AdaTune: Adaptive Tensor Program Compilation Made Efficient

Menghao Li[*] **Minjia Zhang**[*] **Chi Wang** **Mingqin Li**
Microsoft Corporation
`{t-meli,minjiaz,wang.chi,mingqli}@microsoft.com`

Abstract

Deep learning models are computationally intense, and implementations often have to be highly optimized by experts or hardware vendors to be usable in practice. The DL compiler, together with *Learning-to-Compile* has proven to be a powerful technique for optimizing tensor programs. However, a limitation of this approach is that it still suffers from unbearably long overall optimization time. In this paper, we present a new method, called AdaTune, that significantly reduces the optimization time of tensor programs for high-performance deep learning inference. In particular, we propose an adaptive evaluation method that statistically early terminates a costly hardware measurement without losing much accuracy. We further devise a surrogate model with uncertainty quantification that allows the optimization to adapt to hardware and model heterogeneity better. Finally, we introduce a contextual optimizer that provides adaptive control of the exploration and exploitation to improve the transformation space searching effectiveness. We evaluate and compare the levels of optimization obtained by AutoTVM, a state-of-the-art Learning-to-Compile technique on top of TVM, and AdaTune. The experiment results show that AdaTune obtains up to 115% higher GFLOPS than the baseline under the same optimization time budget. Furthermore, AdaTune provides 1.3–3.9× speedup in optimization time over the baseline to reach the same optimization quality for a range of models across different hardware architectures.

1 Introduction

The enormous computational intensity of Deep Neural Network (DNN) models has attracted great interest in optimizing their performance. In particular, popular deep learning (DL) frameworks such as TensorFlow [6] and PyTorch [32] adopt custom optimized kernels such as Nvidia cuDNN [15] or Intel MKL-DNN [2] as back-end. However, given the increasing complexity of tensor operations in DNNs and the volatility of DL algorithms, it calls for developing fast and automated compilation frameworks to handle the unprecedented amount of innovations. To imitate or even surpass the success of hand-optimized libraries, recent research has developed neural network compilers, such as XLA [4], Halide [36], Glow [37], Tensor Comprehension [40], and TVM [13]. Among them, TVM has shown superior performance improvements using a technique called *Learning-to-Compile* (AutoTVM) [14]. AutoTVM optimizes the code by generating many versions of a tensor program and chooses the best through simulated annealing search over a large space of code transformation choices. Furthermore, it employs a learned cost model trained by actual hardware performance measures to predict the performance of diverse inference computations on real hardware targets.

While the *Learning-to-Compile* approach produces highly optimized code of DNN models, they may suffer from excessively long optimization time. As an example, although AutoTVM is able to demonstrate close to 2× performance improvement over TensorFlow on ResNet-18, the compilation time can still take several hours or even tens of hours [14]. With the active research that has been pushing the model size to millions or even billion-scale parameters with a training time of only a

[*]Both authors contributed equally. Order of appearance is random.

few hours or less than one hour [46, 19, 47, 29, 38, 48], it makes reducing the DL compilation time for inference of the current solution even more prominent. Furthermore, since many of these DL compilers have been adopted by major players in the industry [43, 44, 26, 30], many users of these pipelines, including deployment engineers, would have to go through the optimization numerous times. Finally, as new neural architectures [18, 41, 16, 34] come out in rapid speed, and with deeper or wider networks [45, 38, 35, 5] on various hardware platforms [3, 22, 28], we are forced to optimize the networks more frequently. The excessive long optimization time hinders the turnaround time and even puts the practical utility of the current compiler-based solutions into question.

We aim at accelerating innovations by developing an automatic and efficient optimization process for DNN models. For this purpose, we introduce AdaTune, a method that achieves similar or better optimization quality but with shorter optimization time. Furthermore, AdaTune improves the adaptivity of LTC and reduces the hyperparameter tuning required, accelerating the productivity and agility of DNN model deployment. Specifically, the contributions of our paper consist of (1) a preliminary analysis that reveals the inefficiency and challenges of the existing approaches, (2) an *adaptive evaluator* that statistically determines the number of runs for performance measurement, (3) a *surrogate modeling with uncertainty quantification* that allows to better capture hardware and model heterogeneity, and (4) a *contextual optimizer* that provides control of exploration and exploitation dynamically to improve the transformation space searching effectiveness. We conduct extensive experiments to show that the proposed approach consistently outperforms the previous method on various models and hardware. It not only allows us to optimize DNN models 1.3-3.9$\times$ faster than the baseline to reach the same optimization quality but also obtains up to 115% higher GFLOPS under the same time budget. We conduct ablation analysis to study the effects of the proposed techniques, and we will make the source code publicly accessible to encourage further research.

2 Background and Related Work

DL compilation pipeline. DL compilers like TVM [13] have recently become popular for automatically optimizing DL programs [37, 4, 36, 40]. A typical DL compiler contains multiple passes to optimize a model trained by popular DL frameworks such as TensorFlow [6], PyTorch [32], or MXNET [12], as shown in Fig. 1. In the first pass (box with dotted line), the compiler frontend applies target-independent and white-box target-dependent optimizations that do not include a measure of actual execution time. The target-independent passes perform optimizations such as operator fusion and data layout transformation and the while-box target-dependent optimizations apply heuristic rules for code transformation based on domain knowledge, both of which do not need a specification of the target hardware. Recent work such as AutoTVM [14] extends the pipeline with another pass, which is a black-box target-dependent pass, which uses learning machinery to perform optimizations.

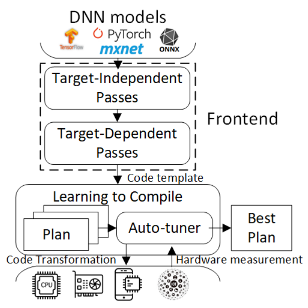


Figure 1: DL compilation pipeline.

Table 1: Example of TVM knobs.

Knobs	Definition	Example values
tile_f	Loop tile decisions on the number of filters, and height and weight of feature maps	84
tile_y		140
tile_x		140
tile_rc	Loop tile reduction decision on the number of channels, height and weight of filters	2
tile_ry		2
tile_rx		2
auto_unroll max_step	The threshold of iterations a loop to be unrolled	3
unroll_explicit	Explicitly unroll loop	2

Black-box target-dependent pass. In this pass, the compiler converts code transformation decisions as code templates, and it makes use of an auto-tuner (with optimization algorithm) and real hardware measurements to efficiently find the best transformation on target hardware (e.g., CPU, GPU, ARM, or IoT devices). Table 1 lists the knobs for code transformation of a convolution block on GPUs as example , which control various aspects of the optimization and determine whether the code (1) fully utilizes the internal parallelism within processors, (2) uses the shared memory wisely, and (3) maximizes data locality. Auto-tuning has been studied for generic program compilation using domain-specific search techniques [9, 10, 42, 7]. Later, [14] builds on top of prior work by using a cost model and simulated annealing to search the transformation space for DNN models. However, there are certain limitations, as described in Section 3. Subsequently, [8] proposes to use reinforcement learning for efficient search space exploration. However, their approach involves a non-trivial amount of additional hyperparameter tuning as well as additional domain-knowledge on the validity of potential solutions on specific hardware.

Problem formulation. Given the vastness of the transformation space, if we denote the space as S_e and function $Perf(\cdot)$ as the performance (i.e., GFLOPS – Giga floating point operations per second) of one transformation plan p on a given hardware target h, the goal of the black-box target-dependent pass is to find a transformation plan $p*$ in S_e that maximizes $Perf(\cdot)$ on h over S_e efficiently.

3 Preliminary Analysis

This section presents several studies that guided the design of the approach introduced in Section 4.

Observation 1. DL compiler generates highly optimized code but results in prolonged optimization time. Prior work [13, 14] uses an auto-tuner to select a plan p from the transformation space, calls a code generator as a subroutine to generate machine code for that plan on a specific hardware, and then executes the generated code on the target hardware n times to obtain an estimated cost of p as $avg(p) = \frac{1}{n}\sum_{i=1}^{n} Perf(p, q_i)$, where q_i represents an input of inference. Existing methods require the number of repeats n as an input. However, for a fixed n, it is not clear a priori whether we should (a) run many repeats (large n) with more accurate measurement but also large measurement cost; or (b) consider a small number of repeats (small n) with inaccurate measurement but small measurement cost. n cannot be too small, because then even if a transformation plan with low *average* cost is chosen, its *variance*, $std(p) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Perf(p, q_i) - avg(p))^2}$ can be high. This is undesirable because then the measured average cost could deviate from its true performance due to the variance in the measurement, and we may end up choosing a suboptimal plan. Therefore, the general practice is to use a large conservative number for n to achieve accurate estimates of the true cost, which is why it takes a very long time for existing auto-tuning methods to find a good solution. Figure 2 shows the execution time (on CPU) breakdown on 12 tasks from ResNet-18 [20]. As can be seen, the majority of time is spent on hardware measurement. Therefore, it is important to cut down the expensive cost of hardware measurements that examine the goodness of a plan.

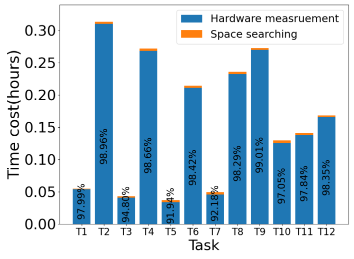
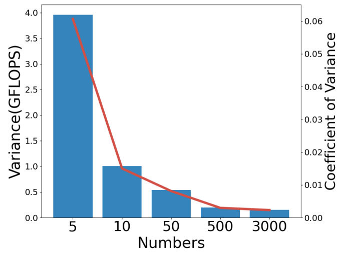
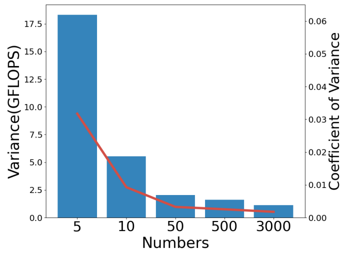


Figure 2: Performance breakdown of AutoTVM on ResNet-18.



(a) Intel Xeon x86 CPU E5-2690 v3

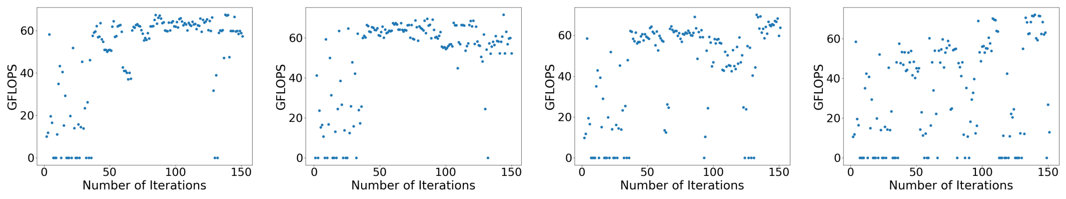


(b) Nvidia Tesla P100

Figure 3: Performance measurement variance and coefficient-of-variation on different hardware.

Observation 2. Existing methods perform measurement without considering model diversity and hardware heterogeneity. Prior study [14] claims that considering variance does not help. Therefore, they use a regression model (e.g., XGBoost) to learn and predict the performance of a transformation plan. Different from their observations, we find that in practice, based on the scenario, a model might need to be optimized against different hardware, including x86 CPU [43, 49], GPUs [21], ARM, and various ML accelerators [31, 25], all of which have very different architectures (e.g., memory hierarchy, instruction level parallelism, hardware prefetching, branch prediction, etc), which appear to have very different variance behaviors. Figure 3a and Figure 3b show the mean-variance range under x86 CPU and GPU for different n. As n increases, the variance (bar) decreases as expected. Although the absolute variance on GPU is higher than that on CPU, the coefficient of variation (curve) on CPU is much larger than that on GPU, especially when n is relatively small. Furthermore, different models may also have diverse execution patterns, e.g., small/large model, regular/irregular computations, inter-op/intra-op parallelism, data dependencies, all of which can affect the run-to-run variance. Since existing work assumes the uncertainty is low by using a large n, if we reduce n, it is more likely that a regression model without accounting for the uncertainty can lead to suboptimal trials.

Observation 3. Existing approaches employ static exploration-vs-exploitation, which limits the adaptivity of the compiler optimization. Existing works [13, 14] employ the genetic algorithm or simulated-annealing to guide the transformation space searching process. To balance exploration and exploitation, they apply $\epsilon-$greedy in each searching iteration by selecting ϵb candidates randomly to ensure exploration, where b is the batch size and ϵ is a fixed value 0.05. This decision appears suboptimal for several practical reasons. As shown in Fig. 4, a small ϵ (e.g., 0.01, 0.05) tends to be overly greedy, as it focuses in an area where the model believes the optimum to be, without efficiently exploring additional areas of the transformation space which may turn out to be more optimal in the long run. On the other hand, a large ϵ (e.g., 0.2, 0.5) induces a large distraction into the searching process and slows down the searching process. However, having a constant ϵ, determined at the start of the beginning, introduces an additional hyperparameter that needs to be tuned. Furthermore, even with a tuned ϵ, its value is going to remain the same during the entire optimization for all tasks, which is sub-optimal. The effectiveness of the existing approach is, therefore, significantly affected by the degree of trade-off between exploration and exploitation. Given that the general goal is to make the optimization fast and more usable, this is clearly not desirable. Thus, we need a more principled way to control the balance between exploration and exploitation.



(a) $\epsilon = 0.01$ (b) $\epsilon = 0.05$ (c) $\epsilon = 0.2$ (d) $\epsilon = 0.5$

Figure 4: Exploration-vs-exploitation: choice of ϵ in $epsilon$-greedy.

4 The Elements of AdaTune

In this section, we present the design decision for AdaTune, and we provide quantified comparisons against corresponding configurations of the original AutoTVM [14] in Section 5. We illustrate the high-level design of AdaTune in Figure 5. There are three main contributions that AdaTune makes over the design choices of AutoTVM.

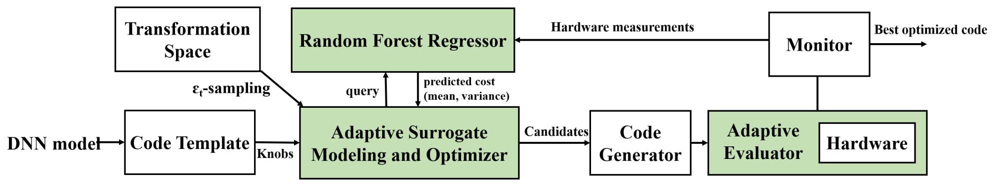


Figure 5: Overall design and optimization overview of AdaTune.

4.1 Adaptive Evaluator: Early Termination by Coefficient of Variation Counting

The adaptive evaluator (AE) is the module in charge of steering the dynamic reconfiguration process of batch measurement for getting the performance. The key challenge in the design of this component is to gather measurements in an accurate and timely way, so as to maximize accuracy (i.e., reduce variance) and minimize the cost for different hardware measurements.

In AdaTune, we use AE to automatically adjust the iterations of measurements n in a model diversity and hardware heterogeneity aware way. This adaptive mechanism is based on the idea of estimating the statistical uncertainty associated with the current performance (e.g., GFLOPS) measurement on the basis of coefficient of variation (CV). More precisely, for a given n, we divide it into *micro-batches* of size B, and we evaluate the performance upon the finish of each micro-batch since the beginning of the hardware measurement (time = 0). If we denote $Time(i)$ ($i \in \{1, 2, ..., B\}$) as the time elapsed since the beginning of the measurement of plan p and the occurrence of the i-th micro-batch, the GFLOPS upon the i-th micro-batch would be $Perf(p)_i = \frac{i \times GFLOP(op(p))}{Time(i)}$. We then use it to estimate the accuracy of the measurement after i micro-batches with the CV of $Perf(p)_i$, i.e., CV($Perf(p)_i$) = $\frac{std(\{Perf(p)_1, Perf(p)_2, ..., Perf(p)_i\})}{avg(\{Perf(p)_1, Perf(p)_2, ..., Perf(p)_i\})}$. If the CV value is smaller than a certain threshold (e.g., 10%), AdaTune adaptively terminates a measurement. AE, therefore, automatically adjusts the hardware measurement costs in a robust and model/hardware-independent way.

Prior studies on hyperparameter tuning such as Hyperband [27] and BOHB [17] define approximate versions of the objective function (e.g., classification tasks) that are parameterized by a concept called budget. They prioritize promising configurations with larger budgets as the optimization progresses. Our approach is similar to the budget concept in the sense that we also create a cheap-to-evaluate version of the hardware measurement function. However, different from their goal, which is to eventually optimize with the largest budget, we collect a stream of measurements of micro-batches and timely stops when the likelihood of getting very different measurement results is low. [8] uses so-called adaptive sampling by performing non-uniform sampling through clustering. Different from their method, our approach cuts the cost of hardware measurement of individual samples. The two methods can be combined to maximize the gains.

4.2 Adaptive Surrogate Modeling and Optimizer

For AdaTune, we propose another two improvements: (1) We create a *surrogate model with uncertainty quantification*, which takes both mean and variance into consideration to adapt performance modeling and drives the exploration of the transformation space by continuously gathering feedback on the quality of the explored transformation plans; (2) We introduce the *contextual simulated annealing* optimizer, which dynamically balances the trade-off between exploration and exploitation based on the expected improvement from the surrogate model.

4.2.1 Surrogate modeling with uncertainty quantification

Given the very different variance behaviors (Section 3), in order to make the optimization process more adaptive to different hardware and models, we consider constructing a surrogate model by accounting for uncertainty. In particular, we consider an ensemble model $\tilde{f}$ of black-box learners, each of which is built on m measurements randomly sampled with repetitions from the entire hardware measurements $\{(p_1, Perf(p_1)), ..., (p_m, Perf(p_m))\}$, where a transformation plan $p_i = (p_{i,1}, ..., p_{i,d})$ is a complete instantiation of the code template's d knobs. Given a new plan p_{m+1}, the model predicts the mean μ and variance σ for its performance $\tilde{f}(p_{m+1})$ through the ensemble.

Among many options, a useful tool for quantifying the uncertainty in a given prediction is random forest, which has proven to be valuable in Sequential Model-based Bayesian Optimization (SMBO) [11] method such as *SMAC* [23]. We choose random forest as our surrogate model because it enjoys advantages, such as better handling of discrete features. It also has a training time complexity of $O(m \cdot log(m) \cdot d \cdot H)$ where H is the number of decision trees, which is more efficient than models like Gaussian Processes, which exhibit $O(m^3)$ training complexity in the number of data points (see comparison results in Appendix B).

Expected positive improvement. We use the same diversity-aware cost function as the one used in AutoTVM [14] to select a list of promising plans for hardware measurements. In particular, we replace the run time cost estimate part with *expected positive improvement* (EI) and keep the other term unchanged. We compute $EI(p) = \mathbb{E}[max(\tilde{f}(p) - Perf(p^*), 0)]$ [11] over the best known measured performance $Perf(p^*)$ so far (p^* is the best plan so far), while taking into account the possible uncertainty in that prediction. Given the predictive mean μ and standard deviation σ of a plan p, we have:

$$EI(p) = \begin{cases} (\mu(\tilde{f}(p)) - Perf(p^*))\Phi(Z) + \sigma(\tilde{f}(p))\phi(Z) & if\ \sigma(\tilde{f}(p)) > 0 \\ max(0, \mu(\tilde{f}(p)) - Perf(p^*)) & if\ \sigma(\tilde{f}(p)) = 0 \end{cases} \tag{1}$$

$$Z = \begin{cases} \frac{\mu(\tilde{f}(p)) - Perf(p^*)}{\sigma(\tilde{f}(p))} & if\ \sigma(\tilde{f}(p)) > 0 \\ 0 & if\ \sigma(\tilde{f}(p)) = 0 \end{cases} \tag{2}$$

where Φ and ϕ are the CDF and PDF of the standard normal distribution. EI is possibly large for configurations with high predicted performance and for those with high predicted uncertainty.

4.2.2 Adaptive control of exploration and exploitation via contextual simulated annealing

Based on the analysis in Section 3, we propose a modification to the ϵ-greedy sampling in the simulated annealing based optimizer. Instead of using a fixed value for ϵ, we replace ϵ with a contextual factor ϵ_t, which is implicitly tied to the task and the underlying model and changes dynamically as the optimization proceeds. In particular, we define:

$$\epsilon_t = \frac{\bar{\sigma}}{Perf(p^*)} \tag{3}$$

where $Perf(p^*)$ is the best seen plan, and $\bar{\sigma}$ is the mean of the standard deviations from a set of yet unsampled plans from posterior distribution (rather than the prior). Note that it should be distinguished from σ, which is the individual standard deviation of a prediction from $\tilde{f}(\cdot)$ for a particular plan in the posterior. This allows AdaTune to dynamically adjust the exploration–exploitation trade-off based on the surrogate model's state at any time point. We call the resulting optimizer contextual simulated annealing.

This is intuitive, as exploration is, on average, preferred when the model has high uncertainty, and exploitation is preferred when the predicted uncertainty is low. Furthermore, if the optimization is being overly greedy (i.e., getting stuck at a local optimum), $\tilde{f}$ will produce a highly unbalanced standard deviation distribution with small standard deviation close to the local optimum already being sampled, and larger standard deviations elsewhere in the (unsampled) transformation space. This results in a larger value for the standard deviation for the posterior, which is equivalent to an increase of ϵ_t in Eqn. 3 and it increases the ratio of randomly sampled points in the next batch of hardware measurements, presumably helping the search escape from local optimum.

4.3 AdaTune: Putting It Together

In previous sections, we describe how we make the optimization process more adaptive and reduce the hardware measurement cost at each tuning step. In this part, we put everything together and call the resulting target-dependent optimization pass AdaTune (Algorithm 1).

5 Evaluation

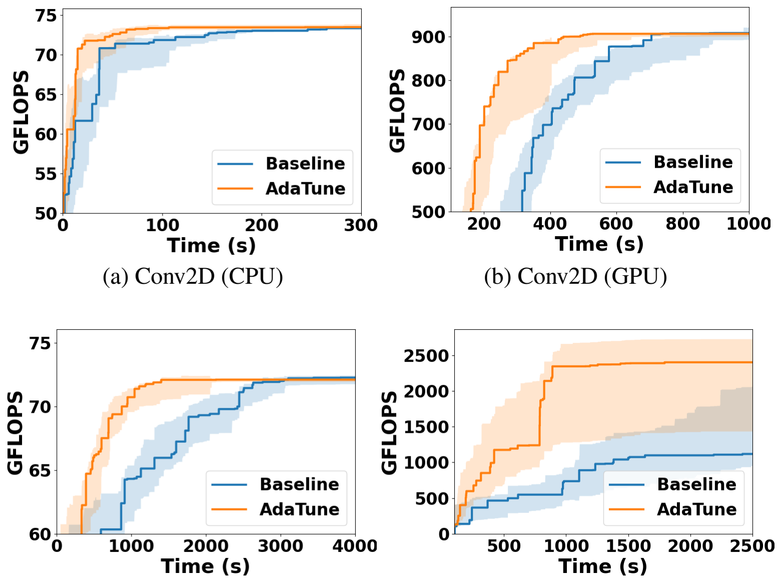
In this section, we evaluate AdaTune experimentally, seeking answers to how AdaTune helps accelerate the optimization process. We integrate AdaTune with TVM [13] and use AutoTVM [14] as our baseline for comparison. We implement AdaTune in Python, and we leverage scikit-learn [33] and forestci [1] to implement the surrogate model and optimizer.

5.1 Comparison of AutoTVM and AdaTune for Searching Transformation Space

We compare the performance of AutoTVM and AdaTune on how much optimization speedup we obtain as a function of the wall-clock time. Due to space limitations, we include four tasks: one convolutional layer sampled from ResNet-18 [20] and one batched GEMM operator from Transformer [41] on both CPU (Intel Xeon CPU E5-2690 v3 @ 2.60GHz 2600 MHz) and GPUs (Nvidia Tesla P100). We use n=500 for all experiments and set micro-batch size $B = 50$ in AdaTune. We use the default settings for other hyperparameters provided by AutoTVM. The detailed parameter settings are included in Appendix A. We perform 15 independent runs of each configuration with different random seeds and report the median together with a 95% confidence interval. Also note that the predicted performance is only used in the transformation space searching process, and we report *real measured latency* in the end-to-end evaluation results.

Algorithm 1 **AdaTune**

```
1: **Input:** Transformation space $S_e$
2: **Output:** Selected transformation plan $p^*$
3: $D \leftarrow \{\}$
4: **while** $n\_iterations < max\_n\_iterations$ **do**
5:     Q $\leftarrow$ run contextual simulated annealing to collect candidates in $S_e$ using the surrogate model
       $\tilde{f}$ and EI in Section 4.2.1                    ▷ Finding the next promising batch
6:     Random sample $K$ plans $p_1, p_2, ..., p_K$ from $S_e$
7:     $\epsilon_t \leftarrow \frac{\frac{1}{K}\sum_{k=1}^{K}(standard\_deviation(\tilde{f}(p_k))}{Perf(p^*)}$
8:     S $\leftarrow$ pick (1 - $\epsilon_t$)b subset from Q
9:     S $\leftarrow$ S $\cup$ {Randomly sample $\epsilon_t$b candidates}
10:    **for** p in S **do** do
11:        **for** i in (1,..,B) **do**                    ▷ Measure the hardware cost with AE
12:            $cv \leftarrow \frac{std(\{Perf(p)_1, Perf(p)_2, ..., Perf(p)_i\})}{avg(\{Perf(p)_1, Perf(p)_2, ..., Perf(p)_i\})}$
13:            **if** $cv < threshold$ **then**
14:                break
15:        $D \leftarrow D \cup (p, Perf(p))$
16:    update $\tilde{f}$ using D                    ▷ Update the model given new measurements
17:    $n\_iterations \leftarrow n\_iterations + b$
18: $p^* \leftarrow$ best found transformation plan
```



(a) Conv2D (CPU)
(b) Conv2D (GPU)
(c) Batched GEMM (CPU)
(d) Batched GEMM (GPU)

Figure 6: Comparison of optimization levels under the same time budget on both CPU and GPUs.

Fig 6 visualizes the results. The x-axis denotes the wall-clock time of auto-tuning. The y-axis denotes the GFLOPS of the best transformation plan found so far. We make the following observations. The first observation is that the best plan AdaTune finds is similar to, and sometimes much better than the baseline. The speedup is especially prominent when the transformation space is extremely large (e.g., Fig. 6d on GPUs), where AdaTune achieves up to 115% higher GFLOPS than the baseline under the same time budget. This indicates that AdaTune is able to explore the transformation space in a more efficient way. The second observation is that AdaTune is 1.3–3.9× faster than AutoTVM to find the best plan (Fig. 6a–6c). This improved speed to find the best plan is important for achieving better anytime performance in optimizing new models. These results confirm that AdaTune is capable of finding good code transformation at a much faster speed than the baseline.

Fig. 6d shows a much larger variance than the other figures because, for that workload, there are lots of zero points (invalid knobs) in the search space; thus the modeling is highly dependant on the nonzero points found at the beginning. If there is not enough exploration, the search tends to be trapped in local optima.

5.2 Comparison on Optimization Time and Model Performance Improvements.

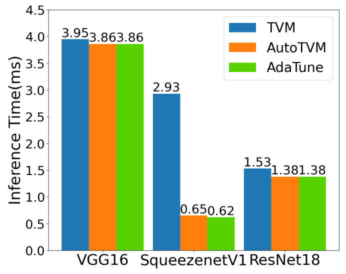


Figure 7: Inference time comparison.

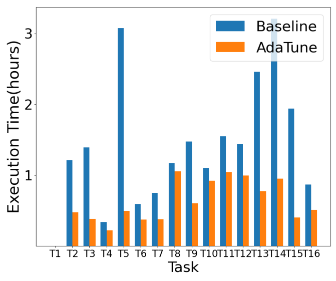


Figure 8: Optimization wall-clock time comparison.

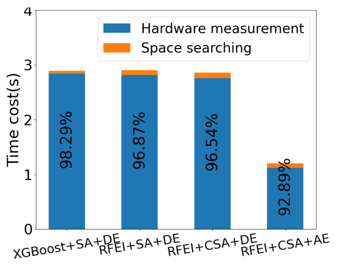


Figure 9: Breakdown comparison of tuning cost.

We compare the end-to-end optimization results on ResNet-18 [20], VGG16 [39], and Squeezenet-V1 [24]. Fig. 7 compares the final inference time from ResNet-18 optimized by TVM, AutoTVM, and AdaTune respectively. Overall, AdaTune achieves up to 4.6% faster inference speed over AutoTVM, and up to 78.8% faster speed over TVM, respectively. AutoTVM and AdaTune achieve much higher speedup on SqueezeNet, presumably because the heuristic-based optimizations in TVM are sub-optimal. In contrast, the Learning to Compile approach is able to quickly identify code transformation leads to a significantly faster speed. While achieving comparable optimization quality as AutoTVM, AdaTune significantly reduces the lengthy optimization time. Due to space limitations, Fig. 8 shows the optimization time on ResNet-18 on GPU. AutoTVM takes 22.6 hours in total for the optimization, whereas AdaTune takes only 9.6 hours to finish the optimization, which is a 2.35× speedup.

5.3 Ablation Analysis

In this section, we compare the effectiveness of design elements in AdaTune by comparing the following schemes:

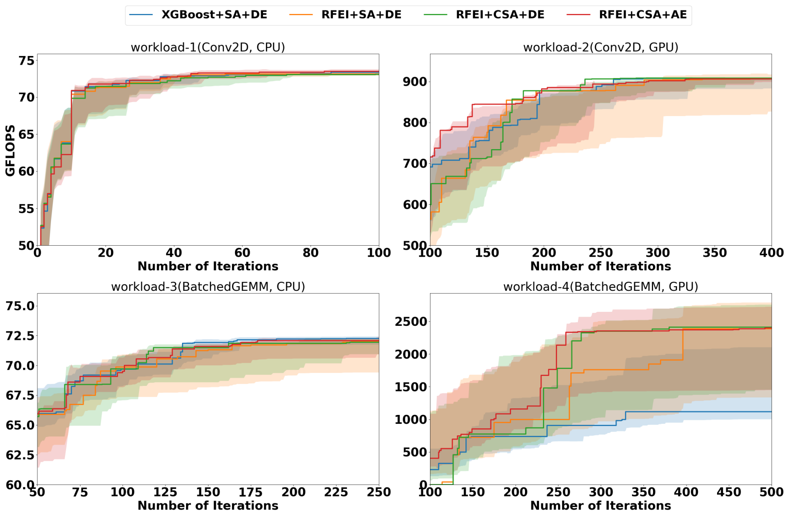


Figure 10: Impact to search effectiveness in iterations.

- XGBoost + SA + DE: This is our baseline, which uses XGBoost as the performance model, simulated annealing (SA) as the optimizer, and deterministic evaluator (DE).

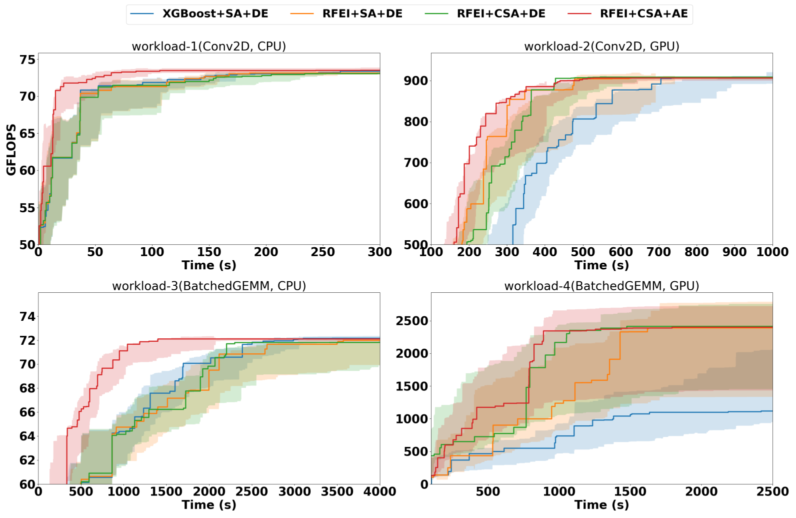


Figure 11: Impact to search effectiveness in wall-clock time.

- RFEI + SA + DE: Like the baseline, but XGBoost is replaced with RFEI. RFEI stands for random forest plus positive expected improvement.
- RFEI + CSA + DE: Like our approach but uses a deterministic evaluator instead of the adaptive evaluator. CSA stands for the contextual simulated annealing.
- RFEI + CSA + AE: Our main algorithm as described in Section 4.3. AE stands for the adaptive evaluator.

Searching effectiveness. The impact on the search effectiveness with respect to iterations is presented in Fig 10. When equipped with the uncertainty-quantifying surrogate model, the search takes a relatively smaller number of iterations to find a better plan (as shown in workload-4 from Fig. 10). In other cases, it performs similarly to the XGBoost model. Contextual simulated annealing further improves the searching effectiveness in some cases (workload 2 and 4 in Fig. 10), presumably because of its effect of regularizing the search to escape from local optima. Finally, with the adaptive evaluator, there is a significant improvement in wall-clock time on all the tasks, as shown in Fig 11.

Cost breakdown. Fig 9 further shows the breakdown in the average time required for transformation space searching (ResNet-18 on CPU) and hardware measurement in one iteration. The other components, such as program code generation, incur only a negligible amount of overhead. Overall, our surrogate model and contextual optimizer add very minimal overhead over the baseline. AdaTune effectively reduces the hardware measurement time by 2.5×, which contributes to the speedup of the end-to-end optimization time.

6 Conclusion

Although highly optimized code can be achieved through existing DL compilers, an obvious drawback is their long code optimization time, required to generate many versions of a tensor program and to profile these versions on hardware. In this paper we have introduced a method, called AdaTune, to make the code optimization process in DL compilers more adaptive to different hardware and models. The adaptive evaluator allows cut hardware measurement cost significantly without losing much accuracy. The uncertainty-aware surrogate model and the contextual optimizer allow us to more efficiently explore the transformation space. As a result, AdaTune achieves higher speedups in terms of finding a good transformation plan on different types of hardware and models, outperforming AutoTVM, a state-of-the-art approach.

Broader Impact

Machine learning and deep learning applications are becoming ubiquitous in large scale production systems. With that growth and the scaling in model size and complexity, the focus on efficiently executing DNN models has become even greater. The push for increased energy efficiency has led to the emergence of diverse heterogeneous systems and hardware architectures. While it is possible to hire deployment engineers to produce highly optimized code for diverse architectures, such an approach is time-consuming. It requires significant manual effort, which is difficult to scale, as new DNN models and operators are coming out on a regular basis. Compilers have historically been the bridge between programming efficiency and high-performance code, which allows fast innovation while producing high-performance code for diverse architectures. Auto-tuning techniques such as AutoTVM modernize a compiler by automatically learning the compiler's optimization decisions as opposed to using heuristic rules. However, the actual cost of running such a tuning process is very expensive. Our techniques speed up the auto-tuning process significantly. It improves the agility of deploying DNN models, fostering fast innovations. It also reduces the amount of hardware resources needed for optimizing DNN models, reducing the corresponding energy consumption and carbon footprint produced.

Acknowledgments and Disclosure of Funding

The authors are grateful for the discussion with Silu Huang, Eric Zhu, and Jon Soifer. The authors appreciate the anonymous NeurIPS reviewers for providing constructive feedback for improving the quality of this paper. All authors are not funded by any other agency.

References

[1] Forest Confidence Interval. http://contrib.scikit-learn.org/forest-confidence-interval/reference/forestci.html. Accessed: 31-May-2020.

[2] Intel(R) Math Kernel Library for Deep Neural Networks. https://github.com/01org/mkl-dnn.

[3] Nvidia A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf. Accessed: 31-May-2020.

[4] The Accelerated Linear Algebra Compiler Framework. https://www.tensorflow.org/performance/xla/.

[5] Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/. Accessed: 19-May-2020.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 265–283, 2016.

[7] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, 2019.

[8] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[9] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. Opentuner: an extensible framework for program autotuning. In José Nelson Amaral and Josep Torrellas, editors, *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 303–316. ACM, 2014.

[10] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5):96:1–96:42, 2019.

[11] 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. *CoRR*, abs/1012.2599, 2010.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv preprint arXiv:1512.01274, 2015.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594, 2018.

[14] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 3393–3404, 2018.

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. arXiv preprint arXiv:1410.0759, 2014.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019)*, pages 4171–4186, 2019.

[17] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 1436–1445, 2018.

[18] Shanghua Gao, Ming-Ming Cheng, Kai Zhao, Xinyu Zhang, Ming-Hsuan Yang, and Philip H. S. Torr. Res2net: A new multi-scale backbone architecture. *CoRR*, abs/1904.01169, 2019.

[19] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[21] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. GRNN: low-latency and scalable RNN inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 41:1–41:16, 2019.

[22] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[23] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, pages 507–523, 2011.

[24] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <1MB Model Size. arXiv preprint arXiv:1602.07360, 2016.

[25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, 2017.

[26] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[27] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[28] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: binarized neural network on FPGA. *Neurocomputing*, 275:1072–1086, 2018.

[29] Ji Lin, Chuang Gan, and Song Han. Training kinetics in 15 minutes: Large-scale distributed training on videos. *arXiv preprint arXiv:1910.00932*, 2019.

[30] Changxi Liu, Hailong Yang, Rujun Sun, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. Swtvm: exploring the automated compilation for deep learning on sunway architecture. *arXiv preprint arXiv:1904.07404*, 2019.

[31] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. VTA: an open hardware-software stack for deep learning. *CoRR*, abs/1807.04188, 2018.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[33] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[34] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[36] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, 2013.

[37] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[39] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[40] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pages 5998–6008, 2017.

[42] Zheng Wang and Michael F. P. O'Boyle. Machine learning in compiler optimization. *Proc. IEEE*, 106(11):1879–1901, 2018.

[43] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[44] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[45] Zifeng Wu, Chunhua Shen, and Anton van den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognit.*, 90:119–133, 2019.

[46] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet another accelerated SGD: resnet-50 training on imagenet in 74.7 seconds. *CoRR*, abs/1903.12650, 2019.

[47] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[48] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. Reducing BERT pre-training time from 3 days to 76 minutes. *CoRR*, abs/1904.00962, 2019.

[49] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, Elton Zheng, Olatunji Ruwase, Jeff Rasley, Jason Li, Junhua Wang, and Yuxiong He. Accelerating large scale deep learning inference through deepcpu at microsoft. In *2019 USENIX Conference on Operational Machine Learning, OpML 2019, Santa Clara, CA, USA, May 20, 2019*, pages 5–7, 2019.

A Hyperparameter Settings

We treat all feature inputs as numeric inputs to the Random Forest model. we set the batch size to 32 (i.e., updating the cost model once with 32 new hardware measured points). For the Random Forest Regressor model, We set the number of trees to 10 to keep the computational overhead small. We set $max\_features$ to 10 to avoid over-fitting and use the default values for other settings in scikit-learn. When calculating the contextual ϵ value, we randomly sample 20 plans from the transformation space to obtain the prediction mean and variance.

B Additional Results

B.1 Comparison with Gaussian Process

We compare two uncertainty estimators: Gaussian Process Regressor and Random Forest Regressor. Results show that Random Forest Regressor performs much better than Gaussian Process Regressor. Therefore, we choose the Random Forest Regressor as our cost model.

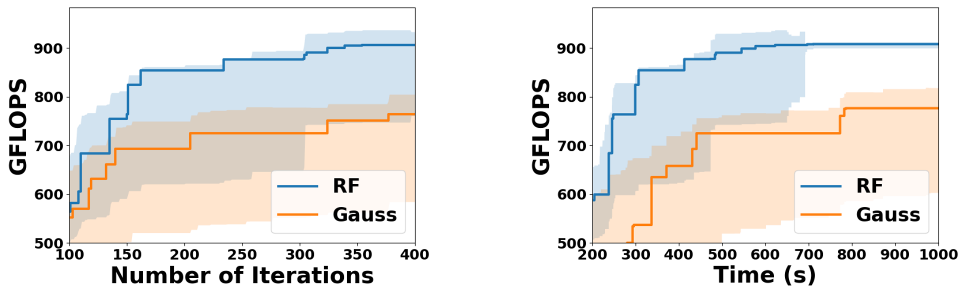


Figure 12: Random Forest vs. Gaussian Process performance.

B.2 Additional Optimization Time Comparison Results

We include more results (Figure 2 and Figure 3) on the optimization time comparisons between AutoTVM and AdaTune. Overall, AdaTune achieves 1.3–2.9X speedup in end-to-end optimization time. The speedup on some GPU tasks is relatively lower because the hardware measurement on GPUs is faster than that of the same task on CPU.

	AutoTVM	AdaTune	Speedup
Resnet-18	22.6h	9.6h	2.4X
Resnet-50	20.0h	14.1h	1.4X
VGG-16	21.9h	16.7h	1.3X
SqueezenetV1	7.6h	5.8h	1.3X
Transformer (Enc.)	3.8h	2.8h	1.4X

Table 2: Optimization time on GPU.

	AutoTVM	AdaTune	Speedup
Resnet-18	2.0h	1.0h	2.0X
Resnet-50	3.6h	1.7h	2.1X
VGG-16	18.9h	6.5h	2.9X
SqueezenetV1	1.2h	0.7h	1.7X
Transformer (Enc.)	8.4h	3.8h	2.2X

Table 3: Optimization time on CPU.

B.3 Additional Inference Time Comparison Results

We include more results (Figure 4 and Figure 5) on the inference time comparisons between AutoTVM and AdaTune. Although being faster to optimize, AdaTune achieves comparable optimization quality and sometimes outperforms AutoTVM in inference time.

B.4 Additional Optimization Cost Results

We include detailed optimization time breakdown results for all tasks in ResNet-18, ResNet-50, VGG-16, Squeezenet-V1.1, and Encoder on both CPU and GPU (Figure 14 Figure 15, Figure 16, and Figure 17). Overall, AdaTune improves the optimization time for individual tasks on both CPU

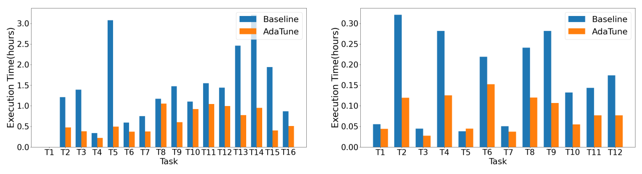
	TVM	AutoTVM	AdaTune
Resnet-18	1.53ms	1.38ms	1.38ms
Resnet-50	4.82ms	4.37ms	4.37ms
VGG-16	3.95ms	3.86ms	3.86ms
SqueezenetV1	2.93ms	0.65ms	0.63ms
Transformer (Enc.)	78.15ms	52.25ms	47.46ms

Table 4: Inference time comparison on GPU.

	TVM	AutoTVM	AdaTune
Resnet-18	79.24ms	52.64ms	52.64ms
Resnet-50	217.12ms	115.76ms	115.68ms
VGG-16	884.94ms	442.01ms	438.68ms
SqueezenetV1	14.41ms	11.36ms	11.25ms
Transformer (Enc.)	2897.27ms	1620.88ms	1607.67ms

Table 5: Inference time comparison on CPU.

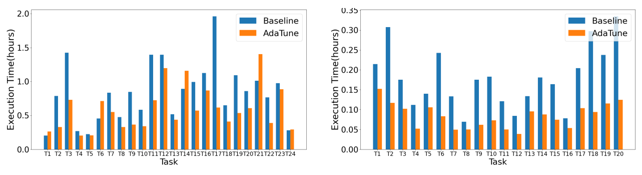
and GPU for the models being tested. On GPU, sometimes AdaTune takes a slightly longer time in certain tasks (e.g., T18 in VGG-16 and T1 in SqueezeNet-V1.1). That is because the auto-tuning process stops when it can no longer find a better solution. We find that AutoTVM sometimes stops earlier because it quickly gets stuck at a local optimum. In contrast, the contextual optimizer in AdaTune constantly pushes AdaTune out of local optimum, which yields a longer time for AdaTune to trigger the stop condition.



(a) ResNet-18, GPU

(b) ResNet-18, CPU

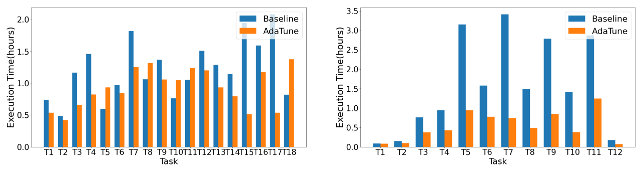
Figure 13: Optimization wall-clock time comparison for ResNet-18.



(a) ResNet-50, GPU

(b) ResNet-50, CPU

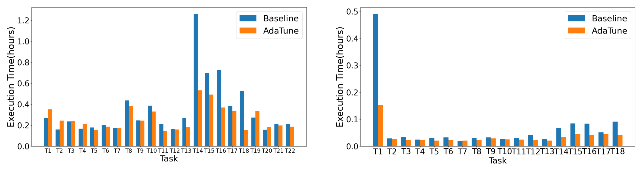
Figure 14: Optimization wall-clock time comparison for ResNet-50.



(a) VGG-16, GPU

(b) VGG-16, CPU

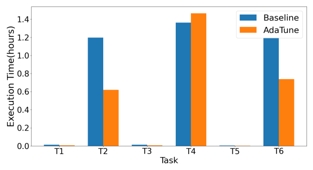
Figure 15: Optimization wall-clock time comparison for VGG-16.



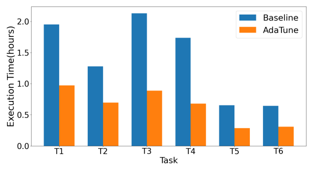
(a) SqueezenetV1.1, GPU

(b) SqueezenetV1.1, CPU

Figure 16: Optimization wall-clock time comparison for SqueezenetV1.1.



(a) Transformer (Enc.), GPU



(b) Transformer (Enc.), CPU

Figure 17: Optimization wall-clock time comparison for Encoder.