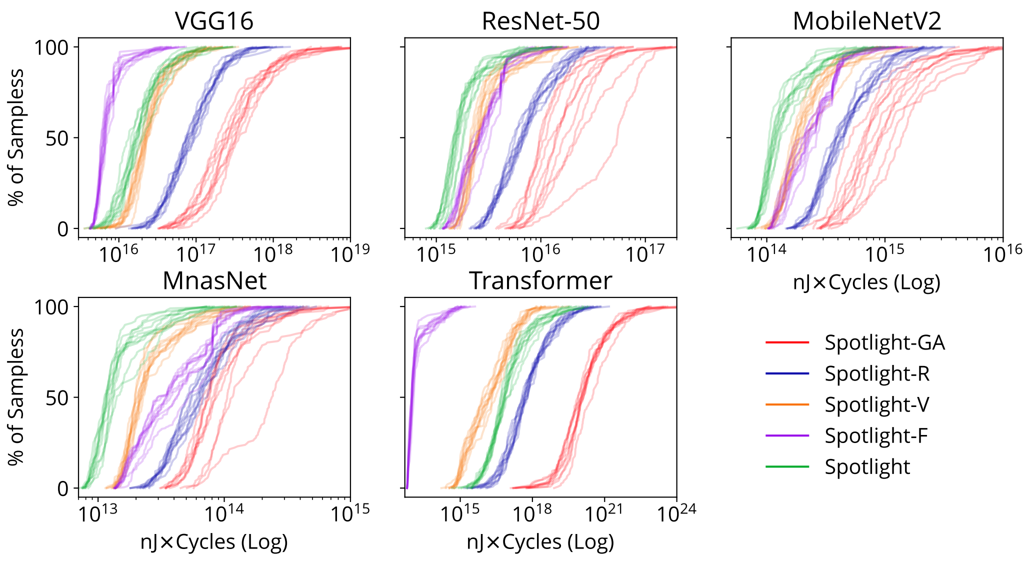
on the surrogate model's prediction than our selected features. This observation explains why Spotlight-V outperforms Spotlight, and it highlights the importance of carefully selecting good for each specific application.

Our results also show that BO explores the co-design space more efficiently than other algorithms. The domain of the X axis of Figure 4.6 is set to the shortest wall clock time of the evaluated algorithms—in most cases, Spotlight-GA. Compared to Spotlight-GA, Spotlight-R evaluates 82% of the total number of samples, and Spotlight evaluates 52% of the total number of samples. Though Spotlight spends more time per-sample than Spotlight-GA and Spotlight-R, the improved sample efficiency of daBO results in superior results within the same wall clock time.

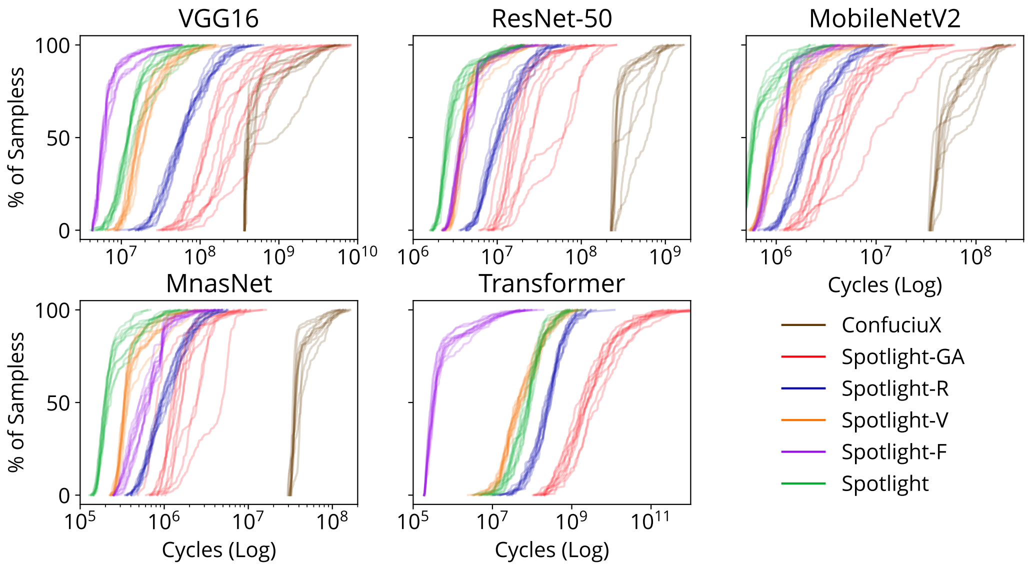
We find that Spotlight-F outperforms Spotlight for VGG16 and Transformer. Eyeriss is designed to be highly efficient when executing VGG16 [11], and indeed we find that when minimizing either EDP and delay, Spotlight-F selects an Eyeriss-like software mapping every time. Transformer is dominated by GEMM operations (converted to convolution), which NVDLA-like and ShiDianNao-like software mappings are able to execute efficiently. Because the software mappings that Spotlight-F explores are already tuned for the layers of VGG and Transformer, Spotlight-F has the advantage of exploring a simple yet high-quality co-design space that can be explored more quickly than the co-design space of Spotlight, so Spotlight-F achieves superior results within the same wall clock time.

To further evaluate the quality of each optimization algorithm, we present Figure 4.7. This figure plots the cumulative distribution function (CDF) of hardware sample points, which shows the percentage of total sample points evaluated that achieve a given EDP or delay. Each line represents 1 of 10 trials.

The CDFs for Spotlight and Spotlight-F are further left than those of the competing algorithms, which indicates that Spotlight does not find just a single



(a) EDP



(b) Delay

Figure 4.7: Cumulative distribution function of hardware samples for each optimization algorithm. Each line represents the results from 1 of 10 trials. Further to the left is better.

good configuration, but it consistently finds designs that outperform the best designs found by competing algorithms.

The CDF for Spotlight-R is Gaussian, while the other optimization algorithms have a steep initial slope, which means that many of the sample points achieve EDP or delay that is similar to the final optimized configuration. Specifically, 81.7% of the hardware samples that Spotlight selects are better than the best results that Spotlight-R finds. So, it is clear that BO is conducting a higher quality optimization.

4.5 Conclusion

In this chapter, we have presented Spotlight, an automated framework for performing hardware/software (HW/SW) co-design of DLAs. We have also presented daBO, our novel Bayesian optimization framework that is critical to Spotlight's success because it incorporates domain information into the automated optimization process. We have empirically demonstrated that Spotlight can produce highly efficient HW/SW co-designs that are orders of magnitude better than competing solutions, including both manually designed DLAs and those designed by state-of-the-art tools.

Philosophically, we observe that prior work [15, 108, 124] manually applies domain information to define dramatically smaller co-design spaces to explore, but because the co-design space is so complex, this manual pruning apparently removes many of the best design points from the design space. By contrast, Spotlight gets great power by embracing a vast co-design space and incorporating the domain information into the automated optimization process, thereby giving Spotlight a mechanism for finding many of the best design points.

Chapter 5: Starlight

One of the key design choices when building a HW/SW co-design tool is the choice of evaluation framework. The co-design tool could evaluate configurations with high fidelity using, e.g., an RTL simulator. However, the long runtime of such techniques would allow the co-design tool to only consider a small number of configurations, severely limiting the tool's ability to explore the design space. Alternatively, the co-design tool could evaluate a large number of configurations using a fast method, e.g., an analytical model. But, such models have low fidelity because they do not capture the nuances of real execution. Figure 5.1 illustrates this tradeoff: As the fidelity of the model increases, the number of configurations that a co-design tool can evaluate dramatically decreases.

From the perspective of a HW/SW co-design tool, this tradeoff implies that either (1) the tool thoroughly explores a low-fidelity representation of the design space, meaning its designs may be sub-optimal when translated to hardware, or (2) the tool sparsely explores a high-fidelity representation of the design space, meaning that it is unlikely to find an optimal design.

We might be tempted to break this tradeoff by using a *data-driven model* that is trained to map a design to its performance as defined by a high-fidelity model. Such a data-driven model could be queried even faster than an analytical model and produce results that approach the accuracy of an RTL simulator. Unfortunately, the training to produce such a data-driven model would itself traditionally require thousands of evaluations by the high-fidelity model [35, 58]—which is difficult to collect even as a one-time investment—presenting the same tradeoff that the data-driven model was intended to break.

In this chapter, we break this tradeoff by using prior knowledge in the form of *transfer learning* to efficiently train a data-driven model. In particular, we train a data-driven model on tens of thousands of low-fidelity samples—which are fast to collect—measured by an analytical model, and then we transfer part of that model to a new model that is then trained on as little as 500 high-fidelity samples—which are slow to collect—measured by an RTL simulator. We are the first to apply transfer learning to transfer prior knowledge from a low-fidelity DLA performance model to a high-fidelity DLA performance model. The resulting model is called Starlight.

We develop Starlight with a specific application in mind: Starlight will be used as a surrogate model in a HW/SW co-design tool that evaluates candidates with a slow, high-fidelity performance model. Typically, a Gaussian process is used as a surrogate model. However, Gaussian processes cannot be used with transfer learning, which is our key approach to making Starlight computationally feasible. So, we employ a recent technique called deep kernel learning [113] that combines the benefits of a traditional neural network and a Gaussian process. Figure 5.1 shows that our approach is faster to query than an analytical model and achieves 99% accuracy when predicting the EDP of a DLA.

We make the following contributions:

- We build a dataset of the delay and energy consumption, as measured by an analytical model, of 2^{16} DLA designs and software mappings. We also build a dataset of the delay, as measured by an RTL simulator, performance model, of 2^{12} DLA designs and software mappings.
- We design a neural network that predicts with 99% accuracy the energy-delay product (EDP) measured by an analytical model.
- We demonstrate that transfer learning can be used to transfer prior knowledge from a low-fidelity performance model to a high-fidelity performance model. The resulting model, called Starlight, is trained on

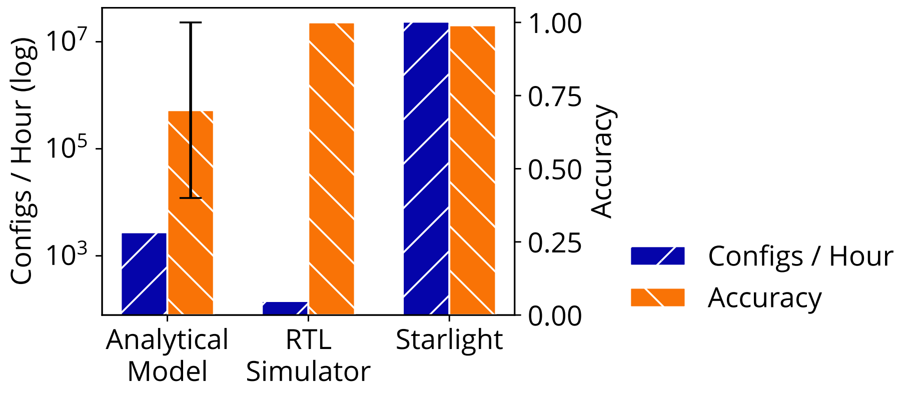


Figure 5.1: Analytical models can be queried thousands of times an hour, but they are inaccurate, whereas an RTL simulator is accurate but slow. Our data-driven model, Starlight, breaks this tradeoff by predicting performance orders of magnitude faster than an analytical model and with 99% accuracy when compared to an RTL simulator. Data is collected from Parashar et al. [81], Karandikar et al. [50], and Mũnoz-Martinez et al. [74].

just 820 RTL simulation evaluations—which can be collected in under one day—and can be queried in milliseconds to predict with 99% accuracy the EDP measured by an RTL simulator.

- We show that deep kernel learning is a robust technique for training an accurate and reliable model that predicts the EDP of a DLA design and software mapping.

The remainder of this chapter is organized as follows. We first motivate Starlight's design in Section 5.1 before presenting details in Section 5.2. We then evaluate the performance and accuracy of Starlight in Section 5.3. Finally, Section 5.5 provides concluding thoughts.

5.1 Motivating Studies

We are motivated to design Starlight because (1) we ideally develop a HW/SW co-design tool that co-designs real hardware, but (2) we observe that

Spotlight is unable to properly explore the co-design space of real hardware because its surrogate model is highly inaccurate, even when trained with tens of thousands of configurations. In this section, we first study the accuracy of Spotlight's surrogate model and then show how transfer learning is a promising approach to improve its accuracy.

5.1.1 Spotlight's Accuracy

To measure the accuracy of Spotlight's surrogate model—a GP with a linear kernel—we first collect a dataset of thousands of HW/SW samples and their respective energy-delay products (EDPs) measured by an analytical model and RTL simulator. We use 90% of the dataset to train the GP in two configurations—with a linear kernel and with a Matérn kernel—using the features described in Table 4.2. We then predict the EDP of the remaining 10% of the dataset. The surrogate model need not predict the absolute EDP value, but it should be able to predict trends so that the acquisition function can accurately select promising configurations [58]. So, we compare the predicted values with the ground truth using the Spearman rank correlation coefficient (ρ) [27], which measures the difference in ordering between vectors such that a score of 1 indicates a strong correlation and -1 indicates an inverse correlation.

Across the test set, ρ is equal to 0.0822 and 0.1127 for the linear and Matérn kernels, respectively. In both cases the correlation is quite low. But, roughly 24% of the top 20% of samples are correctly predicted, which we find is sufficient for the acquisition function to occasionally select a high quality candidate. Hence, Spotlight is able to outperform a state-of-the-art HW/SW co-design tool despite having an inaccurate surrogate model. Though the Matérn kernel achieves a slightly higher correlation than the linear kernel, when we run Spotlight with the Matérn kernel we find no noticeable difference in search quality. This implies that, to notice a difference, we must significantly improve the accuracy of the surrogate

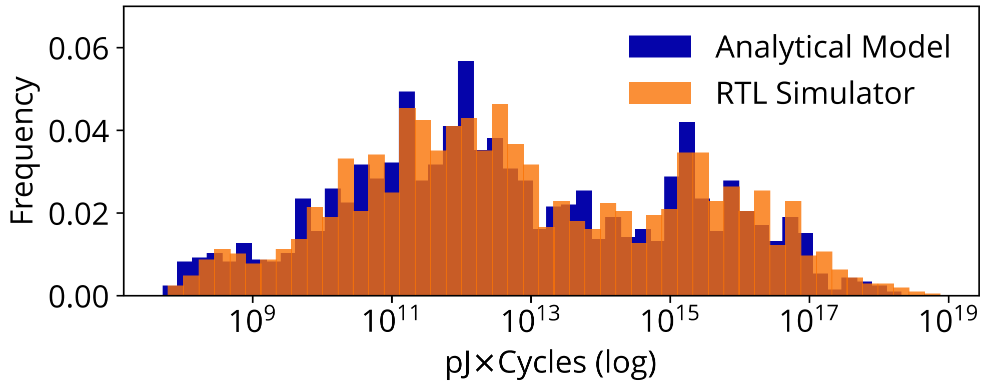


Figure 5.2: The distribution of energy-delay products of HW/SW configurations as measured by an analytical model and RTL simulation. The similarity of the distributions indicates that knowledge can be transferred between models.

model.

5.1.2 Transfer Learning

Recent work shows that data-driven approaches can accurately predict the performance of real hardware [22, 35], but it is time-consuming and costly to build a dataset from real hardware. The deep learning community has faced a similar challenge when performing hyperparameter optimization—i.e., performing DSE on DL model design parameters—and it leans heavily on transfer learning [2, 26, 80] as a solution, so we take the same approach.

Transfer learning can be applied when the knowledge used to predict one task can be transferred to the prediction of a different task. Figure 5.2 shows the distribution of EDPs as measured by an analytical model and an RTL simulator for the same set of DLA designs and software mappings; we see that the two distributions are similar, indicating that knowledge can be transferred between models that predict the two distributions. It is possible that, even though the distributions of EDPs align, the relative ordering of the samples does not. To quell this possibility, we measure ρ, which is equal to 0.99.

Parameter	Values
Spatial Array Dimensions	4x4, 8x8, 16x16, 32x32
Accumulator Size	8 to 256 KB (Step Size: 8)
Scratchpad Size	8 to 256 KB (Step Size: 8)
Loop Order	Permutations of outermost loops
Tiling Factors†	Divisors of layer shape

†Independent values per level of memory hierarchy.

Table 5.1: The ranges of parameter values in the input space of Starlight.

5.2 Model Design

In this section, we first present Starlight's inputs and outputs and the dataset used for training. Then we present Starlight-Low, which is the source model used to transfer knowledge to Starlight. Finally, we present Starlight, which is an accurate performance estimator that predicts the EDP of a DLA as measured by RTL simulation.

5.2.1 Inputs and Outputs

The inputs to Starlight-Low and Starlight are (1) the architectural parameters of a DLA and (2) the software mapping of a single DL layer. The main output of Starlight-Low is a scalar prediction of the energy-delay product (EDP) of the design. Starlight-Low has an additional output, which we describe shortly, that is used for training. The output of Starlight is a Gaussian distribution that provides both a prediction of the EDP and a reliable measurement of uncertainty in the model's prediction.

The precise hardware and software design spaces that Starlight-Low and Starlight are trained on are shown in Table 5.1. In the hardware design space, both models accepts as input the systolic array size and the accumulator and scratchpad sizes. In the software design space, both models accepts as input the loop order and tiling factors. The loop order is encoded as a numerical value from 0 to 6 for each of the seven loops in the convolutional layer loop nest. All inputs are scaled to a unit cube using a min-max scaler.

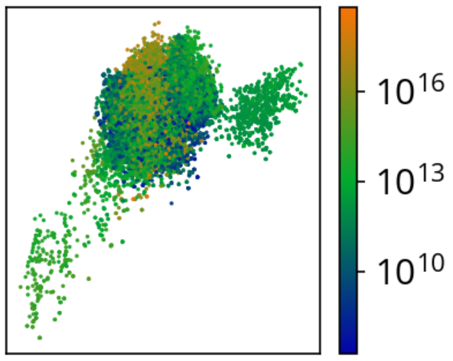
The co-design space that Starlight predicts EDP for encompasses the co-design spaces for a variety of DLAs [48, 47, 59, 39, 81], but it is notably smaller than the co-design space that Spotlight explores. This is because Spotlight performs evaluations using a flexible analytical model that supports a massive design space [81], but Starlight is designed for use with real hardware, which imposes constraints on flexibility. In particular, Starlight is designed to predict EDP for the Gemmini [30] DLA, which exposes the co-design space described above.

5.2.2 Dataset

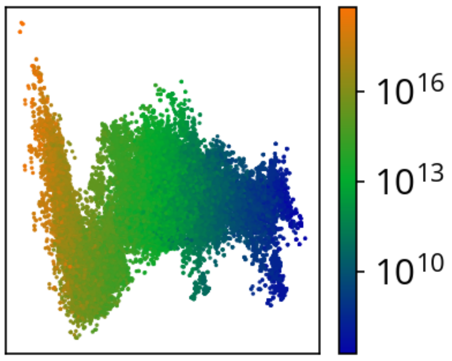
To train Starlight-Low, we collect a dataset of samples from an analytical model called Timeloop [81], and to train Starlight, and we collect a dataset of samples from an RTL simulator called Firesim [50]. The datasets are collected by performing Sobol sampling [97]—a sampling method that prior work shows results in a balanced dataset [21]—on the input space. We collect a total of 2^{16} samples from Timeloop and 2^{12} samples from Firesim.

Both Timeloop and Firesim are measuring the performance of the Gemmini DLA when executing individual layers from one of four DL models, which we describe in more depth in Section 5.3: ResNet-50 [33], RetinaNet [65], BERT [18], and U-Net [84].

A noteworthy limitation of our training data, and consequently of Starlight, is that Firesim does not measure energy consumption, so we always collect energy consumption measurements from Timeloop. For the remainder of this dissertation, when we refer to EDP, we specifically mean the product of energy consumption measurements collected from Timeloop and delay measurements collected from Firesim.



(a) Without predictor network



(b) With predictor network

Figure 5.3: The 2-D latent space of a VAE trained (a) without a predictor network and (b) simultaneously with a predictor network. Each point represents a single DLA design and software mapping that is color-coded by the EDP as measured by an analytical model. The predictor network induces structure, as indicated by the gradient of EDPs, in the latent space.

5.2.3 Starlight-Low

Starlight-Low is a neural network that predicts the EDP of Gemmini as measured by a low-fidelity method, namely Timeloop [81]. Although it achieves high accuracy, our main use for Starlight-Low is to transfer part of the neural network to Starlight.

The model architecture for Starlight-Low is based on a VAE because prior work shows that a VAE can accurately predict the performance of a DLA [36]. Traditionally, a VAE connects an encoder network to a symmetric decoder network and is trained to make the output reproduce the input exactly. We build and train a traditional VAE that encodes inputs into a 2-D latent space, which is shown in Figure 5.3a. Each point represents a single DLA design and software mapping, and the color indicates the EDP as measured by an analytical model. There is no apparent structure to the latent space, which indicates that the encoder is not properly learning the semantics of the inputs. Consequently, the

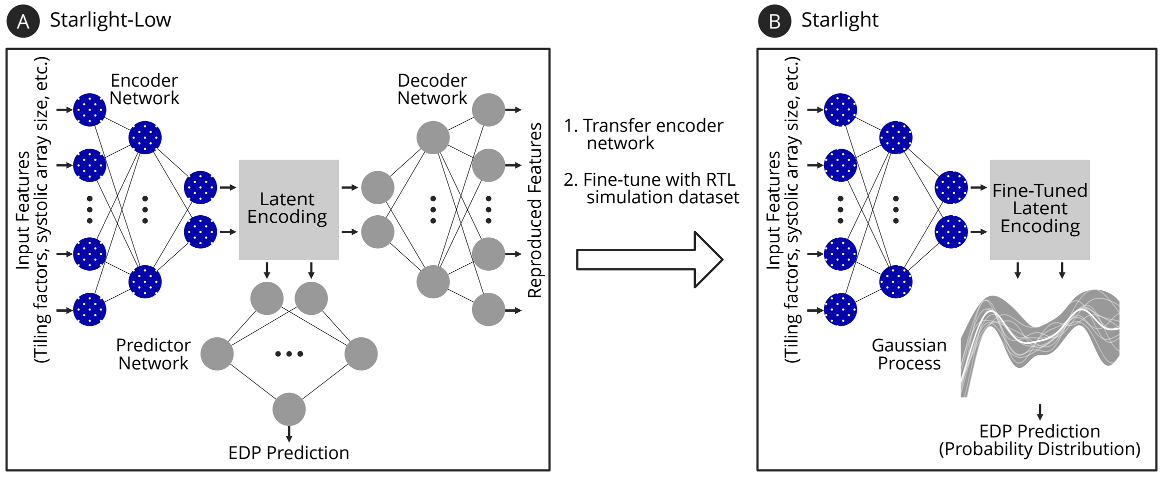

A Starlight-Low
Input Features (Tiling factors, systolic array size, etc.)
Encoder Network
Latent Encoding
Decoder Network
Reproduced Features
Predictor Network
EDP Prediction
1. Transfer encoder network
2. Fine-tune with RTL simulation dataset
B Starlight
Input Features (Tiling factors, systolic array size, etc.)
Fine-Tuned Latent Encoding
Gaussian Process
EDP Prediction (Probability Distribution)

Figure 5.4: Ⓐ Starlight-Low is a neural network that predicts the EDP of a DLA as measured by a low-fidelity method, namely an analytical model. The encoder network (in blue dotted pattern) from Starlight-Low is transferred to Ⓑ Starlight is a neural model based on deep kernel learning that predicts the EDP as measured by a high-fidelity method, namely an RTL simulator.

latent space cannot reliably be used to make EDP predictions.

Prior work [32, 36] solves this problem by simultaneously training a predictor network alongside the encoder and decoder networks. Figure 5.4 Ⓐ shows the model architecture of Starlight-Low, which leverages this technique. The inputs are encoded into the latent space and then fed to two outputs: the predictor network, which predicts the EDP of the configuration, and the decoder network, which reproduces the inputs to ensure significant information is not lost in the latent space. Figure 5.3b shows that the predictor network induces structure, as indicated by the smooth gradient of EDPs, in the latent space.

The final architecture of Starlight-Low is precisely as follows. The encoder network comprises fully connected layers of sizes 40, 24, 12, and 2, and the decoder network is a mirror image. The predictor network comprises fully connected layers of sizes 2, 64, 256, 256, 64, and 1. In all cases, layers are fed through a ReLU activation function.

Starlight-Low is trained to minimize (1) the mean squared error between the predicted EDP and ground truth EDP, (2) the mean squared error between the reproduced inputs and actual inputs, and (3) the Kullback-Leibler divergence [57], which is a measure of the difference between probability distributions, between the output of the encoder and a unit multivariate Gaussian distribution.

5.2.4 Starlight

Starlight is a neural model based on deep kernel learning [113] that predicts the EDP of a DLA as measured by a high-fidelity method, namely an RTL simulator.

To transfer knowledge from Starlight-Low to Starlight, we directly transfer the weights from the encoder network of Starlight-Low. We then fine-tune Starlight using a dataset of EDPs as measured by an RTL simulator. We empirically validate this application of transfer learning in Section **??**.

For Starlight, we make two additional modifications to the architecture of Starlight-Low.

First, we remove the decoder network, which was useful to train Starlight-Low not to lose significant high-dimension information in the latent representations. Because Starlight starts off with a well-behaved latent space, we no longer need to simultaneously train a decoder network.

Second, we replace the predictor network in Starlight-Low with a Gaussian process (GP). This neural model architecture, which ties together a neural network and a GP, is known as deep kernel learning (DKL), and it is an essential modification for Starlight to be used as ultimately intended: as a surrogate model for a Bayesian optimization framework, which requires the surrogate model to produce a reliable measurement of uncertainty. DKL lends two additional benefits: (1) unlike a plain GP, which is the typical surrogate model, DKL supports transfer learning, and (2) for our purposes, DKL trains more reliably than other approaches, as shown in Section 5.4. The GP in Starlight uses a Matérn kernel [71] and gamma prior of predicted EDPs. To train the DKL in Starlight, we maximize the marginal log likelihood of the encoder and GP [113].

The complete architecture of Starlight is shown in Figure 5.4 Ⓑ.

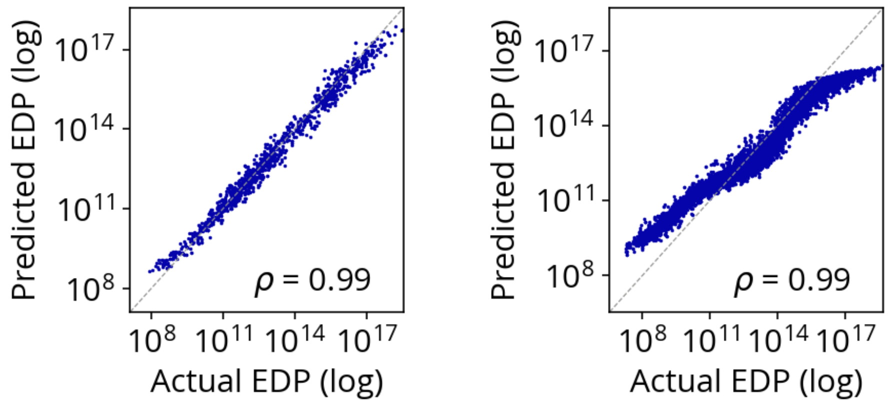
5.3 Evaluation

In this section we evaluate Starlight and Starlight-Low. Unless otherwise specified, we use 80% of the datasets described in Section 5.2.2 for training and the remaining 20% for testing.

Input DL Models We train Starlight and Starlight-Low to predict EDP of executing individual layers from the following set of diverse DL models.

- ResNet-50 [33] is convolutional neural network used for image classification.
- RetinaNet [65] is convolutional neural network that adds on top of ResNet-50 a feature pyramid network, classification head, and regression head. We only evaluate the added layers in RetinaNet.
- U-Net [84] is a convolutional neural network used for biomedical image segmentation.
- BERT [18] is a transformer used for natural language processing.

Performance Metrics We measure the accuracy of Starlight and Starlight-Low using Spearman rank correlation, (ρ) [27], which compares the relative ordering—as opposed to the precise value—of the predicted and ground truth measurements. ρ ranges from -1 to +1, where -1 means the relative orders are exactly reversed and +1 means the relative orders are identical. For Starlight, which is destined for use in a HW/SW co-design tool, a high, positive



(a) Starlight (b) Starlight-Low

Figure 5.5: Accuracy and Spearman rank correlation (ρ) of the actual EDP and the predicted EDP for (a) Starlight and (b) Starlight-Low. Perfect accuracy is $y = x$ and $\rho = 1$.

ρ is sufficient to ensure that the co-design tool properly selects promising candidates [58]. Nonetheless, we measure the more common metric of correlation coefficient and find it to be 97%.

In the remainder of this section, we first present the accuracy of Starlight and Starlight-Low, then we present a study that illustrates the benefits of transfer learning and deep kernel learning.

5.3.1 Accuracy

Figures 5.5a and 5.5b show the accuracy and ρ for Starlight and Starlight-Low, respectively. The X axis is the ground truth energy-delay product (EDP) measured by Firesim for Starlight and by Timeloop for Starlight-Low, and the Y axis is the predicted EDP. Each dot represents a sample. If a sample is predicted with perfect accuracy, it lands on the gray dashed line, $y = x$. Both Starlight and Starlight-Low achieve high accuracy—as is indicated by the average distance across samples from $y = x$—and a ρ of 0.99. Counterintuitively, Starlight achieves

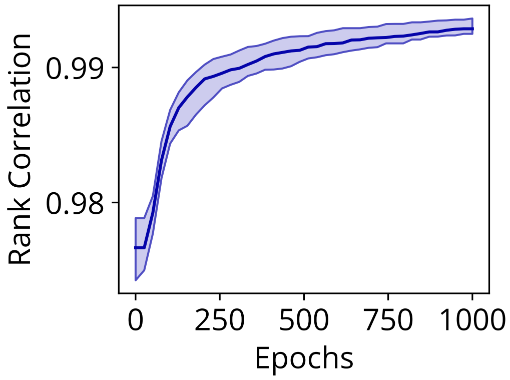


Figure 5.6: Starlight predicts EDP, measured by RTL simulation, with $\rho = 0.99$ after 1000 epochs of training. Across ten independent trials of training, Starlight consistently achieves a median ρ, shown with the solid line, of greater than 0.98 within 100 epochs. Furthermore, the narrow shaded region, which denotes the cumulative minimum and maximum ρ across trials, shows that Starlight is robust to the partition of training data used. Higher is better.

higher accuracy than Starlight-Low on their respective datasets. We show in Section 5.4 that Starlight's high accuracy can be attributed to the use of deep kernel learning.

Figure 5.6 shows ρ for Starlight during training. We perform ten independent trials of training and plot the median, denoted by the central line, and cumulative minimum and maximum, denoted by the shaded region. Starlight achieves $\rho = 0.99$ after 1000 epochs of training (2 minutes of training time on a CPU), and it consistently achieves $\rho \geq 0.98$ after just 100 trials (13 seconds of training time on a CPU). Additionally, Starlight is not sensitive to the partition of training data used, as indicated by the narrowness of the shaded region.

5.4 Transfer Learning and Deep Kernel Learning

To validate our use of both transfer learning and deep kernel learning, we compare Starlight to three other performance estimation approaches. First, we

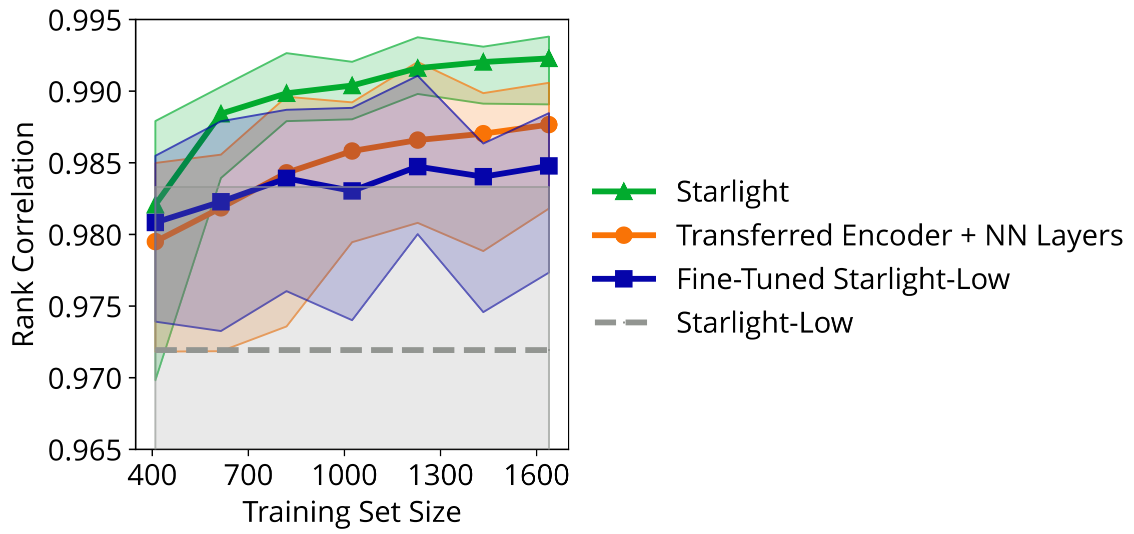


Figure 5.7: ρ versus the Firesim training set size. We evaluate four model architectures: (1) Starlight, (2) a neural network that leverages transfer learning, (3) a simple fine-tuning of Starlight-Low, and (4) Starlight-Low without any fine-tuning. The solid line indicates the median of ten trials, and the shaded region indicates the minimum and maximum. Starlight consistently achieves the highest ρ and is more resilient to the training set size and partition than other models. Higher is better.

compare against a simple fine-tuning of Starlight-Low; we train Starlight-Low as usual, then we fine-tune it on samples from the Firesim dataset. Second, we compare against a model that, first, transfers the encoder from Starlight-Low and second, trains from scratch a neural network predictor network that predicts the EDP of the Firesim dataset. Finally, as a baseline, we compare directly against Starlight-Low.

We repeat this experiment for ten trials and across different training set sizes. Figure 5.7 shows the results. The X axis is the number of samples in the training set, and the Y axis is ρ when each model predicts EDP of the Firesim dataset. The solid line indicates the median of the trials, and the shaded region indicates the minimum and maximum of the trials.

When trained on the full training set, Starlight-Low achieves a median of

$\rho = 0.972$, which means it can accurately predict the EDP of the Firesim dataset, which it was not trained to predict. This is unsurprising given how closely the datasets align in our motivating study in Figure 5.2.

When trained on the full training set, fine-tuned Starlight-Low achieves a median of $\rho = 0.985$, and the neural network trained with transfer learning achieves a median of $\rho = 0.987$. Although the difference is small, the latter consistently achieves slightly higher ρ across ten trials and across training set size. The key difference between these two models is the use of transfer learning, so we conclude that transfer learning is a good approach for this prediction problem.

When trained on the full training set, Starlight achieves the highest ρ of the evaluated models, 0.993. Furthermore, Starlight achieves the smallest variance between trials, indicating that it is robust to the partition of the training set that is used. Starlight also consistently achieves the highest ρ across training set size, indicating that it quickly learns to accurately predict the dataset.

5.5 Conclusion

In this chapter, we have presented Starlight, a data-driven performance estimator that accurately predicts the energy-delay product of DL model execution as measured by RTL simulation. The key techniques that enable Starlight's success are (1) our novel application of transfer learning to transfer prior knowledge from a model trained on a low-fidelity dataset to a model that can predict high-fidelity measurements, and (2) the model architecture, which is based on deep kernel learning.

Chapter 6: Polaris

Most HW/SW co-design tools evaluate candidates using an analytical model [15, 36, 45, 49, 72, 76, 86, 121, 108], but because analytical models do not capture all the nuances of real hardware, the designs produced by these tools may not be optimal if realized in hardware. Consequently, it becomes necessary to incorporate some form of real hardware evaluation in the HW/SW co-design tool. Unfortunately, it is challenging to do so because of two seemingly contradictory constraints: (1) the performance function for hardware is more complex than the performance function for an analytical model [35], so a HW/SW co-design tool must evaluate many samples to accurately learn the shape of the performance function, but (2) hardware evaluation is slow, so the co-design tool must be extremely sample-efficient.

One approach to bypass this dilemma may be to build a proxy model that can be explored in an offline setting [35, 58, 87]. For example, we could explore the input space of Starlight to find a design that it predicts would maximize performance. But even a highly accurate proxy model may deviate from real hardware in regions of the design space, so it is still possible the selected design is not optimal if realized in hardware.

To be certain that a HW/SW co-design tool is accurately exploring the design space of real hardware, the tool must perform hardware evaluations within the optimization loop. So, sample-efficiency is of utmost importance—it is prohibitively expensive for the typical researcher to routinely evaluate hundreds of real hardware samples.

In this chapter, we present Polaris, which is an extremely sample-efficient HW/SW co-design tool built on Starlight that performs hardware evaluation in the optimization loop. Polaris produces designs that reduce average energy-delay product (EDP) by 2.81× over a state-of-the-art co-design tool, DOSA [35], and by 2× over Spotlight. Moreover, it finds these designs with just 48 evaluated samples, which are discovered and measured in under 8 hours.

We make the following contributions:

- We build a HW/SW co-design tool, Polaris, that evaluates with an RTL simulator 48 samples to find DLA designs and software mappings that reduce energy-delay product by up to 2.81× on average over a state-of-the-art co-design tool.
- We empirically demonstrate the benefit of using an online optimization approach over an offline optimization approach.

The remainder of this chapter is organized as follows. We present Polaris in Section 6.1, and we evaluate it in Section 6.2 before providing concluding remarks in Section 6.3.

6.1 Polaris

Polaris is a Bayesian optimization (BO) framework built on Starlight to explore the co-design space of DLA design parameters and software mappings. Specifically, the inputs to Polaris are the shapes of the DL model layers to be optimized, and the outputs are the (1) architectural parameters for a DLA and (2) software mappings that minimize the energy-delay product (EDP) measured by RTL simulation.

In this section, we first describe the Bayesian optimization framework that we build for Polaris, and then we describe the co-design space that is explored. After that, we describe the iterative hardware-software design of Polaris. Finally, we provide the details of the hardware and software optimizers.

Parameter	Values
Spatial Array Dimensions	4x4, 8x8, 16x16, 32x32
Accumulator Size	8 to 256 KB (Step Size: 8)
Scratchpad Size	8 to 256 KB (Step Size: 8)
Loop Order	Permutations of outermost loops
Tiling Factors[†]	Divisors of layer shape

[†]Independent values per level of memory hierarchy.

Table 6.1: The ranges of design parameters that Polaris explores.

6.1.1 Bayesian Optimization Framework

BO frameworks comprise two components: (1) a surrogate model that learns the shape of the performance function, and (2) an acquisition function that selects the next candidate to evaluate.

For Polaris, we use Starlight as the surrogate model. We demonstrate in Chapter 5 that Starlight incorporates prior knowledge to accurately predict the shape of the performance function measured by RTL simulation. Furthermore, Starlight provides the reliable measure of uncertainty that is required of a surrogate model for BO.

A common choice of acquisition function is Expected Improvement [40], but we find that, empirically, upper confidence bound [98] converges more quickly to a minimum EDP, as it did with Spotlight. We demonstrate this behavior in Section 6.2.2.

6.1.2 Co-Design Space

The co-design space used in this work is the same as the co-design space that Starlight is trained to make predictions on—i.e., the co-design space exposed by the paramaterizable Gemmini DLA [30]. We reproduce in Table 6.1 the precise hardware and software design spaces presented in Chapter 5.

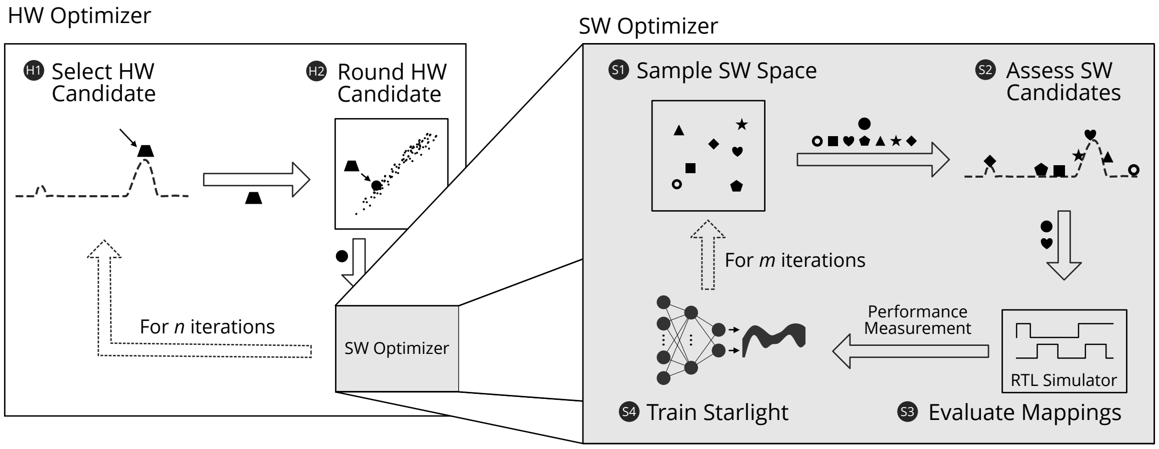


Figure 6.1: Polaris is a HW/SW co-design tool that takes as input DL model layer shapes, and it outputs an optimized DLA design and software mappings. The optimizer is split into an outer loop to optimize hardware and an inner loop to optimize software. At a high level, Polaris operates as follows. A hardware candidate is selected. For each layer of the model, the space of software mappings is sampled and the most promising candidate is determined. The performance of executing the mapping on the DLA is measured using an RTL simulator, and the surrogate model, Starlight, is trained. The software optimizer repeats for *m* iterations, and the hardware optimizer repeats for *n* iterations.

6.1.3 Iterative Hardware-Software Design

It is challenging for a DSE tool to simultaneously co-design both the hardware design space and software mappings for all layers of a model because the co-design space is enormous: It is the Cartesian product of all hardware and software design parameters, e.g., $O(10^{140})$ for ResNet-50, which is a neural network used for image classification. Thus, similar to prior work [36, 66, 86, 108, 121, 128], Polaris is built using an iterative approach; it first selects a hardware candidate, then it optimizes each layer individually to find a software mapping that minimizes the EDP of running that layer on the selected hardware candidate. Figure 6.1 shows an overview of our approach.

At first glance this approach seems straightforward. However, it is challenging to optimize EDP when using a layerwise software optimizer. EDP

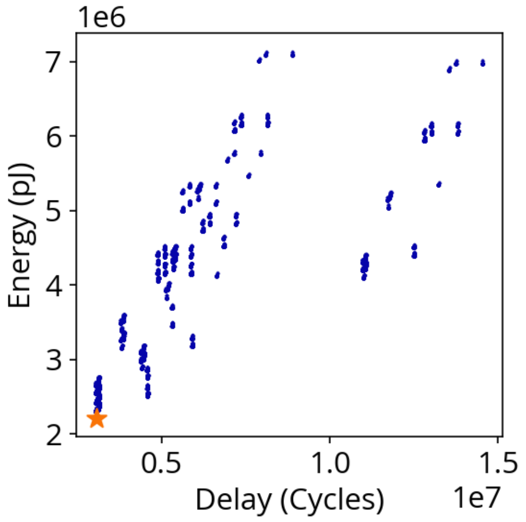


Figure 6.2: The layerwise software optimizer is run for 7 iterations across all layers of BERT. Each point represents BERT's energy consumption and delay for one combination of layers. The starred point, which is the global minimum EDP, is correctly identified even though the layerwise optimizer only optimizes EDP for a single layer at a time.

is typically computed as the product-of-sums across all layers in a model. But, a layerwise software optimizer computes EDP as the product of energy consumption and delay of a single layer, and the individual products are summed to compute the EDP of the full model—i.e., it is computing the sum-of-products. Consequently, Polaris is not minimizing the typical product-of-sums EDP measurement.

To verify that Polaris still finds designs that minimize the product-of-sums EDP measurement, we perform the following experiment. We run the layerwise software optimizer for 7 iterations across all 5 layers of BERT and compute the product-of-sums EDP measurement by computing the energy consumption and delay for all possible combinations of layers—each of the 5 layers has 7 mappings, so we compute energy and delay for all 7^5 possibilities. This is an $O(m^l)$ operation, where m is the number of iterations and l is the number of layers in the model. Figure 6.2 shows all combinations, and the bottom left contains the mappings with the lowest EDP. We also compute the individual EDP for each iteration of each layer. We then find the minimum for each layer across the 7 iterations and select the mappings with the lowest EDP. This is an $O(m)$ operation. The point is marked with the orange star, and it exactly matches the lowest product-of-sums EDP measurement. Thus, we conclude that our layerwise approach safely finds the correct minimum.

6.1.4 Hardware Optimizer

The first step in an iteration of optimization with Polaris is to select a hardware candidate. To do so, Polaris maximizes the upper confidence bound acquisition function as shown in Figure 6.1 H1 . The maximized point (indicated as a trapezoid) is in the continuous input space of Starlight, which means that it is unlikely to be realizable in hardware—e.g., the point may represent a spatial array of size 6.8x6.8. So, it is necessary to convert the point into a realizable hardware candidate. We refer to this step as "rounding". Polaris enumerates all $8 \times 32 \times 32 = 4096$ candidates in the hardware design space and then selects the candidate that minimizes the L1-norm from the maximized point. This step is shown in H2 , and the realizable, rounded HW candidate is marked with a circle.

Given the hardware candidate, Polaris runs the layerwise software optimizer. The hardware optimization process is repeated for n iterations.

6.1.5 Layerwise Software Optimizer

The layerwise software optimizer finds optimized software mappings layer-by-layer given a hardware candidate. The process is as follows for one iteration of the software optimizer for a single layer.

The first step is to select a software candidate. Unlike the hardware design space, the software design space is massive—e.g., there are naively $O(10^{18})$ points for a single layer of ResNet-50. However, we note that, because of constraints imposed by the hardware candidate, only a portion of the design space is realizable. Unfortunately, the realizable design space is still massive, so we enforce two other reasonable constraints: (1) the spatially unrolled dimensions, which are fixed in hardware in Gemmini, should maximize utilization of the hardware, and (2) the tile factors should evenly divide the shape of the layer so that there are no extraneous edge cases that increase the tail latency of running the layer on the hardware candidate. Even after applying these constraints, the software design space can still contain millions of points.

Because we apply non-linear constraints to the software space, Polaris can no longer select a software candidate using the same approach as the hardware candidate selection, which assumes that the dimensions of the design space are independent. Thus, Polaris selects a software candidate as follows: it samples the large, constrained software space using Sobol sampling, and then it assesses each of the candidates with the upper confidence bound acquisition function, selecting the software candidate that maximizes the acquisition function. This process is shown in Figure 6.1 S1 and S2 . The software candidate that is selected is marked with a heart, and the hardware candidate selected by the hardware optimizer is still marked with a circle.

Once the candidate is selected, it is evaluated on an RTL simulator, Firesim [50], as shown in S3 , and Starlight is trained with the new evaluation, as shown in S4 . The software optimization process repeats for m iterations.

6.2 Evaluation

In this section we evaluate Polaris. We first present our methodology.

DL Models We co-design separate DLAs with each of four diverse DL models.

- ResNet-50 [33] is convolutional neural network used for image classification.
- RetinaNet [65] is convolutional neural network that adds on top of ResNet-50 a feature pyramid network, classification head, and regression head. We only evaluate the added layers in RetinaNet.
- U-Net [84] is a convolutional neural network used for biomedical image segmentation.
- BERT [18] is a transformer used for natural language processing.

Hardware Software Co-Design Tools We compare Polaris to three HW/SW co-design tools. First, we compare against DOSA [35], which is a state-of-the-art co-design tool that uses the same evaluation methodology as Polaris. DOSA optimizes a proxy model in an offline setting using stochastic gradient descent, and then it evaluates the final optimized design using Firesim. Second, we compare against an optimizer we call Offline, which performs offline gradient ascent on Starlight's mean function. Third, we compare Polaris against Spotlight. We modify the feature space of Spotlight to accommodate Polaris' co-design space, but otherwise it is unchanged.

We note two caveats with our baselines. First, given its poor performance, Offline does not appear to be functioning correctly. I will investigate this and revise these results as needed. Second, Starlight's surrogate model is trained from scratch for these experiments, so its behavior is nearly equivalent to random search. If we were to warm up Spotlight's surrogate model with the dataset we collect for Starlight, we would fundamentally be changing the behavior of Spotlight, so it would be an unfair comparison.

Design Scenarios We present results for two different design scenarios. First, we perform HW/SW co-design as described previously in this chapter. However,

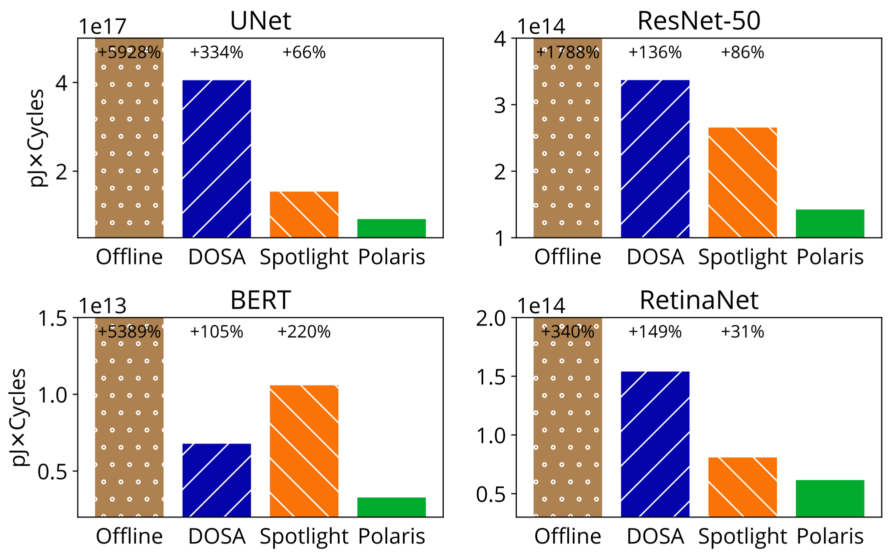


Figure 6.3: We compare the best designs produced by Offline, DOSA, Spotlight, and Polaris when performing HW/SW co-design to minimize EDP. Lower is better. We also present the percent difference of the four baselines when compared to Polaris.

DOSA does not include the spatial array size in its design space, and we find that the spatial array size is a key design parameter when reducing EDP. So, we evaluate Polaris and the other baselines in a second design scenario: software DSE. When performing software DSE, we use the DLA designs found by DOSA.

Number of Iterations When performing HW/SW co-design, we run Polaris and Spotlight for $n = 8$ hardware iterations and $m = 6$ software iterations. When performing software DSE, we run Polaris and Spotlight for $m = 20$ iterations.

In the remainder of this section, we first present the system-level results of Polaris and then present ablation studies to validate our design decisions.

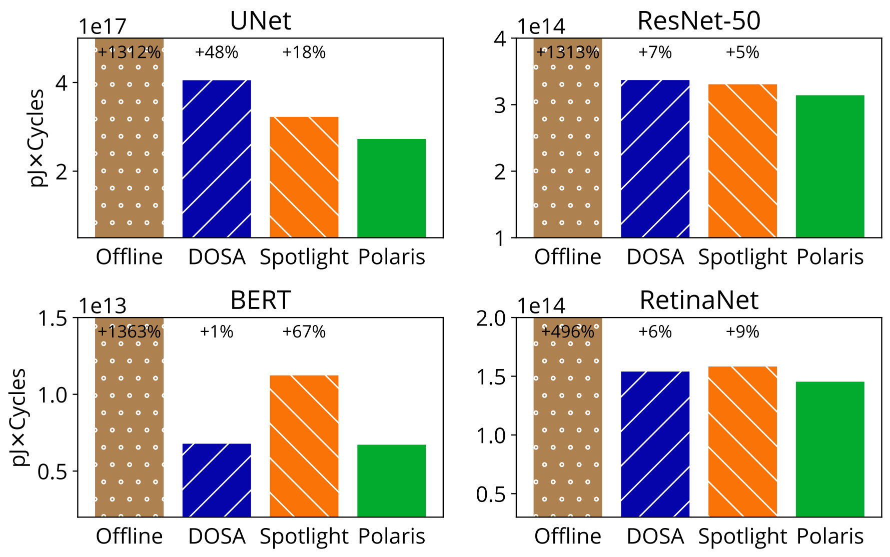


Figure 6.4: We compare the best software mappings produced by Offline, DOSA, Spotlight, and Polaris when performing software DSE to minimize EDP on the DLA design selected by DOSA. Lower is better.

6.2.1 System-Level Results

Figure 6.3 shows the key results of Polaris. We compare the energy-delay product (EDP) of the best designs found by Offline, DOSA, Spotlight, and Polaris. Polaris consistently finds designs with the lowest EDP, and Spotlight typically finds better designs than DOSA. Upon investigation, we find that Polaris and Spotlight always evaluate at least one DLA design with a 32x32 spatial array, which typically has noticeably lower EDP than the DLA designs that DOSA finds, which all use a 16x16 spatial array because DOSA does not incorporate the spatial array size in the search space. However, the spatial array size is not the sole reason for Polaris' success. For BERT, Spotlight explores a design with a 32x32 spatial array, but it is unable to find other commensurate design parameters, so the designs it finds result in higher EDP than the design that DOSA finds. Polaris achieves low EDP because it consistently finds a full set of design parameters that minimize EDP.

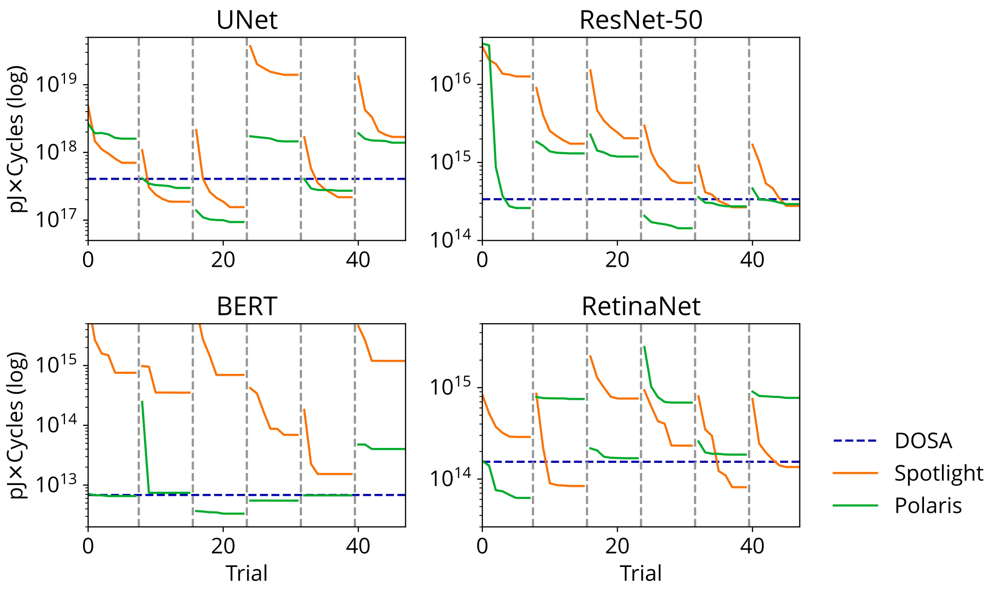


Figure 6.5: The behavior Polaris and Spotlight when performing HW/SW co-design. Each segment demarcated by a gray dashed line is a single hardware candidate, and the solid line indicates the cumulative minimum EDP found by the software optimizer for the hardware candidate. Lower is better.

Next, we constrain Spotlight and Polaris to perform just software DSE on the DLA design that DOSA produces. Figure 6.4 shows that Polaris consistently finds software mappings with the lowest EDP, but its benefit compared to HW/SW co-design is significantly smaller. This indicates that co-design is important; when the optimizer has more degrees of freedom, it is able to find better overall designs.

In both Figure 6.3 and 6.4, we observe that the design produced by Polaris for U-Net results in significantly lower EDP than the design produced by DOSA. U-Net is the largest model that we evaluate, and its software space is massive. On the other hand, BERT is the smallest network that we evaluate, and the designs produced by Polaris result in the smallest reduction in EDP over the design that DOSA produces. So, we hypothesize that Polaris' benefit over prior work increases with the size of the design space it explores.

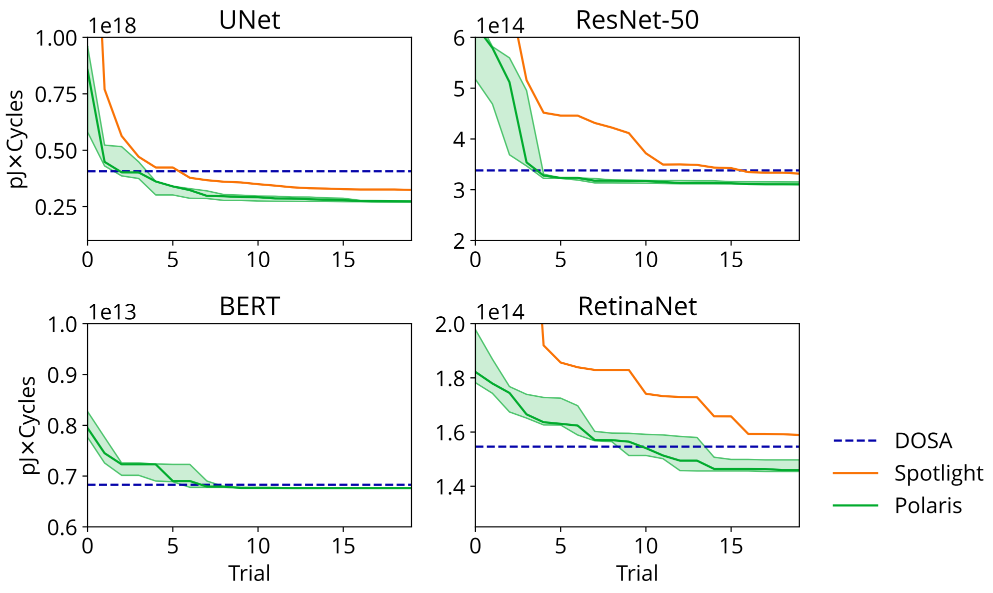


Figure 6.6: The behavior Polaris and Spotlight when performing software DSE. Each segment demarcated by a gray dashed line is an individual hardware iteration, and the solid line indicates the cumulative minimum EDP of the software mappings. Polaris is run for 3 trials. Lower is better.

To more carefully understand the online optimizers, Polaris and Spotlight, we visualize in Figure 6.3 their behavior as they perform HW/SW co-design. The X axis is the overall iteration of the hardware and software optimizers, and the gray dashed lines demarcate the 8 hardware candidates that are evaluated. Within each demarcated segment, we plot the cumulative minimum EDP that the software optimizer finds. Note that the hardware candidates explored by Polaris and Spotlight are unrelated—e.g., in the second segment of RetinaNet, Spotlight appears to find mappings with significantly lower EDP than Polaris, but this is because Spotlight is exploring a DLA design with a 32x32 spatial array, and Polaris is exploring a DLA design with a 4x4 spatial array.

We do not observe any clear trends in the hardware candidates that Polaris or Spotlight selects with each iteration. But, we do observe that the choice of hardware candidate is extremely important. For U-Net, Polaris' software optimizer appears to quickly plateau for each hardware candidate that is selected, and the minimum EDP found for each hardware candidate differs dramatically. For example, it appears as though the first hardware candidate that Polaris selects cannot have a lower EDP than the third hardware candidate that Polaris selects, regardless of how long the software optimizer is run for. We again conclude that the hardware and the software design spaces must be explored in unison to find efficient designs.

For U-Net, ResNet-50, and RetinaNet, we see that with each software iteration both Polaris and Spotlight find mappings that further lower EDP. With Polaris, however, the first iteration of mappings already have low EDP, and the change in cumulative minimum EDP decreases—i.e., there are diminishing returns—after just 2-4 samples. This is even more noticeable for BERT, which has a relatively small design space. The minimum EDP plateaus almost immediately to a value close to the first iteration. We conclude that the Polaris' software optimizer is extremely sample efficient.

To corroborate this claim, we visualize the behavior of Polaris and Spotlight as they perform software DSE. For this experiment, we run Polaris for 3 independent trials. Figure 6.6 shows the results and provide a similar takeaway, but it also shows Polaris' benefit over Spotlight. Polaris very quickly finds good mappings, so the cumulative minimum EDP quickly plateaus. Moreover, Polaris quickly finds good mappings across all 3 trials, and for U-Net, ResNet-50, and BERT every trial finds mappings that result in the same low EDP. On the other hand Spotlight is unable to find mappings for BERT that result in an EDP even close to the EDP of the mappings that Polaris and DOSA find. And, Spotlight's software optimizer does not plateau for RetinaNet even after 15 iterations. We conclude that Polaris reliably finds better mappings than prior work.

6.2.2 Acquisition Function Ablation

6.3 Conclusion

In this section, we have presented Polaris, which is a HW/SW co-design tool, built on top of Starlight, that reliably finds DLA designs and software mappings that minimize EDP of real hardware. Polaris performs hardware evaluation in the optimization loop, so it guarantees that the designs it finds perform well. Moreover, Polaris is extremely sample-efficient. After evaluating just 2-4 software samples, Polaris already finds mappings that result in lower EDP than prior work. And, Polaris quickly finds a full set of DLA design parameters that enable the software optimizer to find good mappings.

With Polaris, we show the importance of performing HW/SW co-design when designing DLAs. The large co-design space is challenging to explore, but, with an intelligent co-design tool, there is ample opportunity to discover efficient designs.

Chapter 7: Conclusion and Future Directions

In this dissertation, we have presented techniques that incorporate prior knowledge to efficiently design deep learning accelerators. We first demonstrated how to incorporate hand-crafted domain information into a Bayesian optimization framework so that a domain expert can guide the optimizer to profitable regions of the design space. We then demonstrated how prior knowledge can be transferred from a low-fidelity model to a high-fidelity model to efficiently train the latter.

We have also presented three open-source tools that can be used to reduce the cost and effort of developing DLAs. Spotlight is a HW/SW co-design tool that can be used in the early stages of the development process, when high-level architectural design decisions are being explored. Polaris is a data-driven performance estimator that can be used in the middle stages of the development process, when RTL simulations are available and it is necessary to estimate the impact of microarchitectural design decisions. And Polaris is a HW/SW co-design tool that can also be used in the middle stages of the development process to automatically tune real hardware parameters.

HW/SW co-design is an important procedure that results in efficient DLA design, which is one of our key tactics to mitigate the burden of the unbridled growth of AI. As a field, HW/SW co-design is constantly evolving to accommodate innovations in AI models and applications. We hope that in the near future it crosses a threshold—like neural architecture search has—that makes it an integral part of the DLA development process. Not only would this improve the productivity of thousands of engineers, it would democratize DLA development. And, with some more work, perhaps it would do for hardware design in general what frameworks such as TensorFlow have done for AI: give any interested party the means to translate their creativity into something tangible.

References

[1] A. Altmann, L. Toloşi, O. Sander, and T. Lengauer, "Permutation Importance: A Corrected Feature Importance Measure," *Bioinformatics*, no. 10, May 2010.

[2] T. Bai, Y. Li, Y. Shen, X. Zhang, W. Zhang, and B. Cui, "Transfer Learning for Bayesian Optimization: A Survey," *arXiv*, Feb. 2023.

[3] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," in *Data Mining and Knowledge Discovery Handbook*, L. Rokach, O. Maimon, and E. Shmueli, Eds., 2023.

[4] M. Binois and N. Wycoff, "A Survey on High-dimensional Gaussian Process Modeling with Application to Bayesian Optimization," *Transactions on Evolutionary Learning and Optimization*, no. 2, Aug. 2022.

[5] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, D. Deng, and M. Lindauer, "Hyperparameter Optimization: Foundations, Algorithms, Best Practices, and Open Challenges," *WIREs Data Mining and Knowledge Discovery*, no. 2, 2023.

[6] L. Breiman, "Random Forests," *Machine Learning*, no. 1, Oct. 2001.

[7] D. Broomhead and D. Lowe, "Multivariable Functional Interpolation and Adaptive Networks," *Complex Systems*, 1988.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess,

J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *arXiv*, Jul. 2020.

[9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "{TVM}: An Automated {End-to-End} Optimizing Compiler for Deep Learning," in *Operating Systems Design and Implementation (OSDI)*, 2018.

[10] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," *Computer Architecture News (CAN)*, no. 1, Feb. 2014.

[11] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *Solid-State Circuits*, no. 1, Jan. 2017.

[12] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *Emerging and Selected Topics in Circuits and Systems*, no. 2, Jun. 2019.

[13] K. Choi, D. Hong, H. Yoon, J. Yu, Y. Kim, and J. Lee, "DANCE: Differentiable Accelerator/Network Co-Exploration," in *Design Automation Conference (DAC)*, Dec. 2021.

[14] A. Damian, J. Lee, and M. Soltanolkotabi, "Neural Networks Can Learn Representations with Gradient Descent," in *Conference on Learning Theory (COLT)*, Jun. 2022.

[15] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "dMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators," *Transactions on Embedded Computing Systems*, no. 5s, Oct. 2019.

[16] A. de Vries, "The Growing Energy Footprint of Artificial Intelligence," *Joule*, no. 10, Oct. 2023.

[17] R. Desislavov, F. Martínez-Plumed, and J. Hernández-Orallo, "Compute and Energy Consumption Trends in Deep Learning Inference," *Sustainable Computing: Informatics and Systems*, Apr. 2023.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv*, May 2019.

[19] P. Dhilleswararao, S. Boppu, M. S. Manikandan, and L. R. Cenkeramaddi, "Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey," *IEEE Access*, 2022.

[20] H. Esmaeilzadeh, S. Ghodrati, J. Gu, S. Guo, A. B. Kahng, J. K. Kim, S. Kinzer, R. Mahapatra, S. D. Manasi, E. Mascarenhas, S. S. Sapatnekar, R. Varadarajan, Z. Wang, H. Xu, B. R. Yatham, and Z. Zeng, "VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis," in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2021.

[21] H. Esmaeilzadeh, S. Ghodrati, A. B. Kahng, J. K. Kim, S. Kinzer, S. Kundu, R. Mahapatra, S. D. Manasi, S. Sapatnekar, Z. Wang, and Z. Zeng, "An Open-Source ML-Based Full-Stack Optimization Framework for Machine Learning Accelerators," *arXiv*, Aug. 2023.

[22] H. Esmaeilzadeh, S. Ghodrati, A. B. Kahng, J. K. Kim, S. Kinzer, S. Kundu, R. Mahapatra, S. D. Manasi, S. S. Sapatnekar, Z. Wang, and Z. Zeng, "Physically Accurate Learning-Based Performance Prediction of Hardware-Accelerated ML Algorithms," in *Workshop on Machine Learning for CAD (MLCAD)*, Sep. 2022.

[23] M. B. Fazi, "Beyond Human: Deep Learning, Explainability and Representation," *Theory, Culture & Society*, no. 7-8, Dec. 2021.

[24] M. Ferianc, H. Fan, D. Manocha, H. Zhou, S. Liu, X. Niu, and W. Luk, "Improving Performance Estimation for Design Space Exploration for Convolutional Neural Network Accelerators," *Electronics*, no. 4, Jan. 2021.

[25] M. Feurer and F. Hutter, *Hyperparameter Optimization*, 2019.

[26] M. Feurer, B. Letham, F. Hutter, and E. Bakshy, "Practical Transfer Learning for Bayesian Optimization," *arXiv*, Oct. 2022.

[27] E. C. Fieller, H. O. Hartley, and E. S. Pearson, "Tests for Rank Correlation Coefficients, I," *Biometrika*, no. 3-4, Dec. 1957.

[28] A. I. Forrester, A. Sóbester, and A. J. Keane, "Multi-fidelity Optimization via Surrogate Modelling," *Royal Society A: Mathematical, Physical and Engineering Sciences*, no. 2088, Oct. 2007.

[29] J. Gawlikowski, C. R. N. Tassi, M. Ali, J. Lee, M. Humt, J. Feng, A. Kruspe, R. Triebel, P. Jung, R. Roscher, M. Shahzad, W. Yang, R. Bamler, and X. X. Zhu, "A Survey of Uncertainty in Deep Neural Networks," *Artificial Intelligence Review*, no. 1, Oct. 2023.

[30] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," in *Design Automation Conference (DAC)*, Dec. 2021.

[31] M. G. Genton, "Classes of Kernels for Machine Learning: A Statistics Perspective," *Machine Learning Research (JMLR)*, Mar. 2002.

[32] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik, "Automatic Chemical Design Using a

Data-Driven Continuous Representation of Molecules," *ACS Central Science*, no. 2, Feb. 2018.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Computer Vision and Pattern Recognition (CVPR)*, 2016.

[34] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind Mappings: Enabling Efficient Algorithm-Accelerator Mapping Space Search," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2021.

[35] C. Hong, Q. Huang, G. Dinh, M. Subedar, and Y. S. Shao, "DOSA: Differentiable Model-Based One-Loop Search for DNN Accelerators," in *Microarchitecture (MICRO)*, Dec. 2023.

[36] Q. Huang, C. Hong, J. Wawrzynek, M. Subedar, and Y. S. Shao, "Learning A Continuous and Reconstructible Latent Space for Hardware Accelerator Design," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, May 2022.

[37] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "CoSA: Scheduling by Constrained Optimization for Spatial Accelerators," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2021.

[38] A. Ivakhnenko and V. Lapa, *Cybernetic Predicting Devices*, 1965.

[39] L. Jia, Z. Luo, L. Lu, and Y. Liang, "Analyzing the Design Space of Spatial Tensor Accelerators on FPGAs," in *International Symposium on VLSI (ISVLSI)*, Jul. 2021.

[40] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient Global Optimization of Expensive Black-Box Functions," *Global Optimization*, no. 4, Dec. 1998.

[41] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, "TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2023.

[42] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2021.

[43] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2017.

[44] L. R. Juracy, A. de Morais Amory, and F. G. Moraes, "A Fast, Accurate, and Comprehensive PPA Estimation of Convolutional Hardware Accelerators," *Transactions on Circuits and Systems I: Regular Papers*, no. 12, Dec. 2022.

[45] S.-C. Kao, G. Jeong, and T. Krishna, “ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning,” in *Microarchitecture (MICRO)*, Oct. 2020.

[46] S.-C. Kao and T. Krishna, “GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm,” in *International Conference on Computer-Aided Design (ICCAD)*, Dec. 2020.

[47] S.-C. Kao, H. Kwon, M. Pellauer, A. Parashar, and T. Krishna, “A Formalism of DNN Accelerator Flexibility,” *Measurement and Analysis of Computing Systems*, no. 2, Jun. 2022.

[48] S.-C. Kao, A. Parashar, P.-A. Tsai, and T. Krishna, “Demystifying Map Space Exploration for NPUs,” in *International Symposium on Workload Characterization (IISWC)*, Nov. 2022.

[49] S.-C. Kao, M. Pellauer, A. Parashar, and T. Krishna, “DiGamma: Domain-aware Genetic Algorithm for HW-Mapping Co-optimization for DNN Accelerators,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2022.

[50] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, “Firesim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud,” in *International Symposium on Computer Architecture (ISCA)*, Jun. 2018.

[51] S. Kaufman, P. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows, “A Learned Performance Model for Tensor Processing Units,” in *Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., 2021.

[52] S. Kim, J. Wang, Y. Seo, S. Lee, Y. Park, S. Park, and C. S. Park, "Transaction-level Model Simulator for Communication-Limited Accelerators," *arXiv*, Jul. 2020.

[53] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," *arXiv*, Dec. 2022.

[54] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Programming Language Design and Implementation (PLDI)*, Jun. 2018.

[55] E. Korneeva, N. Olinder, and W. Strielkowski, "Consumer Attitudes to the Smart Home Technologies and the Internet of Things (IoT)," *Energies*, no. 23, Jan. 2021.

[56] S. Krishnan, A. Yazdanbakhsh, S. Prakash, J. Jabbour, I. Uchendu, S. Ghosh, B. Boroujerdian, D. Richins, D. Tripathy, A. Faust, and V. Janapa Reddi, "ArchGym: An Open-Source Gymnasium for Machine Learning Assisted Architecture Design," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2023.

[57] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *The Annals of Mathematical Statistics*, no. 1, 1951.

[58] A. Kumar, A. Yazdanbakhsh, M. Hashemi, K. Swersky, and S. Levine, "Data-Driven Offline Optimization for Architecting Hardware Accelerators," in *International Conference on Learning Representations (ICLR)*, Oct. 2021.

[59] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings," *IEEE Micro*, no. 3, May 2020.

[60] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2018.

[61] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring Strategies for Training Deep Neural Networks," *Machine Learning Research (JMLR)*, no. 1, 2009.

[62] A. Lavely, "Powering Extreme-Scale HPC with Cerebras Wafer- Scale Accelerators," Cerebras Systems, Inc, Tech. Rep., 2022.

[63] Y. L. Li, T. G. J. Rudner, and A. G. Wilson, "A Study of Bayesian Neural Network Surrogates for Bayesian Optimization," *arXiv*, May 2023.

[64] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W.-m. Hwu, and D. Chen, "EDD: Efficient Differentiable DNN Architecture and Implementation Co-search for Embedded AI Solutions," in *Design Automation Conference (DAC)*, Jul. 2020.

[65] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal Loss for Dense Object Detection," in *International Conference on Computer Vision (ICCV)*, 2017.

[66] Y. Lin, M. Yang, and S. Han, "NAAS: Neural Accelerator Architecture Search," in *Design Automation Conference (DAC)*, Dec. 2021.

[67] L. Lu, N. Guan, Y. Wang, L. Jia, Z. Luo, J. Yin, J. Cong, and Y. Liang, "TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2021.

[68] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "TABLA: A unified template-based framework for accelerating statistical machine learning," in *High-Performance Computer Architecture (HPCA)*, Mar. 2016.

[69] F. Martínez-Plumed, S. Avin, M. Brundage, A. Dafoe, S. Ó. hÉigeartaigh, and J. Hernández-Orallo, "Accounting for the Neglected Dimensions of AI Progress," *arXiv*, Jun. 2018.

[70] N. Maslej, L. Fattorini, E. Brynjolfsson, J. Etchemendy, K. Ligett, T. Lyons, J. Manyika, H. Ngo, V. Parli, Y. Shoham, R. Wald, J. Clark, and R. Perrault, "The AI Index 2023 Annual Report," Institute for Human-Centered AI, Tech. Rep., Apr. 2023.

[71] B. Matérn, *Spatial Variation*, D. Brillinger, S. Fienberg, J. Gani, J. Hartigan, and K. Krickeberg, Eds., 1986.

[72] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators," *Transactions on Computers*, no. 8, Aug. 2021.

[73] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A Hardware–Software Blueprint for Flexible Deep Learning Specialization," *IEEE Micro*, no. 5, Sep. 2019.

[74] F. Muñoz-Martínez, J. L. Abellán, M. E. Acacio, and T. Krishna, "STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators," in *International Symposium on Workload Characterization (IISWC)*, Nov. 2021.

[75] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep., 2009.

[76] L. Nardi, D. Koeplinger, and K. Olukotun, “Practical Design Space Exploration,” in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Oct. 2019.

[77] A. Ng, “The Deep Learning Specialization.”

[78] NVIDIA, “NVIDIA Deep Learning Accelerator,” NVIDIA, Tech. Rep., 2017.

[79] J. Ong, “C++ Neural Network in a Weekend,” Oct. 2020.

[80] S. J. Pan and Q. Yang, “A Survey on Transfer Learning,” *Transactions on Knowledge and Data Engineering*, no. 10, Oct. 2010.

[81] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, “Timeloop: A Systematic Approach to DNN Accelerator Evaluation,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2019.

[82] D. Patterson, J. Gonzalez, U. Hölzle, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. R. So, M. Texier, and J. Dean, “The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink,” *Computer*, no. 7, Jul. 2022.

[83] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, “A Case for Efficient Accelerator Design Space Exploration via Bayesian Optimization,” in *International Symposium on Low Power Electronics and Design (ISLPED)*, Jul. 2017.

[84] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds., 2015.

[85] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in *Explorations in the Microstructure of Cognition*, Jan. 1986.

[86] C. Sakhuja, Z. Shi, and C. Lin, "Leveraging Domain Information for the Efficient Automated Design of Deep Learning Accelerators," in *High-Performance Computer Architecture (HPCA)*, Feb. 2023.

[87] A. Samajdar, J. M. Joseph, and T. Krishna, "AIrchitect: Automating Hardware Architecture and Mapping Optimization," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Apr. 2023.

[88] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Aug. 2020.

[89] SambaNova, "Accelerated Computing with a Reconfigurable Dataflow Architecture," Tech. Rep., 2021.

[90] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *Computer Vision and Pattern Recognition (CVPR)*, 2018.

[91] M. Seeger, "Gaussian Processes for Machine Learning," *International Journal of Neural Systems*, no. 02, Apr. 2004.

[92] L. Sekanina, "Neural Architecture Search and Hardware Accelerator Co-Search: A Survey," *IEEE Access*, 2021.

[93] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos, "Compute Trends Across Three Eras of Machine Learning," in *International Joint Conference on Neural Networks (IJCNN)*, Jul. 2022.

[94] J. Shalf, "The Future of Computing Beyond Moore's Law," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, no. 2166, Jan. 2020.

[95] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture," in *Microarchitecture (MICRO)*, Oct. 2019.

[96] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[97] I. M. Sobol', "On the Distribution of Points in a Cube and the Approximate Evaluation of Integrals," *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, no. 4, 1967.

[98] N. Srinivas, A. Krause, S. Kakade, and M. Seeger, "Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design," in *International Conference on Machine Learning (ICML)*, Jun. 2010.

[99] A. Stjerngren, P. Gibson, and J. Cano, "Bifrost: End-to-End Evaluation and Optimization of Reconfigurable DNN Accelerators," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, May 2022.

[100] E. Strubell, A. Ganesh, and A. McCallum, "Energy and Policy Considerations for Deep Learning in NLP," in *Association for Computational Linguistics (ACL)*, A. Korhonen, D. Traum, and L. Màrquez, Eds., Jul. 2019.

[101] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *IEEE*, no. 12, Dec. 2017.

[102] E.-G. Talbi, “Automated Design of Deep Neural Networks: A Survey and Unified Taxonomy,” *Computing Surveys*, no. 2, Mar. 2021.

[103] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, “MnasNet: Platform-Aware Neural Architecture Search for Mobile,” in *Computer Vision and Pattern Recognition (CVPR)*, 2019.

[104] N. Thompson, K. Greenewald, K. Lee, and G. F. Manso, “The Computational Limits of Deep Learning,” in *Computing Within Limits*, Jun. 2023.

[105] S. Tuli, C.-H. Li, R. Sharma, and N. K. Jha, “CODEBench: A Neural Architecture and Hardware Accelerator Co-Design Framework,” *Transactions on Embedded Computing Systems*, no. 3, Apr. 2023.

[106] M. Vaidya, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, “Comprehensive Accelerator-Dataflow Co-design Optimization for Convolutional Neural Networks,” in *Code Generation and Optimization (CGO)*, Apr. 2022.

[107] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is All You Need,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[108] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany, “MAGNet: A Modular Accelerator Generator for Neural Networks,” in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2019.

[109] J. Wang, L. Guo, and J. Cong, “AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA,” in *Field-Programmable Gate Arrays (FPGA)*, Feb. 2021.

[110] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "DSAGEN: Synthesizing programmable spatial accelerators," in *International Symposium on Computer Architecture (ISCA)*, Sep. 2020.

[111] C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter, "Neural Architecture Search: Insights from 1000 Papers," *arXiv*, Jan. 2023.

[112] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, no. 4, 2009.

[113] A. G. Wilson, Z. Hu, R. Salakhutdinov, and E. P. Xing, "Deep Kernel Learning," in *Artificial Intelligence and Statistics (AISTATS)*, May 2016.

[114] M. Wistuba and J. Grabocka, "Few-Shot Bayesian Optimization with Deep Kernel Surrogates," *arXiv*, Jan. 2021.

[115] M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1982.

[116] D. Wright, C. Igel, G. Samuel, and R. Selvan, "Efficiency is Not Enough: A Critical Perspective of Environmentally Sustainable AI," *arXiv*, Sep. 2023.

[117] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[118] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs," in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2019.

[119] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, “Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling,” in *Microarchitecture (MICRO)*, Oct. 2022.

[120] S. L. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, “SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads,” *Transactions on Architecture and Code Optimization (TACO)*, no. 4, Nov. 2020.

[121] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang, “HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation,” in *International Symposium on Computer Architecture (ISCA)*, Jun. 2021.

[122] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang, C. Li, Z. Guan, D. Chen, and Y. Lin, “AutoDNNchip: An Automated DNN Chip Predictor and Builder for Both FPGAs and ASICs,” in *Field-Programmable Gate Arrays (FPGA)*, Feb. 2020.

[123] R. Xu, S. Ma, Y. Guo, and D. Li, “A Survey of Design and Optimization for Systolic Array Based DNN Accelerators,” *Computing Surveys*, Jun. 2023.

[124] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, “Interstellar: Using Halide’s Scheduling Language to Analyze DNN Accelerators,” in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2020.

[125] A. Yazdanbakhsh, B. Akin, and K. K. Seshadri, “An Evalution of Edge TPU Accelerators for Convolutional Neural Networks,” *arXiv*, 2021.

[126] A. Yazdanbakhsh, C. Angermueller, B. Akin, Y. Zhou, A. Jones, M. Hashemi, K. Swersky, S. Chatterjee, R. Narayanaswami, and J. Laudon, “Apollo: Transferable Architecture Exploration,” *Workshop on ML for Systems*, 2020.

[127] Z. Zeng and S. S. Sapatnekar, "Energy-efficient Hardware Acceleration of Shallow Machine Learning Applications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Apr. 2023.

[128] D. Zhang, S. Huda, E. Songhori, K. Prabhu, Q. Le, A. Goldie, and A. Mirhoseini, "A Full-Stack Search Technique for Domain Optimized Deep Learning Accelerators," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb. 2022.

[129] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A Comprehensive Survey on Transfer Learning," *IEEE*, no. 1, Jan. 2021.

[130] B. Zoph and Q. Le, "Neural Architecture Search with Reinforcement Learning," in *International Conference on Learning Representations (ICLR)*, 2017.