Using Bayesian Optimization for Hardware/Software Co-Design of Neural Accelerators

Zhan Shi, Chirag Sakhuja
The University of Texas at Austin
{zshi17, chirag.sakhuja}@utexas.edu

Milad Hashemi, Kevin Swersky
Google Research
{miladh, kswersky}@google.edu

Calvin Lin
The University of Texas at Austin
lin@cs.utexas.edu

Abstract

The use of deep learning has grown at an exponential rate, giving rise to numerous specialized hardware and software systems for deep learning. Because the design space of deep learning software stacks and hardware accelerators is diverse and vast, prior work considers software optimizations separately from hardware architectures, effectively reducing the search space. Unfortunately, this bifurcated approach means that many profitable design points are never explored. This paper instead casts the problem as hardware/software co-design, with the goal of automatically identifying desirable points in the joint design space. The key to our solution is a new constrained Bayesian optimization framework that avoids invalid solutions by exploiting the highly constrained features of this design space, which are semi-continuous/semi-discrete. We evaluate our optimization framework by applying it to a variety of neural models, improving the energy-delay product by 18% (ResNet) and 40% (DQN) over hand-tuned state-of-the-art systems, as well as demonstrating strong results on other neural network architectures, such as MLPs and Transformers.

1 Introduction

The compute requirements of deep learning are growing at a double exponential rate [11]. There are opportunities to increase DNN efficiency at each layer of the deep learning stack, from improved neural network architectures [24], to deep learning compilers that increase software efficiency [3], to specialized DNN accelerators that increase hardware efficiency [4, 5]. In this paper, we consider two components from the deep learning stack: the hardware accelerator and the software compiler that maps a model onto that hardware, with the goal of automatically optimizing the $energy \times delay$ product of executing a particular model on a hardware accelerator. This area is commonly referred to as hardware/software co-design, and since it requires human expertise from multiple disciplines (software engineers, compiler writers, hardware architects), it is typically driven by manual heuristics or heuristic-based search [27].

We take a different approach, recognizing that for a given DNN model, this hardware/software co-design can be framed as a joint search of the space of all of the valid mappings and hardware architectures that can correctly execute the model. We formally parameterize this space based on prior work [19], and we find that standard optimization techniques are ill-suited to the exploration of the parameterized space because the vast majority of the points in the space are infeasible.

Our solution is to cast the search as a bilevel optimization problem. The outer loop optimizes over hardware architectures, while the inner loop optimizes over software mappings for a given architecture.

Both of these are heavily constrained black-box global optimization problems that require expensive simulations to obtain performance estimates. We therefore propose a nested, constrained Bayesian optimization (BO) formulation that uses Bayesian models of hardware and software performance to guide the search towards promising regions of the design space.

We find that when compared against the state-of-the-art manually-designed hardware accelerators that use heuristic software mappings, our BO-based approach provides significant improvements, improving the energy-delay product (EDP) by 16.0% to 40.2% on a series of neural networks.

2 Related Work

This section describes prior work in the hardware and software optimizations for DNNs.

2.1 Hardware to Optimize DNNs

Prior work has designed specialized hardware to execute BLAS kernels. Google's TPU [15] uses large hardware structures called systolic arrays [16], and NVIDIA's GPUs have tensor cores [1]. Acclerators such as Eyeriss [5] and focus on CNNs, introducing a specific dataflow that exploits a reuse pattern exhibited by 2D convolutions. Recent work [27] recognizes that the design space of specialized hardware is vast and proposes heuristics that can be leveraged to automatically synthesize hardware using a domain-specific language, Halide.

2.2 Software to Optimize DNNs

Software optimization process has been recognized as a search problem, and compilers such as TVM [3] have used learned cost models to optimize execution efficiency. Similarly, Timeloop uses a grid or random search to optimize software mappings on a user-specified hardware architecture [19]. However, all software optimizers treat hardware as a black box without explicitly considering the interaction between hardware and software.

Ours is the first work that systematically explores the space of both hardware and software optimizations; this larger search space requires a more principled search method, which motivates our constrained Bayesian optimization framework.

3 A Formal Representation of Software and Hardware

Hardware/software co-design is typically performed manually based on human insight, heuristics, and intuition. To facilitate an intelligent automation, this section formally defines the hardware and software design spaces.

3.1 Parameterizing the Design Space

Software design points can be parameterized by the loop ordering, loop tiling, and computational parallelism of the seven-level loop nest used to compute a convolutional layer (see appendix) [19, 27]. These software parameters are subject to hardware constraints, such as the quantity and layout of processing elements (PEs) and the size of storage elements.

Hardware parameters are generally more specific to the low-level resource and memory configurations or the layout of PEs. These can be broken down into a two broad categories:

Resource configurations represent the physical aspects of hardware, such as buffer sizes, tile sizes, and the cluster size of global buffers, as well as the layouts of the PE array and of the global buffer.

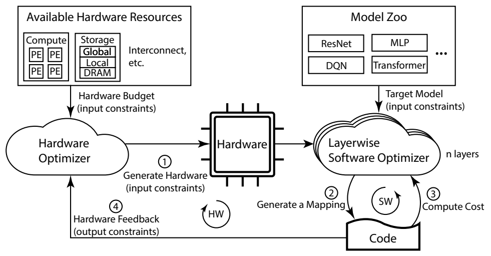
Dataflow configurations represent the usage of the PE array that are implemented in hardware, such as the blocking factors and degree of parallelism at the PE level, which also determines the communication patterns among PEs.

The appendix also shows the full listing of the parameters that we optimize.

3.2 Constraints in the Design Space

Hardware designs are fundamentally constrained by area (the total amount of compute and storage resources) and factors such as available memory bandwidth. For a specific hardware accelerator, the software optimization problem can be viewed as a search for the most efficient use of available hardware PEs and buffers. For example, the loop blocking optimization factors a neural network across multiple hardware storage buffers—and the feasible factorizations are constrained by the size of the hardware buffers.

4 Bayesian Optimization for Hardware/Software Co-design


Available Hardware Resources
Compute
PE
Storage
Global
Local
DRAM
Interconnect, etc.
Model Zoo
ResNet
MLP
DQN
Transformer
...
Hardware Budget (input constraints)
Target Model (input constraints)
Hardware Optimizer
Hardware
Layerwise Software Optimizer
n layers
1 Generate Hardware (input constraints)
2 Generate a Mapping
SW
3 Compute Cost
4 Hardware Feedback (output constraints)
HW
Code

```
// pick a target model (e.g. ResNet) and hardware budget
target = "ResNet"
// hardware optimization loop
for $i$ = 1 : number of hardware trials
    (1) hardware BO optimizer generates hardware $h_i$
    // software optimization loop
    for $j$ = 1 : number of layers in target
        // get Layer_EDP on $h_i$
        for $k$ = 1 : number of software trials
            (2) software BO optimizer generates a mapping $s_{ijk}$
            (3) evaluate mapping $s_{ijk}$ on hardware $h_i$ as
                Layer_EDP and feed back to layerwise
                software BO optimizer
    // compute Model_EDP on $h_i$
    (4) compute Model_EDP on $h_i$ as sunm of best layerwise EDP
        $\sum_j \min_k(s_{ijk})$ and feed back to hardware BO optimizer
```

Figure 1: Overview of BO-based nested search for hardware/software co-design.

4.1 Overview of Nested Hardware/Software Optimization

We propose a nested approach for co-optimizing hardware/software parameters. The overall approach is outlined in Figure 1. The goal is to find the optimal hardware parameters for a neural model and the optimal set of software parameters for each layer in the neural model.

Specifically, let $\mathbf{x}_h$ and $\mathbf{x}_s$ denote the set of hardware and software parameters. In the nested search process, we first use the hardware optimizer to generate a design of hardware. In particular, we perform the hardware search in the space of possible hardware $\mathcal{S}_h$ to optimize all hardware parameters, where the objective is to minimize $f(\mathbf{x}_h \mid \text{NN})$ which we define as the energy-delay product (EDP) of running the neural network (NN) model on the given hardware. This step produces a hardware specification and can be formalized as $\text{argmin}_{h \in \mathcal{S}_h} f(\mathbf{x}_h \mid \text{NN})$.

For the chosen hardware design, our framework performs the software search for each individual neural layer in its constrained software mapping space $\mathcal{S}_s|h, \text{NN}_j$ to optimize the mapping parameters, where NN_j denotes the jth layer in the neural network model, and the objective becomes $f(\mathbf{x}_s \mid \mathbf{x}_h, \text{NN}_j)$, which is defined as the EDP of running the layer j on the fixed hardware. This step produces a design point that represents the best set of software mappings for each layers on the given hardware structure, and can be formalized as $\text{argmin}_{s \in \mathcal{S}_s|h} f(\mathbf{x}_s \mid \mathbf{x}_h)$. The layerwise EDPs are then summed up as the EDP of the neural model, which is fed back to the hardware optimizer to generate the next hardware setting.

The iterative search between hardware and software will repeat for a user-defined number of trials. In this work, we set 50 for hardware search and 250 for software search. In our Bayesian optimization (BO) framework, we use separate BO models to search in the hardware and software space. We now describe their design considerations.

4.2 BO for Optimizing Hardware Architectures

Kernel design. The main design choice for BO is the GP kernel to use. For the hardware search, we choose a linear kernel on top of feature transformations that represent the relationship between the different parameters. We also add a noise kernel to deal with noise in the hardware evaluation. This is because the software optimizer is not guaranteed to find the best software mapping for each layer.

Constraints. There are both known and unknown constraints in the hardware search. The known constraints, such as the compute and storage budget, are treated as input constraints that reject invalid samples. The unknown constraints have to do with feasibility (if there exists valid software mappings of neural layers onto the hardware). These constraints are treated as output constraints and are modeled by a GP with a squared exponential kernel.

4.3 BO for Optimizing Software Mappings

Kernel design. Similar to hardware optimization, we use a linear kernel and transform the parameters to features that encode relational information. The evaluation of a mapping on a given hardware is deterministic in our infrastructure, thus there is no need for a noise kernel in the GPs.

Constraints. As both the hardware and neural model are known during software optimization, all constraints are known and are treated as input constraints that automatically reject invalid samples.

5 Evaluation

5.1 Methodology

Infrastructure. We conduct our evaluation on Timeloop [19], which is an open-source infrastructure for evaluating the hardware design and software optimization of DNN accelerators. In the evaluation, Timeloop takes two inputs: the hardware configuration and the software mapping. We limit the use case to inference in this work and leave training for future work.

Workloads. We use our BO framework to optimize critical layers from CNNs (ResNet [8] and DQN [18]), as well as an MLP and Transformer [26].

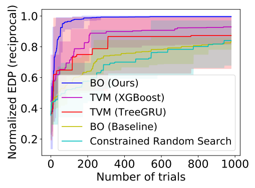
Experimental Setup. We use Eyeriss [5], a state-of-the-art DNN accelerator, as our main baseline. In the software mapping search, we use Eyeriss's hardware specifications and search for the best software mapping for each neural layer. In the hardware search, we perform the search under the same compute and storage resource constraints as Eyeriss for each neural model.

Metrics. We adopt the widely used energy-delay product (EDP) as the objective. As the actual EDP values vary across an order of magnitude, we normalize by dividing by the best (minimal) EDP value, and take the reciprocal for optimization curves.

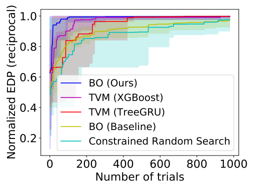
Baselines. In hardware search, we compare against constrained random search that repeatedly takes the first random sample in the design space that satisfies the constraints. In software search, we compare against constrained random search, and out-of-the-box BO and two TVM variants [3].

5.2 Software Mapping Optimization

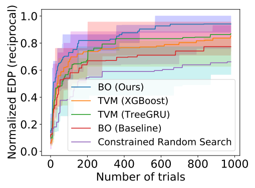
We show the results of software mapping optimization first, as the capability of finding a good mapping is the base of evaluating a hardware design. Figure 2 shows the improvements of BO over all baselines.



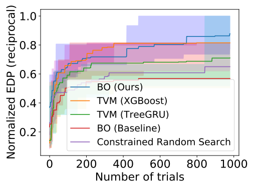
(a) ResNet-K2



(b) DQN-K2



(c) MLP-K2

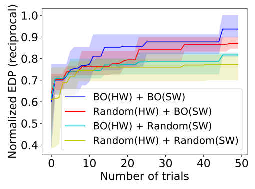


(d) Transformer-K2

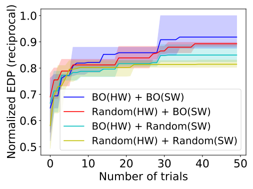
Figure 2: Software mapping optimization on layer 2 of ResNet, DQN, MLP, and Transformer. The y-axis shows the reciprocal of energy-delay product (EDP) (normalized against the best EDP value). Higher is better. Results for other layers can be found in the appendix. Best viewed in color.

5.3 Hardware Configuration Optimization

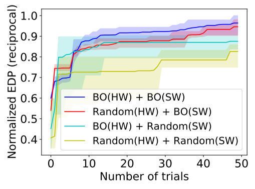
Figure 3 shows the optimization curves for hardware/software co-design. The comparison of hardware search algorithms shows that BO provides consistently better performance than the constrained random search, and the comparison of software search algorithms shows the importance of mapping optimization in the co-design process. We find that the designs searched by BO achieve significantly better EDP on all neural models compared to the state-of-the-art manually designed accelerator (18.3%, 40.2%, 21.8% and 16.0% for ResNet, DQN, MLP and Transformer respectively).



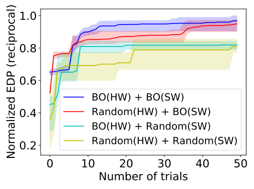
(a) ResNet



(b) DQN



(c) MLP



(d) Transformer

Figure 3: Hardware/software co-optimization. The x-axis shows the number of trials for hardware search. Best viewed in color.

6 Conclusion

In this paper, we have cast hardware/software co-design as a Bayesian optimization problem. We have shown that standard mechanisms have difficulty navigating the complex, highly constrained design space, so we have presented a novel constrained formulation that allows the optimizer to efficiently identify desirable points in this design space. The techniques described here are not limited to DNN architectures, which is significant because as we enter the golden age of computer architecture [9], it is essential that we develop automatic mechanisms for architectural exploration that quickly produce custom hardware accelerators.

References

[1] NVIDIA Tesla V100 GPU Architecture, The World's Most Advanced Data Center GPU. *NVIDIA Corporation*, 2017.

[2] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[3] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, 2014.

[5] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[6] Peter I Frazier. *Knowledge-gradient methods for statistical learning*. PhD thesis, Citeseer, 2009.

[7] Michael A Gelbart, Jasper Snoek, and Ryan P Adams. Bayesian optimization with unknown constraints. *Uncertainty in Artificial Intelligence*, 2014.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[10] Philipp Hennig and Christian J Schuler. Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, 13(Jun):1809–1837, 2012.

[11] Danny Hernandez and Tom B. Brown. Measuring the algorithmic efficiency of neural networks, 2020.

[12] José Miguel Hernández-Lobato, Matthew W Hoffman, and Zoubin Ghahramani. Predictive entropy search for efficient global optimization of black-box functions. In *Advances in neural information processing systems*, pages 918–926, 2014.

[13] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[14] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[15] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[16] HT Kung and Charles E Leiserson. Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics, 1979.

[17] Benjamin Letham, Brian Karrer, Guilherme Ottoni, Eytan Bakshy, et al. Constrained bayesian optimization with noisy experiments. *Bayesian Analysis*, 14(2):495–519, 2019.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[19] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315. IEEE, 2019.

[20] Carl Edward Rasmussen and Christopher K.I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[21] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[22] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[23] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[24] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2019.

[25] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[27] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–383, 2020.

7 Bayesian Optimization

7.1 Overview

Bayesian optimization [14, 2, 21] is an effective approach for the optimization of expensive, possibly noisy black-box functions. BO has been successfully used for hyperparameter optimization [22], algorithm configuration [13], optimizing A/B experiments [17], and more. For our problem, we have a parameterized representation and access to a simulator. Since one of our main concerns is sample efficiency, Bayesian optimization is particularly suitable.

The actual cost of evaluation depends on the experimental infrastructure, but in general, the evaluation is much more expensive on hardware than software. For example, the evaluation of a hardware design choice requires a faithful hardware implementation and a search for the optimal software mappings, a process that can take hours (with a simulator or FPGA) to days or even months (with an ASIC). By contrast, the cost of applying a list of software mappings and running the transformed workload is relatively low, ranging from seconds (in a simulator) to minutes (on a real hardware, FPGA or ASIC).

Bayesian optimization has two major components: 1) a surrogate model provides a Bayesian posterior probability distribution that predicts potential values of the objective function. 2) an acquisition function uses the model to identify the next point to evaluate.

7.2 Gaussian processes

A common surrogate model is a Gaussian process (GP) due to its simplicity and flexibility. A GP is prior distribution over the space of functions that is comprised of a mean function $m(\mathbf{x})$ and a covariance, or kernel function $k(\mathbf{x}, \mathbf{x}')$. Suppose we are given a dataset of N input/output pairs over a bounded domain Ω with D input dimensions and scalar outputs. For brevity, we write this as $(X, \mathbf{y})$, where $X \in \Omega^{N \times D}$ and $\mathbf{y} \in \mathbb{R}^N$. The posterior predictive distribution over function values f for a new input $\mathbf{x}$ is given by $P(f \mid \mathbf{x}, X, \mathbf{y}) = \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x}))$, where

$$\mu(\mathbf{x}) = K_{\mathbf{x}X} K_{XX}^{-1} (\mathbf{y} - \mathbf{m}_X) + m(\mathbf{x}),$$
$$\sigma^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - K_{\mathbf{x}X} K_{XX}^{-1} K_{\mathbf{x}X}^\top.$$

Where K_{XX} is a matrix formed by evaluating the kernel on X, $K_{\mathbf{x}X}$ is the vector of kernel evaluations between $\mathbf{x}$ and X, and $\mathbf{m}_X$ is the vector of mean function evaluations on the input dataset.

A common choice for the kernel is squared exponential. Given two input vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, this is defined as $k(\mathbf{x}_i, \mathbf{x}_j) = \alpha^2 \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\boldsymbol{\ell}^2}\right)$. α and $\boldsymbol{\ell}$ are kernel hyperparameters.

Another kernel that we find particularly useful is a linear kernel on top of explicit features. Given a feature mapping $\phi(\mathbf{x}) : \mathbb{R}^D \to \mathbb{R}^K$, the linear kernel can be written as $k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$. When we have strong prior information about the relevant feature interactions that govern the black-box function, this kernel allows us to encode these interactions directly and produces a more sample-efficient posterior estimate.

In cases where the observations from the black-box function are noisy, we can add a noise kernel $K_{\text{noise}} = \tau^2 \mathrm{I}$ to K_{XX}, where τ^2 is a hyperparameter. This implies a Gaussian observation likelihood.

Following common practice, we use a relatively simple mean function. In particular, we use the constant mean $m(\mathbf{x}) = c \quad \forall\, \mathbf{x}$. All kernel and mean hyperparameters are learned by maximizing the marginal likelihood of the GP on the current dataset.

7.3 Acquisition functions

A critical component in the BO framework is the choice of acquisition function $a(\cdot)$ that assigns each design point a value that represents the utility of testing this point. Two commonly used acquisition functions are expected improvement (EI) and lower confidence bound (LCB).

EI computes the amount we expect to improve upon the current best observed objective value $y_* \equiv \max\{y_i\}_{i=1}^N$ by evaluating a design point $\mathbf{x}$. Formally, it can be written as

$$a_{\text{EI}}(\mathbf{x}) = \int_{-\infty}^{\infty} \max(y_* - f, 0) P(f \mid \mathbf{x}, X, \mathbf{y}) \mathrm{d}f.$$

where f is the latent function from the surrogate model, and y_* is the best value observed.

LCB [23] provides an explicit tradeoff between the predictive mean and variance and is defined as

$$a_{\text{LCB}}(\mathbf{x}) = \mu(\mathbf{x}) + \lambda \sigma(\mathbf{x}).$$

Where λ represents a tradeoff parameter. A small λ promotes greater exploitation, and a large λ promotes greater exploration. We found $\lambda = 1$ to work well in our experiments. Beyond these, there are many other possible acquisition functions that could be used in future exploration [25, 10, 12, 6].

7.4 Constraints

In our problem, the vast majority of the design space will produce invalid solutions. When the constraints are a known function of the input features, we can directly account for them as input constraints. Otherwise, we must run the simulation and treat invalid points using an output constraint. Here, we will describe these constraint types, and how they are incorporated into BO.

Input constraints are explicit constraints that are used when optimizing the acquisition function. They directly prevent the search from suggesting points that will violate the constraints. As some constraints are non-linear, this optimization is itself very challenging. We optimize the acquisition function in a crude way by performing rejection sampling on the design space: we randomly sample parameters until we obtain 150 feasible points, and then choose the one the maximizes the acquisition function. On average the sampling takes 22K random samples to get a pool of 150 feasible points.

Output constraints are used when we do not know the form of the constraint a-priori and must run the simulator to test feasibility. This is also referred to as an "unknown" constraint, and BO has been adapted to incorporate a constraint model in addition to the regression model [7]. These simultaneously learn about the constraint boundaries while modeling the objective.

Let $\mathcal{C}(\mathbf{x})$ denote the event that $\mathbf{x}$ satisfies constraint $\mathcal{C}$. Constrained BO uses a Bayesian classifier to model $P(\mathcal{C}(\mathbf{x}))$. It is relatively straightforward to adapt a GP regressor to classification [20].

Under a Bayesian classifier, the acquisition function $a(\mathbf{x})$ is modified to account for the probability that the constraint is satisfied, with 0 utility if it is not satisfied.

$$\bar{a}(\mathbf{x}) = \mathbb{E}[a(\mathbf{x})\mathrm{I}[\mathcal{C}(\mathbf{x})]] = P(\mathcal{C}(\mathbf{x}))a(\mathbf{x}).$$

Where $\mathrm{I}[\mathcal{C}(\mathbf{x})]$ is the indicator function that evaluates to 1 if the constraint is satisfied and 0 otherwise. We therefore maintain two models: one regression model to capture the objective and one classifier to model the constraint in order to avoid evaluations in infeasible regions.

8 Parameters and constraints

Type	Index	Hardware Parameters	Valid Range	Meaning
PE	H1	PE mesh-X	Factors of # PEs	Decide the arrangement of the 2-D PE array.
	H2	PE mesh-Y	Factors of # PEs	
Local buffer	H3	Input entries in Local buffer	0 to # local buffer entries	Decide the partition of local buffer. The partition leads to sub-buffers with inflexible sizes. This is useful as the latency to access each smaller sub-buffer decreases.
	H4	weights entries in Local buffer	0 to # local buffer entries	
	H5	outputs entries in Local buffer	0 to # local buffer entries	
Global buffer	H6	Global buffer instances	Factors of #PEs	Determine the arrangement of global buffer, and its connection between global buffer and per PE's local buffer (Local buffer of PEs along the X-axis shares the instances of global buffer along the X-axis).
	H7	Global buffer mesh-X	Factors of PE-mesh-X	
	H8	Global buffer mesh-Y	Factors of PE-mesh-Y	
	H9	Global buffer block size	Factors of 16	Determines the width of a global buffer entry
	H10	Global buffer cluster size	Factors of 16	Determines of the number of wider structures where multiple entries are ganged into
Dataflow	H11	Dataflow option of filter width	1, 2	Options that determine the size of filter width in PE's local buffer
	H12	Dataflow option of filter height	1, 2	Options that determine the size of filter height in PE's local buffer

Figure 4: Hardware parameters.

9 Hyperparamters for BO

In Figure 8 we report the hyperparamters for BO.

Type	Hardware Constraints
PE	PE mesh-X (H1) * PE mesh-Y (H2) = # PEs
Local buffer	The sum of local sub-buffers (H3, H4, H5) does not exceed buffer size
Global buffer	Global buffer mesh-X (H7) * global buffer mesh-Y (H8) = # Global buffer instances (9)
Local buffer & global buffer (unknown)	A valid software mapping exists depending mainly on local buffer partition (H3, H4, H5) and global buffer arrangement (H6, H7, H8)

Figure 5: Hardware constraints.

Type	Index	Software Parameters	Valid Range	Meaning
Loop blocking and degree of parallelism	S1	Blocking factors of R	Factors of R	Determines the size (parallelism) of each type of data (inputs, weights and outputs) in each storage layer (except those that are in the hardware dataflow).
	S2	Blocking factors of S	Factors of S	
	S3	Blocking factors of P	Factors of P	
	S4	Blocking factors of Q	Factors of Q	
	S5	Blocking factors of C	Factors of C	
	S6	Blocking factors of K	Factors of K	
Loop reorder	S7	Loop order in local buffer	Permutations of non-1 factors	Affects the reuse of each type of data (inputs, weights and outputs) in each storage layer.
	S8	Loop order in global buffer	Permutations of non-1 factors	
	S9	Loop order in DRAM	Permutations of non-1 factors	

Figure 6: Software parameters.

10 Neural Model Specifications.

In Figure 9 and Figure 10 we report the specifications of neural models benchmarked in this paper.

11 Paramterization of 2D Convolution

Listing 12 gives the seven-level nested loop that comprises a 2D convolution.

Figure 13 shows a design point for the CONV4 layer of ResNet. The architecture components are again the same as in the 1D example, but since the memory footprint is significantly larger, the PE can no longer capture all data reuse, so the Global Buffer must store large portions of the inputs and outputs.

12 Additional results

12.1 Software optimization

In Figure 14 we show more examples of the software optimization over multiple layers of the different architectures. Our Bayesian optimization formulation consistently outperforms the baselines [3].

12.2 Ablations

In Figure 15 we compare different surrogate models and acquisition functions for Bayesian optimization of the software mapping. We found Gaussian processes with LCB to consistently outperform other alternatives.

In Figure 16 we investigate the robustness of LCB for software optimization using different values of λ. We found that $\lambda = 0.1$ tends to be too greedy, but that above $\lambda = 0.5$, LCB tends to be fairly robust.

Type	Software Constraints
Loop blocking and degree of parallelism	Product of all blocking factors of R (S1) equals R of the target neural layer
	Product of all blocking factors of S (S2) equals S of the target neural layer
	Product of all blocking factors of P (S3) equals P of the target neural layer
	Product of all blocking factors of Q (S4) equals Q of the target neural layer
	Product of all blocking factors of C (S5) equals C of the target neural layer
	Product of all blocking factors of K (S6) equals K of the target neural layer
Buffer capacity (local)	Inputs/weights/outputs sizes (S1-S6) cannot exceed corresponding local sub-buffer capacity
Buffer capacity (global)	Size of all types of data (S1-S6) does not exceed global buffer capacity
Parallelism	Product of blocking factors in global buffer X-axis (S1-S6) cannot exceed # PEs in X-axis
	Product of blocking factors in global buffer (S1-S6) cannot exceed total # PEs

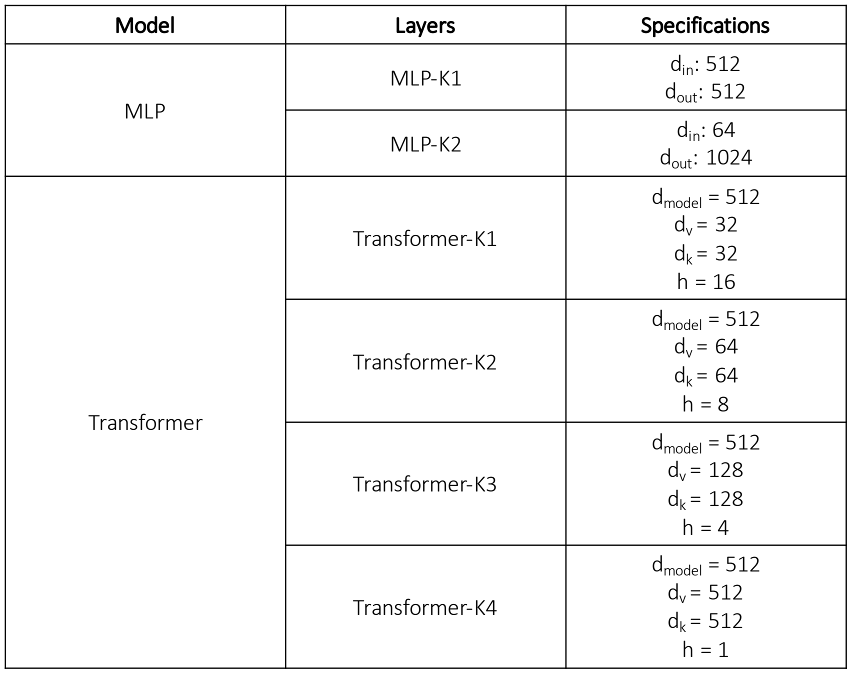
Figure 7: Software constraints.

number of independent trials	5 (HW), 10 (SW)
number of random data points	50 (HW), 150 (SW)
number of warmup data points	5 (HW), 30 (SW)
number of samples for EI	1000
lambda for LCB	1.0

Figure 8: Hyperparamters for BO.

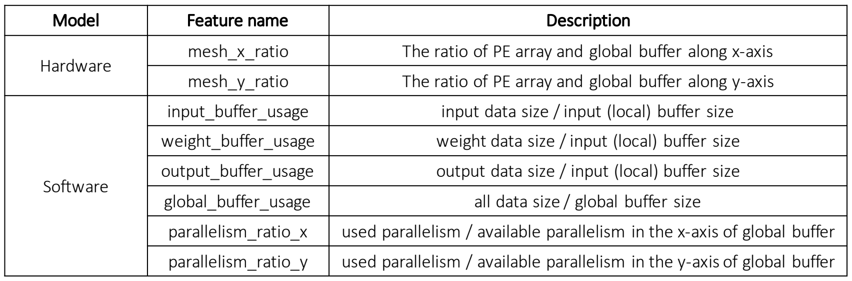
Model	Layers	Specifications
ResNet	ResNet-K1	Filter size: 3×3 Output size: 56×56 # input channel: 64 # output channel: 64 Stride: 2
	ResNet-K2	Filter size: 3×3 Output size: 28×28 # input channel: 128 # output channel: 128 Stride: 1
	ResNet-K3	Filter size: 3×3 Output size: 14×14 # input channel: 256 # output channel: 256 Stride: 1
	ResNet-K4	Filter size: 3×3 Output size: 7×7 # input channel: 512 # output channel: 512 Stride: 1
DQN	DQN-K1	Filter size: 8×8 Output size: 20×20 # input channel: 4 # output channel: 16 Stride: 4
	DQN-K2	Filter size: 4×4 Output size: 9×9 # input channel: 16 # output channel: 32 Stride: 2

Figure 9: Specifications of ResNet (ResNet-18) [8] and DQN [18]



Model	Layers	Specifications
MLP	MLP-K1	d_{in}: 512 d_{out}: 512
	MLP-K2	d_{in}: 64 d_{out}: 1024
Transformer	Transformer-K1	d_{model} = 512 d_v = 32 d_k = 32 h = 16
	Transformer-K2	d_{model} = 512 d_v = 64 d_k = 64 h = 8
	Transformer-K3	d_{model} = 512 d_v = 128 d_k = 128 h = 4
	Transformer-K4	d_{model} = 512 d_v = 512 d_k = 512 h = 1

Figure 10: Specifications of MLP and Transformer [26]

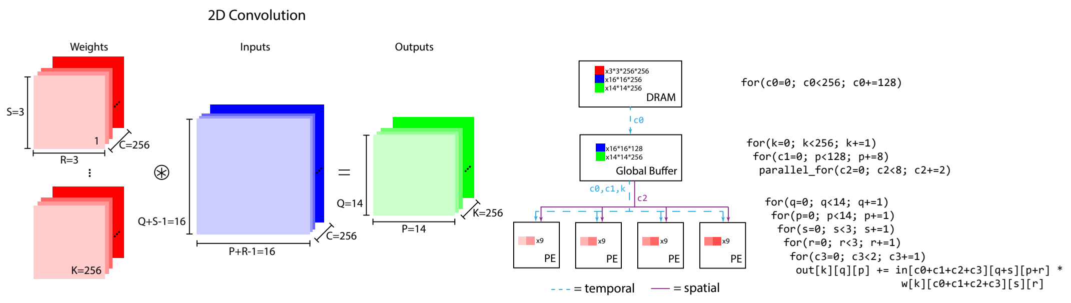


Model	Feature name	Description
Hardware	mesh_x_ratio	The ratio of PE array and global buffer along x-axis
	mesh_y_ratio	The ratio of PE array and global buffer along y-axis
Software	input_buffer_usage	input data size / input (local) buffer size
	weight_buffer_usage	weight data size / input (local) buffer size
	output_buffer_usage	output data size / input (local) buffer size
	global_buffer_usage	all data size / global buffer size
	parallelism_ratio_x	used parallelism / available parallelism in the x-axis of global buffer
	parallelism_ratio_y	used parallelism / available parallelism in the y-axis of global buffer

Figure 11: Extra features used by the hardware and software BO optimizers.

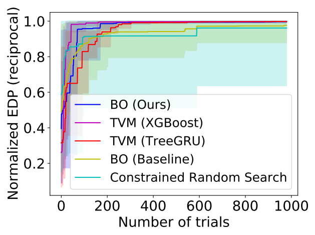
```
for n in [0:N)
  for k in [0:K)
    for r in [0:R)
      for s in [0:S)
        for p in [0:P)
          for q in [0:Q)
            for c in [0:C)
              outputs[n][k][q][p] += weights[k][c][s][r] *
                                     inputs[n][c][q+s][p+r]
```

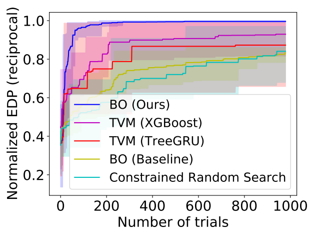
Figure 12: Computing a 2D convolution with a seven-level nested loop.

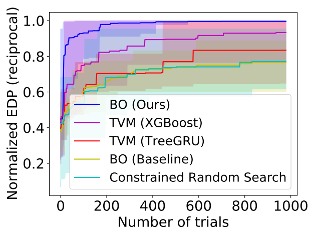


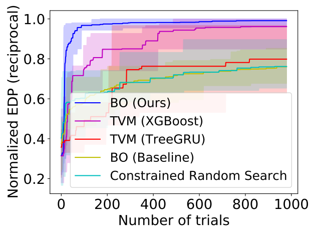
```
for(c0=0; c0<256; c0+=128)

 for(k=0; k<256; k+=1)
  for(c1=0; p<128; p+=8)
   parallel_for(c2=0; c2<8; c2+=2)

    for(q=0; q<14; q+=1)
     for(p=0; p<14; p+=1)
      for(s=0; s<3; s+=1)
       for(r=0; r<3; r+=1)
        for(c3=0; c3<2; c3+=1)
         out[k][q][p] += in[c0+c1+c2+c3][q+s][p+r] *
                                  w[k][c0+c1+c2+c3][s][r]
```

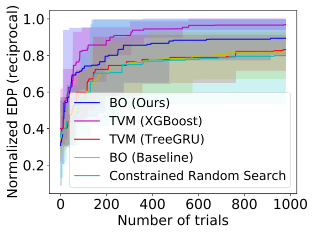
Figure 13: An architecture computing the CONV4 layer of ResNet.

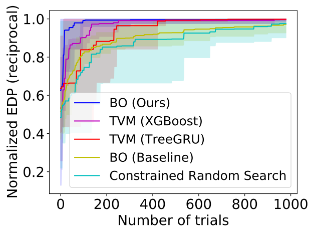

(a) ResNet-K1

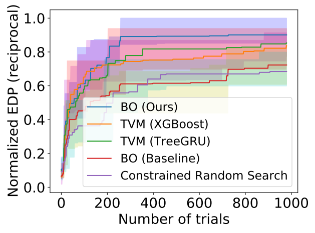

(b) ResNet-K2

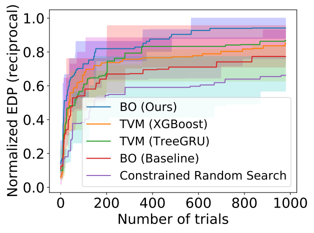

(c) ResNet-K3

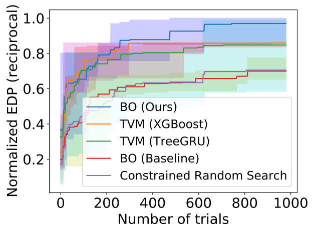

(d) ResNet-K4

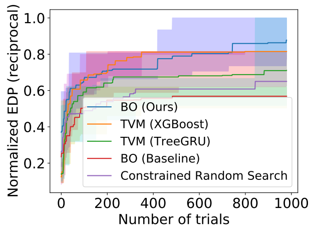

(e) DQN-K1

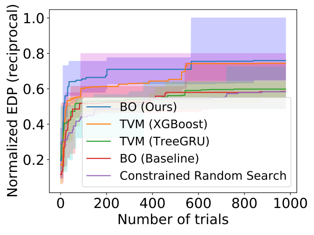

(f) DQN-K2


(g) MLP-K1


(h) MLP-K2


(i) Transformer-K1


(j) Transformer-K2


(k) Transformer-K3

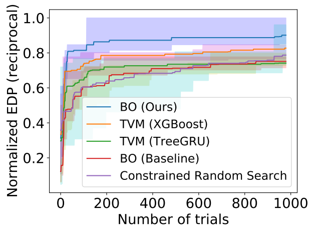
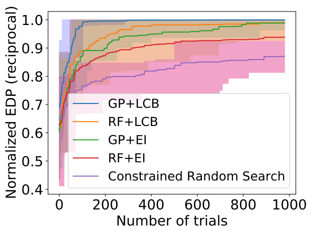
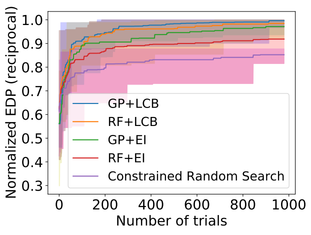

(l) Transformer-K4

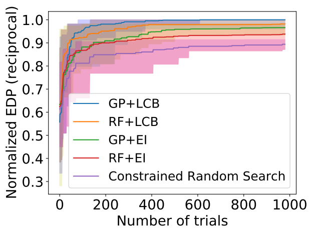
Figure 14: Software mapping optimization on ResNet, DQN, MLP, and Transformer. The Y-axis shows the reciprocal of energy-delay product (EDP) (normalized against the best EDP value). Higher is better.



(a) ResNet-K2

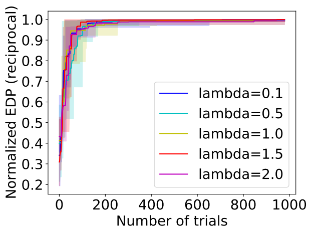


(b) ResNet-K3

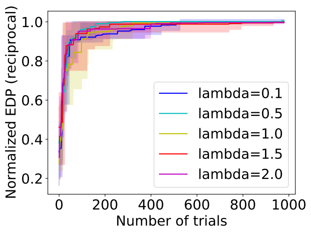


(c) ResNet-K4

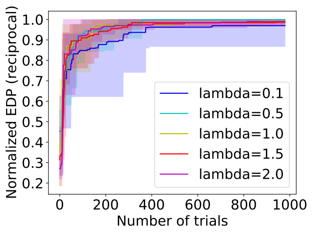
Figure 15: GP with different surrogate models and acquisition functions.



(a) ResNet-K2



(b) ResNet-K3



(c) ResNet-K4

Figure 16: LCB acquisition function with different lambda values.