Reinforcement Learning for Optimized DNN Compilation

Byung Hoon Ahn¹  Prannoy Pilligundla¹  Hadi Esmaeilzadeh¹

Abstract

Compiler optimizations play a crucial role in maximizing the performance of a program on a target platform. However, many compiler optimization problems are NP-complete or even undecidable, resulting in long optimization times. This issue becomes particularly worse while optimizing deep learning models where it might even take days to optimize a single model. Such long optimization time comes from the need to explore an immense design space of possible configurations and efficient search is crucial to minimizing optimization time. Therefore, in order to improve compiler optimization times, it is crucial to develop an efficient search algorithm.

We use reinforcement learning as our search method to effectively maneuver the design space. Furthermore, we also develop an adaptive sampling algorithm that reduces the number of real hardware measurements during the optimization. We compare our algorithm with state-of-the-art compilation framework, and the results show that our algorithm accelerates the optimization time by 1.89× while showing head-to-head performance of the output code.

1. Introduction

Deep neural networks (DNNs) have been successful in tasks such as image classification (Krizhevsky et al., 2012), automatic speech recognition (Miao et al., 2015), game playing (Mnih et al., 2015), device placement optimization (Mirhoseini et al., 2017) etc. DNNs are multiple blocks of large matrix multiplication and high dimensional convolutions stacked together to perform their given tasks. Therefore, despite DNN’s impressive performance, their huge computational intensity is a major challenge against further applications and deployment of DNNs.

Some efforts have been made to address the issue by taking advantage of the algorithmic characteristics of DNNs. For example, manually optimized libraries like cuDNN, MKL, and NNPACK have been developed so that machine learning researchers can implement and train DNNs more easily. Furthermore, such libraries have facilitated the development of machine learning frameworks (Abadi et al., 2015; Jia et al., 2014; Paszke et al., 2017) to further simplify implementation steps for DNNs.

However, such frameworks are limited to improving usability and less concerned about optimization of these DNNs on multitude of hardware platforms, such as ASICs, and FPGAs. More importantly, these machine learning frameworks are limited in their ability to optimize newly developed compute blocks that are crucial in improving the performance of DNNs.

Deep learning compilers such as TensorComprehensions (Vasilache et al., 2018), TVM (Chen et al., 2018a) have been proposed to tackle this issue. For new DNN models, each layer with unique shapes need to be optimized individually for a given hardware platform. Moreover, design space of possible configurations for the output code of every layer is so large that it takes hours or even days to optimize a single layer. In order to understand the impact of different parameters during optimization, different configurations are run on physical hardware to get real runtime measurements. However, applying a brute force search is infeasible due to the enormous design space. Instead, these frameworks make use of genetic algorithms or random search algorithms to reduce the optimization time, but cost of hardware measurement is still a major bottleneck.

Therefore, there have been efforts to employ machine learning based cost models to mitigate the cost of real hardware measurement. AutoTVM (Chen et al., 2018b), which uses a gradient boosting based cost model (Chen & Guestrin, 2016) along with simulated annealing for exploration has been shown to perform better than the aforementioned algorithms. However, optimization times of the current state-of-the-art still leaves room for further improvement.

All in all, we notice that such slow optimization speed comes from (1) inefficient search algorithm, and (2) the number of times compiler reaches for real hardware measurements. We refer to this optimizing part of the compiler as optimiz-
2. Background

2.1. Compilation Workflow

Figure 1 illustrates a common compilation workflow, where the compiler takes a program as an input and emits a code as an output.

Target-independent optimization. First stage of the compilation is target-independent optimization, where the input program is transformed regardless of the hardware. For example, dead-code elimination or loop-invariant code motion in LLVM (Lattner & Adve, 2004) and operator fusion in TVM (Chen et al., 2018a) are some examples of this.

Target-dependent optimization. Second stage of the compilation is target-dependent optimization. This is where the compiler takes the hardware architecture (target) into account while optimizing the program, and this can be integrated with machine code generation to emit the final code.

2.2. Optimizing Compiler

Some compiler frameworks (Vasilache et al., 2018; Chen et al., 2018a) choose to further optimize program by formulating compiler optimization into a black box optimization problem and try to solve it using an optimizing compiler, as exemplified by Figure 1.

However, meeting such objective is very difficult because of infinite possibilities of output code. In order to make this problem more tractable, output code is formulated into templates with tuning knobs. Each combinations of knobs’ options is defined as a configuration and entire population of them make up the design space.

Optimizing compiler receives such code template and design space for each layer of the network, and makes use of a search algorithm to efficiently find the best configuration within the design space. In this context, there are two variables that determine the performance of the optimizing compiler: (1) large enough design space that covers variety of optimizations, and (2) effective search algorithm to sweep the space.

Design space. As stated above, design space of optimizing compiler should be large enough to encompass various optimization techniques that takes architectural characteristics into account. For example in GPUs, it is crucial that the code (1) maximizes data reuse, (2) uses the shared memory wisely, and (3) minimizes bank conflicts.

For instance, while optimizing convolution for GPUs, there...
are several dimensions of optimization which are exemplified in Table 1. *tile_* are used to hint code generator with factors for tiling and binding such as size of the block and scheduling of threads. *_unroll_* are used to hint the code generator about unrolling the code to maximize parallelism. Combinations of these dimensions could be as large as $10^{10}$.

**Search algorithm.** One way to search the aforementioned design space is to do a brute force search. However, when the design space could be so large ($\sim 10^{10}$), such method become too time-consuming.

One strategy is to incorporate an exploration module that would intelligently search for configurations that would yield good performance. In each iteration of optimization, such module makes several predictions for configurations and gather runtime information from hardware to improve subsequent predictions. After a certain number of iterations, compiler emits the final optimized code by choosing the best configuration from the history.

Some frameworks (Vasilache et al., 2018; Chen et al., 2018a) choose to integrate random search or genetic algorithms to efficiently search this large design space. On the other hand, state-of-the-art algorithms employ machine learning based cost models to estimate performance of the output code and use simulated annealing to make predictions on top of the information from the cost model.

### 3. Motivation and Design Goals

#### 3.1. Motivation

Even with the techniques discussed in Section 2.2 to speed up DNN compilation, it takes nearly a whole day to optimize a network with only 12 layers, as illustrated in Figure 2. This gets worse for deeper networks which have over 50 convolution layers. This could hinder getting timely feedback about performance in agile production cycles. Such long compilation time comes from an inefficient search algorithm and the number of times the optimizing compiler reaches out for real hardware measurements.

All in all, in such frameworks where compilation is formulated into an optimization problem, problems can be divided into two subproblems: (1) developing an efficient search algorithm (2) reducing the number of times the compiler reaches for real hardware measurements.

#### 3.2. Design Goals

**Efficient search algorithm.** Due to the immense design space of configurations, it is simply impossible to enumerate all possible combinations. Therefore, efficient search technique is crucial in finding good configuration in a reasonable amount of time.

Some works (Chen et al., 2018a; Shen, 2009; Mei et al., 2002) have used simulated annealing in the context of compiler optimization problems. Since simulated annealing statistically guarantees finding an optimal solution, previous work (Chen et al., 2018a) finds configurations that extract reasonable computational efficiency from the hardware. However, simulated annealing is known for its slow speed and could be an overkill, especially for problems where the objective function landscape is smooth.

In this work, we explore a new possibility using reinforcement learning that strikes a good balance between exploration and exploitation during the search. In the rest of the paper, we call this the search agent.

**Reducing the number of hardware measurements.** Figure 2 presents the total and the breakdown of the time it takes to optimize convolution layers of ResNet-18. It is clear from the graph that majority of the compile time is spent on reaching for measurements on hardware. Reducing the number of measurements on hardware will be more effective in reducing the overall optimization time.

There are two different issues to address in terms of pick-
In this work, we leverage this observation and methodically sample representative configurations from the distribution of configurations from the search agent to make our compiler make less hardware measurements without compromising the quality of compilation.

4. ReLEASE: Reinforcement Learning Compiler with Adaptive Sampling for Efficiency

As discussed in Section 3, there are two distinct yet interrelated issues that have to be addressed for high-performance yet faster compilation. We propose ReLEASE\(^1\), reinforcement learning based optimizing compiler to solve this problem. Figure 4 illustrates the framework and its modules.

Input to ReLEASE are (1) code template which has information about layers of the DNN and (2) design space specification. ReLEASE uses a machine learning based cost model to approximate the design space, and performs search using reinforcement learning (RL)-based search agent which returns a trajectory. Adaptive sampling module takes the trajectory and adaptively samples configurations to minimize number of real hardware measurements. Then these configurations are measured on real hardware and their runtime information is used to train the cost model.

In ReLEASE, we make two major design choices. First, to find good trade-off between exploration and exploitation, we employ reinforcement learning to our search agent. Also, as a means to reduce number of hardware measurements without compromising performance of output code, we use an adaptive sampling algorithm. Rest of the section explains the details of design choices made for each component.

4.1. Reinforcement Learning based Search Agent

The goal of the search agent is to search for potential configurations. We make use of reinforcement learning to ensure that our agent quickly finds the set of good potential configurations.

In this work, we specifically employ Proximal Policy Optimization (PPO) (Schulman et al., 2017) as our learning algorithm. Figure 5 depicts the overall formulation of our search agent. Details of the hyperparameters used can be seen in Table 2.

**State space.** As shown in Table 1, there are several factors that contribute to the performance of the generated code. Each of the terms under *tile_* and *_unroll_* are different dimensions for optimization. Therefore, the search agent, or the reinforcement learning agent, needs to learn about

---

\(^1\) ReLEASE: Reinforcement Learning Compiler with Adaptive Sampling for Efficiency
the dependencies between all the dimensions of the configuration design space in order to reach optimal overall configuration. Therefore, the state space contains the current values for all dimensions of the current configuration. To be more specific, state input for the search agent is a vector of current values for all the dimensions.

**Action space.** The agent needs to be able to traverse through the configuration design space. Therefore, we define the action space of the agent as the vector of direction for each dimension of the configuration. For every dimension, the direction is either to increase or decrease the current value or stay at the current value. For every step of the search, our agent aims to take steps towards the optimal configuration.

**Reward formulation.** Reward in RELEASE context is the performance of the output code. However, since the real hardware measurement is very costly in our scenario as discussed in previous sections, we use the estimation from the cost model as a surrogate reward. As shown in Figure 4, our search agent makes queries to the cost model after each episode of search.

**Policy and value networks.** Our search agent uses a policy gradient based approach, Proximal Policy Optimization (PPO). PPO is based on a actor critic style algorithm so our search agent consists of both Policy and Value networks. The search agent's first layer is shared to foster information sharing among the two networks. Output of the first layer is fed into both networks, then policy network returns vector of directions for each dimension in configuration and value network returns the value of the action.

**Learning procedure.** The whole procedure begins with a set of initial configurations. As shown in Figure 5, for a given input configuration, the agent takes an action and the configuration updater performs that action on the input configuration which leads to the next configuration and it becomes the input for the next action. Agent takes fixed number of actions in each episode and at the end of each episode, the entire trajectory of configurations are evaluated using the trained cost model. Agent then formulates the return values of the cost model as reward and trains the policy and value networks. During this process, the agent learns to distinguish good configurations from the bad and understand the interplay between different dimensions on the input in order to predict good configurations. The agent repeats the process for another configuration.

After the whole process, the agent returns a trajectory of configurations that is fed into our adaptive sampling module as a population of configurations to sample from.

**Algorithm 1 Adaptive Sampling Strategy**

| Input: search trajectory $S$, visited $V$ |
| Output: NextSamples |
| NextSamples = ∅ |
| PreviousLoss = $\infty$ |
| for $k$ in range(8, 64) do |
| Centroids, Clusters, Loss = K-means.run($S, k$) |
| if Loss > PreviousLoss then |
| Break |
| else |
| PreviousLoss = Loss |
| end if |
| end for |
| for $c$ in Centroids do |
| if $c$ in $V$ then |
| new_c = mode($S$) |
| NextSamples.append(new_c) |
| Continue |
| else |
| NextSamples.append($c$) |
| end if |
| end for |

### 4.2. Adaptive Sampling Module

We notice that hardware measurements are costly and prolong the optimization time significantly. Therefore, we propose an algorithm that adaptively samples configurations to reduce the number of real hardware measurements.

**Adaptive sampling strategy.** We illustrate our adaptive sampling strategy in Algorithm 1. After every episode, our sampling module takes the search trajectory of the search agent for that episode and uses K-means clustering algorithm to determine centroids for all the configurations seen until that point. We iterate through various number of clusters $k$ until it hits a good trade-off point between reducing hardware measurements and reducing loss of the clustering algorithm.

The sampling module checks the history to sift out previously visited configurations from the centroids and selects configurations which further enhance the cost model. Finally, sampled configurations are passed onto code generator to be run on hardware and the resulting runtimes are used to update the cost model.

**Clustering algorithm.** Clustering algorithms such as K-means (Lloyd, 2006) are notorious for their complexity. For example, K-means has a complexity of $O(nKId)$, where $n$, $K$, $I$, and $d$ are number of points, clusters, iterations, and attributes, respectively. Furthermore, determining number
of clusters $k$ is a hyperparameter that adds an extra degree of freedom which is, in our case, a complexity. Since we are dealing with tractable amount of data in this case, K-means doesn’t act like the limiting factor and it is in fact effective in reducing the overall optimization time, which we present in the next section.

5. Evaluation

We integrate our optimizing compiler into TVM (Chen et al., 2018a) as our baseline framework to evaluate the performance. We first evaluate each component of RELEASE in Section 5.2, then layerwise, end-to-end evaluation of runtime and performance in Section 5.3.

5.1. Experimental Setup

We use a set of six convolution layers from two networks: VGG-16 (Simonyan & Zisserman, 2014) and ResNet-18 (He et al., 2016) as our baseline for per layer evaluation, and use VGG-16 and ResNet-18 for ImageNet (Deng et al., 2009) classification during end-to-end evaluation. The details of these models and layers are presented in Table 3 and Table 4, respectively.

We compare the performance of RELEASE on NVIDIA Titan Xp as our GPU with 3.4 GHz Intel Core i7 as the host processor and 32 GB 2400 MHz DDR3 as the main memory.

5.2. Component Evaluation

In this section, we evaluate each component within RELEASE using the set of layers presented in Table 4.

Reinforcement learning based search agent. In the previous approach (Chen et al., 2018a), authors have built a cost model to mitigate the cost of measuring on real hardware then used simulated annealing to search for potentially optimal configurations. In Figure 6, we compare the number of search steps taken per iteration that was used to converge to a good configuration.

Overall, we can see that our reinforcement learning agent requires 2.64 × less search steps compared to simulated annealing to converge. We believe that reinforcement learning quickly learns about the correlation between different dimensions. This allows the search agent to quickly locate the optimal solution. With this observation, we can infer that our reinforcement learning agent makes more effective search over the design space.

Adaptive sampling module. We apply our adaptive sampling algorithm to both the baseline framework and RELEASE. In Figure 7, we summarize the effect of using adaptive sampling module, on both simulated annealing and reinforcement learning based search.

First, the results show that using adaptive sampling helps the framework to make less hardware measurements regardless of the search strategy used. Adaptive sampling reduces the number of measurements by 2.28 × when used with simulated annealing and 2.81 × with reinforcement learning.

Importantly, we see that the adaptive sampling is more effective with reinforcement learning search. This comes from the reinforcement learning agent’s ability to make more effective search leading to more precise cost model. In the

Table 3. Details of the DNN models used in evaluating RELEASE.

<table>
<thead>
<tr>
<th>NETWORK</th>
<th>DATASET</th>
<th>NUMBER OF TASKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16</td>
<td>ImageNet</td>
<td>9</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>ImageNet</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 4. Details of the layers used in evaluating RELEASE.

<table>
<thead>
<tr>
<th>NAME</th>
<th>MODEL</th>
<th>LAYER TYPE</th>
<th>TASK INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>VGG-16</td>
<td>convolution</td>
<td>1</td>
</tr>
<tr>
<td>L2</td>
<td>VGG-16</td>
<td>convolution</td>
<td>2</td>
</tr>
<tr>
<td>L3</td>
<td>VGG-16</td>
<td>convolution</td>
<td>4</td>
</tr>
<tr>
<td>L4</td>
<td>ResNet-18</td>
<td>convolution</td>
<td>6</td>
</tr>
<tr>
<td>L5</td>
<td>ResNet-18</td>
<td>convolution</td>
<td>9</td>
</tr>
<tr>
<td>L6</td>
<td>ResNet-18</td>
<td>convolution</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 6. Comparison of number of search steps for simulated annealing (TVM) and reinforcement learning (RELEASE).

Figure 7. Comparison of number of samples for simulated annealing (SA) in TVM, SA with adaptive sampling, and reinforcement learning (RL) with adaptive sampling (RELEASE).
Figure 8. Trend of output code performance of ResNet-18’s 11th layer over number of hardware measurements made for simulated annealing (TVM) and reinforcement learning with adaptive sampling (ReLeASE).

long run, this acts favorably on the sampling procedure, reducing the number of measurements.

5.3. Runtime and Performance Evaluation

In this section, we evaluate end-to-end performance of ReLeASE. We first run set of layers presented in Table 4, then we run set of end-to-end networks presented in Table 3.

Layer evaluation. ReLeASE integrates two components into the workflow: reinforcement learning based search agent and adaptive sampling module. We compare end-to-end performance of ReLeASE with the baseline on the sampled layers.

Figure 8 shows the trend of output code performance of ResNet-18’s 11th layer over number of hardware measurements during optimization. First, we can see that our reinforcement learning search agent finds better configurations, regardless of sampling module’s presence. Also, we can see that our optimizing compiler reduces the number of hardware measurements during optimization. With ReLeASE’s reinforcement learning search agent and adaptive sampling working in tandem emits better code with shorter optimization time than any other combination.

In Figure 9, we compare the wall-clock optimization time and the performance of the output code in ReLeASE and TVM. ReLeASE achieved 1.03× better performance with 1.76× shorter optimization time compared to our baseline framework, TVM.

Overall, the results suggest that the reinforcement learning agent makes effective search over the design space, and adaptive sampling reduces hardware measurements and overall optimization time without loss of output code performance.

End-to-end evaluation. Up until now, we have focused on component specific evaluation with subset of layers. Now we continue our discussion to the applicability of this system to end-to-end optimization of entire DNN model.

The results presented in Figure 10 show that our compiler spends 1.97× less time than TVM to optimize VGG-16, and 1.82× for ResNet-18. On average, our work shows 1.89× optimization time speedup while giving head-to-head performance of output code. Inference time in Figure 10 (b) specifies the inference time for optimized code and it can be seen that our compiler outperforms both cuDNN and TensorFlow. Such improvements result from more efficient search algorithm and the reduced number of hardware measurements.

5.4. Limitations

Hyperparameter tuning and heuristics. ReLeASE introduces several hyperparameters and heuristics embedded in the implementation. For example, various numbers presented in Table 2 for the search agent, number of iterations and number of dimensions used for distance measure in K-means algorithm of adaptive sampling module are hyperparameters. Furthermore, we implement a heuristic to determine $k$ for the K-means algorithm.
For this work, we determine the values and design through numerous experiments. We believe our work presents an initial point to build upon. Therefore, we leave exploration of hyperparameter tuning (Bergstra et al., 2011), application of different clustering algorithm (Xu & Wunsch, 2005), and heuristic development to future work.

**Speedup of optimization.** Ideal overall speedup of optimization can be determined by Amdahl’s law suited to our case:

$$\text{Overall Speedup} = \frac{1}{(1 - p_{exp} - p_{hw}) + \frac{p_{exp}}{S_{exp}} + \frac{p_{hw}}{S_{hw}}}$$

$p_{exp}$ is the fraction of time spent on search and $p_{hw}$ is the fraction of time spent on hardware measurement. Likewise, $S_{exp}$ is the speedup of time spent on search and $S_{hw}$ is the reduction of hardware measurement.

According to results in Figure 2, hardware measurements account for around 85% of the optimization time. With this information, we can infer that, ideally, we should be able to achieve 2.78× speedup. However, various overhead such as fluctuation of inference and training speed of the cost model seems to be an obstacle. We plan to optimize this in the future work.

6. Related Works

**Automated optimization.** Some computing libraries (Whaley & Dongarra, 1998; Frigo & Johnson, 1998) make use of black box optimization. Both TVM (Chen et al., 2018a) and Tensor Comprehensions (Vasilache et al., 2018) use automated optimization to choose parameters of polyhedral optimization. TVM uses machine learning based cost model and simulated annealing for exploration, and Tensor Comprehensions (Vasilache et al., 2018) use genetic algorithm for their search algorithm.

On the other hand, we use reinforcement learning and active learning to boost the efficacy of search and augment sample efficiency. Furthermore, our optimization framework is agnostic to the base compiler framework and can be applied to any pre-existing framework.

**Reinforcement learning for optimization.** There are a growing body of studies on using reinforcement learning to perform various optimizations. HAQ (Wang et al., 2018) and ReLeQ (Elthakeb et al., 2018) use reinforcement learning to perform deep neural network compression. DeepArchitect and NAS (Negrinho & Gordon, 2017; Pham et al., 2018) use reinforcement learning to automate the process of designing deep neural network models and their associated parameters. Alternatively, reinforcement learning has been applied to device placement optimization (Mirhoseini et al., 2017).

While our work takes inspiration from prior work on using reinforcement learning to optimize computation, it explores a fundamentally a different optimization problem.

7. Conclusion

In this work, we proposed an optimizing compiler dubbed RELeASE to accelerate DNN optimization. We used reinforcement learning to search the design space more effectively and furthermore, we employed an adaptive sampling algorithm to reduce the number of hardware measurements without sacrificing performance of compilation.

We compared our work with state-of-the-art compiler framework for DNN compilation, and showed that our compiler is significantly faster while providing head-to-head performance against state-of-the-art frameworks. We see that speed-up comes directly from the effective search and the efficacy of adaptive sampling in reducing number of hardware measurements.

This promising result suggests potential employment of reinforcement learning in compiler optimization. In future work, we will employ more effective sampling schemes and extend this work to various platforms. Furthermore, we hope to apply this insight to generic compiler optimization problems.

References


Reinforcement Learning for Optimized DNN Compilation


Shen, Y. Tuning compiler optimization options via simulated annealing. 01 2009.


