Glimpse: Mathematical Embedding of Hardware Specification for Neural Compilation

Byung Hoon Ahn  Sean Kinzer  Hadi Esmaeilzadeh
University of California, San Diego  {bhahn,skinzer,hadi}@eng.ucsd.edu

ABSTRACT
Success of Deep Neural Networks (DNNs) and their computational intensity has heralded Cambrian explosion of DNN hardware. While hardware design has advanced significantly, optimizing the code for them is still an open challenge. Recent research has moved past traditional compilation techniques and taken a stochastic search algorithmic path that blindly generates rather stochastic samples of the binaries for real hardware measurements to guide the search. This paper opens a new dimension by incorporating the mathematical embedding of the hardware specification of the GPU accelerators dubbed Blueprint to better guide the search algorithm and focus on sub-spaces that have higher potential for yielding higher performance binaries. While various sample efficient yet blind hardware-agnostic techniques have been proposed, none of the state-of-the-art compilers have considered hardware specification as hints to improve the sample efficiency and the search. To mathematically embed the hardware specifications into the search, we devise a Bayesian optimization framework called Glimpse with multiple exclusively unique components. We first use the Blueprint as an input to generate prior distributions of different dimensions in the search space. Then, we devise a light-weight neural acquisition function that takes into account the Blueprint to conform to the hardware specification while balancing the exploration-exploitation trade-off. Finally, we generate an ensemble of predictors from the Blueprint that collectively vote to reject invalid binary samples. We compare Glimpse with hardware-agnostic compilers. Comparison to AutoTVM [3], Chameleon [2], and DGP [16] with multiple generations of GPUs shows that Glimpse provides 6.73×, 1.51×, and 1.92× faster compilation time, respectively, while also achieving the best inference latency.

ACM Reference Format:

1 INTRODUCTION
Prevalent adoption of Deep Neural Networks (DNNs) in voice assistants, smart speakers, and enterprise applications has triggered a Cambrian explosion of DNN hardware to cope with the colossal computational intensity of DNNs. While the hardware designs have

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

DAC ’22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s).
https://doi.org/10.1145/3489517.3530590

advanced significantly, inseparable task of generating optimized code for them is still an open challenge. In fact, hand-optimized libraries such as NVIDIA cuDNN or Intel MKL that serve backend for programming interfaces such as TensorFlow [1] and PyTorch [13] have been the go-to solutions for higher performance DNN execution. However, recent research in neural compilers has taken a leap beyond hand-optimized libraries and traditional compilation techniques, and embraced stochastic search algorithms such as simulated annealing to improve the search. These search algorithms navigate an exponentially large search space for the optimized code, which is one of the main reason behind the success of optimizing compilers [4]. To traverse the search space in a sample efficient manner, recent innovations in optimizing compilers strived to reduce the compilation time with cost models to approximate the large search space [3, 14] and effective search algorithms [2, 16]. However, these search algorithms [2, 3, 9, 14, 16, 18], classified as black-box optimization, are blindly and solely guided by the real hardware measurements. These measurements, however, comes at a large cost in terms of time yet barely provides any architectural hints to effectively guide the search algorithms due to their blindness. As such, although these neural compilers have made their way into the deep learning pipelines of major deep learning solutions providers including Amazon, Xilinx, and Qualcomm, the current paradigm of hardware-agnostic neural compilers takes hours to optimize even a small model. In fact, this even grows to days on GPUs to optimize multitude of models on many GPU accelerators1, which curtails the overall productivity in DNN model deployment.

This paper sets out to explore a new path where we provide neural compilers with perception such that it can take a glimpse of the mathematical embedding of the hardware blueprints to better guide the search algorithm. We devise a Bayesian optimization framework called Glimpse that uniquely explores the mathematical embedding of the GPU specifications dubbed Blueprints to expedite the neural compilation while also improving the resulting binary performance. We first use Blueprints to generate a set of prior distributions of different dimensions of the search space. Then, we devise a lightweight neural acquisition function learned using meta-learning-based algorithm that takes into account the Blueprint to conform to the hardware while balancing the exploration-exploitation trade-off. Finally, we generate an ensemble of predictors from the Blueprint that collectively vote to reject invalid binary samples. We compare Glimpse with state-of-the-art hardware-agnostic neural compilers AutoTVM [3], Chameleon [2], and DGP [16] with modern DNNs including AlexNet [8], ResNet-18 [7], VGG-16 [15] on multiple generations of GPUs including Titan Xp, RTX 2070 Super, RTX 2080 Ti, RTX 3090. Integration of Glimpse to TVM [4] shows that Glimpse

1For example, 10 DNN models on 100 different GPUs would take around 10,000 GPU hours to optimize which translates to $9,000 with Amazon EC2 instances (on-demand, p2.xlarge). This is an exorbitant (per model update) cost for businesses considering the swift evolution of the neural architectures deployed in real world applications.
Although the problem of neural compilation as stated in Section 2.1 provides 6.73x, 1.51x, and 1.92x faster compilation time over AutoTVM, Chameleon, and DGP, respectively, while also achieving the best inference latency. Further analysis show up to 2.18× improvement in the initial configurations over transfer learning, 5.07× and 2.55x reduction in the number of search steps compared to AutoTVM and Chameleon. Glimpse also reduces invalid configurations by 5.56× and 4.53× over AutoTVM and Chameleon.

2 CHALLENGES IN NEURAL COMPILATION

After the models are trained using programming interfaces such as TensorFlow [1] or PyTorch [13], they are sent to the deployment engineers whose goal is to make sure the models meet various Quality-of-Service (QoS) requirements such as inference latency in end-to-end applications. The deployment engineers utilize optimizing compilers such as TVM [4] to tune the performance on a given target hardware, we use the term Neural Compilers throughout the paper. In fact, major deep learning solution providers such as Amazon, Xilinx, and Qualcomm incorporate these neural compilers within their Software Development Kit (SDK).

2.1 Neural Compilation for Model Deployment

Current neural compilers generally try to optimize \( s \in S \) while considering the target hardware as a black-box function \( f(x_s) \), where \( x \) and \( s \) are the code templates (e.g., Conv2D, Dense, and etc.) and their configuration (sampled from combinations of tiling, bindings, unrolling, and etc.), respectively. Usually the size of the overall search spaces \( S \) is astronomically large, which render simple grid search algorithms impractical. For example, the first layer of VGG-16 has over 200 million combinations. To make this worse, these search spaces are not differentiable, and the optimal configurations are sparsely distributed throughout the search space making it a complex problem to solve. Recent advances in neural compilation [2, 3, 9, 14, 16, 18] have introduced a cost model \( \hat{f} \approx f \) that approximates the vast search space and proposed intelligent search algorithms that better navigates the search space. However, the neural compilers still suffer from long compilation times of over tens of hours to days for even a single neural network.

2.2 Challenges and Opportunities in Neural Compilation

Although the problem of neural compilation as stated in Section 2.1 is already difficult, current neural compilation formulation has a narrow focus on a single hardware. However, in reality, there are multiple generations of hardware that are embedded in the intelligent devices. For instance, if we consider GPUs that are widely used to execute DNNs, generations of the GPU (e.g., Pascal, Turing, Ampere, etc.) vary machine by machine. To this end, the deployment engineers are left with a formidable task of tuning the DNN model for multiple not single target hardware, meaning \( n \) repetitions of the overall neural compilation for \( n \) hardware. In other words, considering \( \theta \in \Theta \) (where \( \Theta \) encodes the hardware configurations such as number of different cores, clocks, bandwidth, bus types, and etc.), problem formulation must be updated to:

\[
\hat{s}_* = \arg\max f(x_s|\theta_k) \quad \text{for } s \in S \text{ and many } k \in \mathbb{N}
\]

Simplest approach to cope with the variations in hardware is to just ignore and reuse the optimized configuration from another hardware. For example, using \( \hat{s}_* \) from Titan Xp to compile DNN on RTX 2080 Ti. However, this may not result in the optimized performance we desire. In fact, Figure 1 shows that while the overall search space takes a similar shape for different hardware, the optimal configuration differs among them. For ResNet-18 7th layer, reusing \( \hat{s}_* \) led to 27.79% slowdown of the output code for Titan Xp→RTX 2080 Ti, and 31.33% for RTX 2080 Ti→Titan Xp. On the other hand, transfer learning [3] is the most common way of reusing the compilation experiences. However, this also suffers from similar degradation in the performance of the resulting binary. An alternative approach would be to develop multiple neural compilers, one for each hardware, but this is neither cost-effective nor scalable solution to the long neural compilation time problem. Most importantly, such approach cannot cope with the constant evolution of the hardware. Simply put, current hardware-agnostic techniques are not scalable. On the other end of the spectrum, some analytical model or a simulator within the neural compilation loop to give full view of the hardware to run a white-box optimization, the confidentiality of the hardware design and the potential slow down of the compilation process from the complex hardware prohibits this.

However, silver lining here is that: (i) while the precise blueprints of the hardware are difficult if not impossible to get and use in neural compilation, some features or the specification of the hardware are available in public data sheets [12], and (ii) despite the fact that optimal solutions are different for different hardware, their search spaces have similar characteristics that open up opportunities to transfer the optimization experiences. Overall, the macro view of the problem of neural compilation for multiple hardware makes the problem more challenging, yet introduces a new unexplored dimension in designing neural compilers: hardware-awareness.

3 Glimpse: Mathematical Embedding of Hardware Specification for Faster Neural Compilation

Deviating from the current blind and hardware-agnostic neural compilers, we propose Glimpse, a novel neural compiler with perception to take a sneak peek of the hardware specifications in the form of mathematical embedding dubbed Blueprint. We first devise a
mathematical embedding **Blueprint** to encapsulate the hardware specifications. Then, we develop a **hardware-aware** neural compiler dubbed **Glimpse** that takes the **Blueprints** to take a glimpse of the hardware blueprint to adaptively and quickly optimize the input DNNs to the target hardware. To this end, this work can be subdivided into two main components that work together: (i) **Blueprint**, a mathematical embedding that encodes key specifications of the hardware, and (ii) **Glimpse** that translates the embedding into useful knowledge such as prior distributions to guide the search, search strategy in the form of neural acquisition function that can expedite the optimizing compilation, and ensemble of predictors to reject the invalid configurations. Figure 2 illustrates the overall flow of the compilation with **Glimpse** and **Blueprint**.

### 3.1 **Blueprint**: Mathematically Embedding Architectural Features of Hardware

To provide **hardware-awareness** to the neural compiler, we need to feed the neural compiler with the specification about the target hardware. However, unlike with white-box optimization where we would have the full view of the design and the specification of the hardware enabling explicit description of the hardware within the neural compiler, the complexity of the hardware designs as well as the confidentiality of the designs make it hard if not impossible to get the design. To close the structural gap between the demand for faster DNN deployment hence faster neural compilation and the practical difficulty in incorporating hardware information, **Glimpse** utilizes the architectural specifications provided by the vendors in public data sheets [12]. The data sheet lists the number of different processors/cores, bus interfaces, cache size, clock cycles, and the compute capacity in GFLOPS provided by the manufacturer. **Glimpse** creates a mathematical embedding of these specifications. These mathematical embeddings can provide neural compilers with a sneak peek of the architecture, and as a result provide hints about the search space and assist compiler while learning to quickly optimize tensor programs to better optimality.

**Design.** We devise a novel abstraction of the hardware dubbed **Blueprint** which is a mathematical embedding vector that summarizes the important features of the target hardware. Two key considerations while developing the **Blueprint** are (i) minimizing the loss of information while (ii) maintaining low overhead. While (i) is an obvious objective, (ii) is one of the key subtleties. As suggested in Section 2, one of the key challenges we face in developing neural compilers is the eons of time required for optimization. Therefore, one of the key design consideration was reducing the size of the embeddings that can impact the compilation time. In fact, parsing overhead for the neural compilers to gain architectural insights from the **Blueprint** may accrue to constitute a significant fraction of the neural compilation time. We perform a **dimensionality reduction** of the original feature vectors using **Principal Component Analysis (PCA)** to get the minimal mathematical embedding vector that summarizes the hardware. We use PCA over neural autoencoders as PCA provides an intuitive knob that allows us to balance the size with the information loss. On the other hand, using neural autoencoders would require more complex design space exploration of the neural model. Also, neural networks required more computation to achieve the same dimensionality reduction.

Prior distribution generation from **Blueprint**. We consider the neural compilation as a Bayesian optimization problem where the optimization begins with a prior distribution and updates the distribution over multiple iterations to gradually improve the posterior distribution, improving the quality of sampled binaries as we progress through the compilation. While this prior distribution can be learned from scratch, this has been shown to be very inefficient [2, 3]. As such, we use the aforementioned **Blueprint** and the network specification to generate the prior distribution that can speed up the compilation significantly. We use a parametric neural model \( f_k^*(\pi) \equiv f_k \) instead of non-parametric Gaussian processes to approximate the spaces. Then, taking inspiration from HyperNetworks [6], we devise a prior distribution generator \( \mathcal{H} \) that takes a layer specification and **Blueprint** as input and outputs the parameters \( \pi \) for the prior distribution \( f_k^*(\pi) \). To train \( \mathcal{H} \), we gathered a large scale dataset similar to [19] of \( s \) and \( f \). One important design choice for \( \mathcal{H} \) was generating \( n \) distributions for \( n \) dimensions of the search space. \( \mathcal{H} \) generates \( f_{k,l_{tile,x}} \) and \( f_{k,l_{tile,y}} \) for the dimensions \( l_{tile,x} \) and \( l_{tile,y} \), respectively. To get the initial samples from the search space, **Glimpse** enumerates combinations of the \( \text{argmax} (f_{k,s}) \), weighted by the \( \Pi f_{k,s} \). Overall, this prior distribution generator \( \mathcal{H} \) serves an effective initialization for the optimizing compilation procedure, reducing the number of costly hardware measurements to locate optimal configuration \( s^* \). Importantly, as prior distribution generation from **Blueprint** is a one-off process per layer, the computational cost of \( \mathcal{H} \) was negligible.

### 3.2 Hardware-Aware Exploration: Adapting Optimization Steps with Meta-learning

Current **hardware-agnostic** techniques [2, 3, 9, 14, 16, 18] take **black-box** approach and utilize stochastic optimization algorithms. To transfer the experience among different compilation instances, above method such as AutoTVM [3] uses the cost model as a proxy to transfer knowledge among similar layers. While these approaches allow the users to reuse the cost model, they still require significant number of real hardware measurements before they start yielding satisfactory output code. Likewise, reusing cost models among different hardware usually yield sub-optimal output code as stated in Section 2.2. The main reason for such sub-optimal performance is because the subtle differences in the architecture leads to significant, yet nonlinear, changes in the performance for the target hardware. Unlike these naive approaches to transfer experiences, **Glimpse** leverages the information encapsulated in **Blueprints** to improve the **hardware-awareness** of the exploration process. The main insight is that, while the exact locations of the optimal configuration in the search spaces may be different among multiple hardware, the know-hows on how to achieve that optimal configuration may be transferable. **Glimpse** incorporates a **hardware-aware**
strategy to conduct the search. In particular, we take inspiration from MetaBO [17] to learn the Meta-Optimizer in the Figure 3 to emit neural acquisition functions $f(·|θ)$ for Hardware-Aware Exploration that dictates the exploration and exploitation strategy.

Training. Training first begins by sampling the maximums $X_0$ from the prior distribution from Section 3.1. Then, we follow the natural Bayesian optimization pass of (i) sampling initial solutions from the surrogate cost model $f$, (ii) Hardware-Aware Exploration to determine the configurations $X_0$ to explore, and (iii) Hardware-Aware Sampling to prune invalid configurations to determine the candidates for real measurements $X_0′$. Measurements $f(·|θ)$, Tuples of configuration and the optimization budget $(X_0, t, T)$ (state where $t$ and $T$ are the optimization step and the budget, respectively, and the optimal configuration $x_*$ $\in$ $X_0$ (action) are collected as the dataset to train the Meta-Optimizer. Highlighted inside the brackets translates the Glimpse training setting into the reinforcement learning parlance, similar to the [17]. We iterate through various hardware and networks to train our Meta-Optimizer. As we progress through the Meta-Optimizer training, the Hardware-Aware Exploration that gets emitted gradually improves and learns to (i) make the optimal trade-off between exploration-exploitation and, more importantly, (ii) learn how to incorporate the hardware-awareness in the Hardware-Aware Exploration module. Final outcome of this off-line process is the hardware-aware optimization strategy ingrained in the Hardware-Aware Exploration module.

3.3 Hardware-Aware Sampling: Using Statistics to Minimize Invalid Configurations

Besides the above innovations, Glimpse tackles an innate issue in neural compilers: frequent invalid configurations. Chameleon [2] suggested using clustering to samples the centroids to reject invalid configurations. However, clustering-based sampling is hardware-agnostic, and it fails to filter out many of the invalid configurations, leading to significant waste in GPU time and low (real measurements) sample efficiency. In contrast, Glimpse incorporates the hardware-guided approach to reject invalid configurations. Glimpse generates an ensemble of predictors $p$ for different dimensions of the search space from the Blueprints. For example, $p_{\beta_x}$ and $p_{\beta_y}$ are generated for $\beta_x$ and $\beta_y$, respectively. For each configuration sampled from the Hardware-Aware Exploration, ensemble predictors vote the validity of the configuration. Sampler rejects the configuration if considered invalid by more than $\tau^2$ of the predictors.\footnote{We use $\tau = \frac{1}{4}$, found through a gridsearch hyperparameter search.}

Design. Instead of a large and complex monolithic predictor could be an alternative design point for Hardware-Aware Sampling in Glimpse, we use an ensemble of light-weight predictors for two reasons. First, statistically speaking, ensemble methods have been shown to yield a better predictive performance than could be obtained from any of the constituent predictor alone. In this case, comparable to a large complex monolithic predictor. In fact, smaller predictors are more appropriate considering the dearth amount of data. Furthermore, as key design consideration for neural compilers is the compilation speed for higher overall productivity, ensemble of light-weight predictors were used to minimize computational overhead of prediction. These predictors are super fast as they are threshold-based: their time complexity is $O(1)$ over Chameleon [2]’s $O(nkl)$, where $n$ is the number of samples, $k$ is the number of clusters, and $l$ is the number of iterations.

Integration and implementation. Algorithm 1 summarizes the overall flow of Glimpse with Blueprint. We use PyTorch [13] to implement $H$ for prior generation and the meta-optimization.

4 EVALUATION

We integrate Glimpse with Apache TVM v0.8 [4] to perform evaluation of both component and end-to-end scenario. We ran our framework on host machine with AMD Ryzen 7 3700X, 64GB DDR4, with NVIDIA RTX 2070 Super, and used CUDA 11.3 to program DNNs onto GPUs. We compare Glimpse against the state-of-the-art optimizing compilers: AutoTVM [3], Chameleon [2], and DGP [16]. We optimize AlexNet [8], VGG-16 [15], and ResNet-18 [7] on multiple generations of GPUs connected via RPC (Titan Xp, RTX 2070 Super, RTX 2080 Ti, RTX 3090) as summarized in Table 1.

4.1 Blueprint

Design space exploration of Blueprint. Unlike hardware-agnostic proposals [2, 3, 9, 14, 16, 18] Glimpse utilizes the information embedded in Blueprint to speed up the neural compilation. As such, minimizing the information loss about the architectural specifications listed in the data sheets [12] is imperative. Importantly, the Blueprint needs to be designed to have as low overhead as possible. Figure 8 summarizes the design space exploration of Blueprint. Our design of Blueprint strikes balance between the amount of information in the vector ($< 0.5\%$ for minimal information loss in terms of Root-Mean-Squared-Error (RMSE) while using Blueprint) versus the size of the embedding (for fast compilation).
Prior distribution generation with Blueprint. Figure 4 plots the distribution of the initial configurations sampled with and without Blueprint for representative GPU / DNN Model / Layer combinations. The results show that using Blueprint improves the initial configuration. In fact, some layers even reach the optimal configuration within first few steps of optimization, enabling sub-minute compilation time. In contrast, AutoTVM [3] and Chameleon [2] reports that it takes at least few hundred steps (around hour per layer) to reach a similar performance. We also compare against transfer learning which is the core mechanism used in AutoTVM [3] to reuse knowledge from prior optimization runs. We used logs from all but combination of target network and hardware for transfer learning, and plot the output code performance when provided 100 seconds of budget per layer. Figure 5 shows that Blueprint outperforms both AutoTVM with and without transfer learning by 40.0\%. Despite the belief that transfer learning would be sufficient to transfer knowledge among tasks, it sometimes performed worse than baseline AutoTVM. In fact, the results in AutoTVM [3] also suggests that transfer learning only achieves fraction of the final binary performance that are achieved with hundreds to thousands of hardware measurements. These results suggest that knowledge from transfer learning not only necessitates significant number of additional real hardware measurements but also is prone to being misguided. In contrast, Blueprint provides effective initializations to the Glimpse compiler and consistently yields the best performance.

4.2 Hardware-Aware Explorer

Speed of convergence. In AutoTVM [3] and Chameleon [2], authors formulate a cost minimization with a batch of Markov chains and use optimization algorithms such as simulated annealing and reinforcement learning. While the output code performance is determined by the final cost the optimization achieves, the number of updates or steps these Markov chains take is the key factor that determines the optimization time. Figure 6 compares the number of search steps among the three works: AutoTVM [3], Chameleon, and Glimpse. Glimpse achieves 5.07× and 2.55× speed-up against AutoTVM and Chameleon, which shows that Glimpse’s Hardware-Aware Explorer may converge significantly faster than optimizing compilers for single hardware. This notable reduction in the number of search steps come from the Glimpse compiler’s ability to take hints from the mathematical embeddings of the Blueprints about the optimization steps, on when and where to explore and exploit.

4.3 Hardware-Aware Sampling

There is an intrinsic issue of the search space provided by TVM [4] where there exists numerous invalid configurations leading to large delays in compilation speed and waste in GPU hours. In current compilers, around 10% of the measurements made were invalid. Figure 7 presents the reduction in fraction of invalid configurations with respect to the number of hardware measurements for sampling in Chameleon [2] and Glimpse compared to AutoTVM [3]. Glimpse reduces the invalid configurations by 5.56× and 4.53× compared to AutoTVM and Chameleon, respectively. The results suggest, that weak statistical guarantees of the sample synthesis and the adaptive sampling to reduce the frequency of these invalid configurations are insufficient to cope with the above issue. Instead, Hardware-Aware Sampling in Glimpse effectively reduces the number of hardware measurements using the statistical approach.

4.4 Putting It All Together

Figure 9 compares the end-to-end compilation time and the output binary performance of Glimpse compared to state-of-the-art hardware-agnostic techniques: AutoTVM [3], Chameleon [2], and DGP [16]. First, Glimpse cuts down the search time 6.73×, 1.51×, and 1.92× compared to AutoTVM, Chameleon, and DGP respectively, while achieving the best inference latency of the output binary. The gains come from the collaboration of (i) prior distributions generated from Blueprint, (ii) effective balance of exploration-exploitation as well as hardware-awareness of Hardware-Aware Exploration, and (iii) hardware measurements reduction with statistical Hardware-Aware Sampling. Table 2 summarizes the search reduction (GPU time), inference time improvement. Also, following [16], we present Hyper-Volume (HV) to measure the efficacy of different approaches considering multi-objectives.

HV = Search Reduction × Inference Reduction × 100

Glimpse cuts down the search time significantly compared to hardware-agnostic techniques while achieving the fastest inference. Therefore, Glimpse shows the highest HV score: the best trade-off between search time and inference speed. Even if inference speed is the main criterion [16], Glimpse provides the best inference speed.
5 RELATED WORKS

A large body of designing works on neural compilers have been introduced to generate high-performance binaries for innovative neural accelerators [10]. While many neural compilers such as TVM [4] blindly rely on the statistical guarantees of stochastic optimization, this paper uniquely explores the use of hardware blueprints, a proxy of the complete architecture description to improve the initialization, exploration, and the sampling to improve neural compilation. Below, we discuss the most related works:

Neural compilers. While TVM [4] significantly improves inference speed of DNNs, it comes with an intractable search space. AutoTVM [3] develops learned cost models and TenSet [19] provides a large scale dataset to improve the cost models to approximate this large search space. To find optimal configurations, TVM [4] builds on random search and genetic algorithms while AutoTVM [3], GGA [11], and Chameleon [2] explored simulated annealing, guided genetic algorithm, and reinforcement learning to further improve the search efficacy. [16] explored deep Gaussian process to transfer knowledge to different layers on a single target GPU. Prior works were blind about the hardware during optimization, discarding the opportunity to transfer experiences between optimization runs on different hardware. While these blind approaches incur large GPU hours for compilation, this paper explores the use of Blueprint as a mechanism to let compilers perceive the target hardware and predict the search space landscape to expedite the search, reducing the overall GPU hours while also achieving faster inference.

Meta-learning for neural compilation. Meta-learning [5] proposes a mechanism to learn to learn that guides and expedites optimization. For example, MetaBO [17] explored meta-learning in the context of Bayesian optimization for more sample efficient optimization. In the context of neural compilation, MetaTune [14] leverages meta-learning to expedite the convergence of the cost models. In contrast, Glimpse incorporates a unique blend of meta-optimizer that takes domain knowledge about the architectures as input. Specifically, we develop a mechanism that feeds the Hardware-Aware Exploration with information in Blueprint, which led to significant reduction in compilation time as well as the inference latency.

6 CONCLUSION

This paper presents Glimpse, a neural compiler that exclusively explores mathematical embeddings of the hardware Blueprints to improve both the speed and the performance of neural compilation. Experiments on modern DNNs on a multiple generations of hardware shows that hardware-awareness of Glimpse significantly reduces the compilation time while achieving the best inference latency. Encouraging results with Glimpse of Blueprint for neural compilation suggest significant potential in abstractions that encode domain knowledge to improve optimization.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments. We also thank Jaeyoung Kang and Jangho Kim for the constructive discussions and feedbacks on the manuscript. This work was in part supported by generous gifts from Google, Samsung, Qualcomm, Microsoft, Xilinx as well as the National Science Foundation (NSF) awards CCF#2107598, CNS#1822273, National Institute of Health (NIH) award R01EB028350, Defense Advanced Research Project Agency (DARPA) under agreement number HR0011-18-C-0020, and Semiconductor Research Corporation (SRC) award #2021-AH-3039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of Google, Qualcomm, Microsoft, Xilinx, Samsung, NSF, SRC, NIH, DARPA or the U.S. Government.

REFERENCES