Tensor Program Optimization with Probabilistic Programs

Junru Shao
OctoML
jshao@octoml.ai

Xiyou Zhou
OctoML
xiyou@octoml.ai

Siyuan Feng
Shanghai Jiao Tong University
hzfengsy@sjtu.edu.cn

Bohan Hou
Carnegie Mellon University
bohanhou@cs.cmu.edu

Ruihang Lai
Carnegie Mellon University
ruihangl@cs.cmu.edu

Hongyi Jin
Carnegie Mellon University
hongyij@cs.cmu.edu

Wuwei Lin
OctoML
wlin@octoml.ai

Masahiro Masuda
OctoML
mmasuda@octoml.ai

Cody Hao Yu
Amazon Web Services
hyuz@amazon.com

Tianqi Chen
Carnegie Mellon University, OctoML
tqchen@cmu.edu
tqchen@octoml.ai

Abstract

Automatic optimization for tensor programs becomes increasingly important as we deploy deep learning in various environments, and efficient optimization relies on a rich search space and effective search. Most existing efforts adopt a search space which lacks the ability to efficiently enable domain experts to grow the search space. This paper introduces MetaSchedule, a domain-specific probabilistic programming language abstraction to construct a rich search space of tensor programs. Our abstraction allows domain experts to analyze the program, and easily propose stochastic choices in a modular way to compose program transformation accordingly. We also build an end-to-end learning-driven framework to find an optimized program for a given search space. Experimental results show that MetaSchedule can cover the search space used in the state-of-the-art tensor program optimization frameworks in a modular way. Additionally, it empowers domain experts to conveniently grow the search space and modularly enhance the system, which brings 48% speedup on end-to-end deep learning workloads.

1 Introduction

Deep learning has become pervasive in daily life. From video understanding [28], natural language understanding [15], and recommendation system [29] to autonomous driving [21], different deep learning models are deployed on different hardware platforms and devices. Deep learning frameworks usually rely on manually optimized libraries [13, 22] to accelerate deployment. Engineers need to choose from many tensor programs that are logically equivalent but differ significantly in performance due to memory access, threading, and the use of specialized hardware primitives. The engineering effort required for tensor program optimization has become a significant bottleneck for machine learning deployment with the growing number of models and hardware backends.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

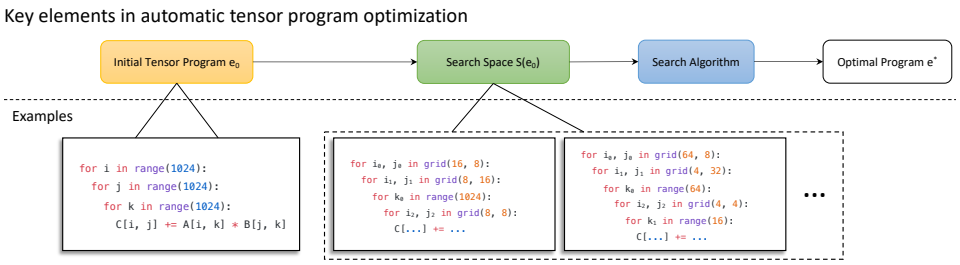


Figure 1: Automatic tensor program optimization contains two key elements: the search space $S(e_0)$ and the search algorithm that finds the optimal tensor program $e^\star$. The search space usually incorporates choices over loop transformation, vectorization, threading patterns, and hardware acceleration.

Automatic program optimization [10, 43, 1] is a recent sequence of efforts that aims to use machine learning to solve this problem. There are two vital elements of automatic tensor program optimizations. First, a search space is defined to provide a possible set of equivalent tensor programs. Then existing systems use learning-based search algorithms to find an optimized tensor program in the search space with feedback from the deployment environment. Most of the current approaches [10, 43, 1, 25, 4] use pre-defined search spaces that effectively encode the domain knowledge of the authors *once* and focus on developing efficient search algorithms.

While efficient search is essential, the search space itself fundamentally limits the best possible performance search algorithms can get. To construct a good search space, domain experts have to make numerous choices over loop transformation, vectorization, threading patterns, and hardware acceleration. Additionally, the best search space itself evolves as new tensor program optimization techniques [24] and hardware primitives [30] grow. As a result, there is a strong need to enable easy customization and construction of the search space at scale by taking inputs from system engineers and domain experts. Unfortunately, any change to search space construction currently requires surgical modifications to the automatic program optimization frameworks.

This research asks the following question: *can we decouple the search space construction from search and provide an adequate abstraction for domain experts and the learning system to collaborate on search space construction?* We give an affirmative answer to the question with two key observations. First, we can parameterize an optimization search space by the initial program followed by a sequence of transformations on the program. Next, using this parameterization, domain experts can then provide probabilistic choices that represent possible transformations after examining the program state. These two observations lead to a simple yet powerful abstraction for search space construction through a domain-specific probabilistic language. Finally, our framework composes multiple possible probabilistic transformations to form a rich search space. We make the following contributions:

- We introduce a simple yet powerful probabilistic language abstraction for tensor program search space construction.
- We build a learning-driven framework to find optimized tensor programs specified by the search space constructed using our abstraction.
- We build an end-to-end system that can take prior knowledge from domain experts to construct optimization search space to optimize deep learning deployment on multiple platforms.

Our end-to-end system can easily expand search space that matches previous approaches without surgical changes and achieve comparable performance on multiple hardware backends. Experimental results show that our abstraction is expressive enough to cover the optimization space of a diverse set of tensor programs, delivering a competitive performance of popular deep learning models, and convenient to incorporate hardware-specific knowledge into the search space to outperform the state-of-the-art frameworks.

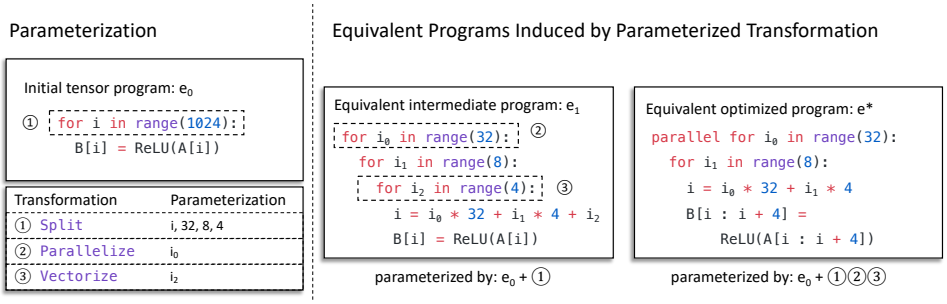


Figure 2: Parameterizing programs with the initial program and sequence of transformations. Tensor program e_1 is parameterized by the initial program e_0, plus Step ①, which is further parameterized by loop i and resulting loop extents 32, 8, 4, respectively.

2 Background and Problem Overview

Figure 1 shows a typical workflow for tensor program optimization. For a given program e_0, a typical tensor program optimization framework will generate candidates from a pre-defined search space $S(e_0)$ containing semantically-equivalent programs. Then the framework finds optimized tensor program $e^* \in S(e_0)$ with the minimum latency on the target hardware.

A typical search space $S(e_0)$ contains choices over threading, loop ordering, memory access, and hardware primitives. Defining the search space $S(e_0)$ for a wide range of tensor programs brings several challenges. First, $S(e_0)$ is highly dependent on e_0. For example, $S(e_0)$ of a compute-intensive program (e.g., `Dense`) needs to consider many more possible configurations than a communication-intensive program such as `ReLU`. The space also differs significantly in different hardware domains. For example, $S(e_0)$ on CPU involves multi-core parallelism and vectorization, while $S(e_0)$ on GPU involves thread binding and tensorization. Finally, as the hardware and model settings change, we need to bring in fresh domain experts to update the $S(e_0)$ to leverage the latest improvements.

This paper aims to provide a programmable abstraction to construct $S(\cdot)$ in a composable and modular way. Our key goals are listed as follows: **Expressiveness.** We need to be able to build a rich search space that covers the optimization programs that domain experts will write. **Modularity.** Tensor program optimization likely will involve inputs from multiple domain experts over different periods. Therefore, we need to be able to combine prior knowledge in a composable and modular way. **Designed for learning.** We need to build a generic learning-based framework to enable diverse variations of the cost model and search for search space specified in our abstraction. We will address the above goals in the following two sections.

3 Composable Search Space Parameterization

This section presents MetaSchedule, a probabilistic approach to search space parameterization.

3.1 Stochastic Search Space Construction

MetaSchedule constructs a search space $S(\cdot)$ with stochastic program transformations as the primitive building blocks. Traditional program optimization can usually be represented by a sequence of *transformations* τ, where at step i, the program e_{i-1} is transformed into a semantically-equivalent program e_i, which finally leads to the optimized program e_n. MetaSchedule generalizes this idea by allowing further parameterization of each transformation step in τ.

Taking Figure 2 as an example: e_0 is the initial program for the program $B = \texttt{ReLU}(A)$[1]. In MetaSchedule, transformation $t_1 = \texttt{Split}$ is parameterized by a loop i and a sequence of integers indicating the loop extents after splitting; Similarly, transforms $t_2 = \texttt{Parallelize}$ and $t_3 = \texttt{Vectorize}$ are parameterized by loops respectively. As a result, an optimized program e_n is obtained by applying a sequence of parameterized transformations τ to the initial program e_0. Accordingly, the search space $S(e_0)$ is composed of e_0 and all possible sequences of parameterized transformations.

[1] In practice, we ingest models from PyTorch/TensorFlow/JAX. See Appendix A.6 for details.

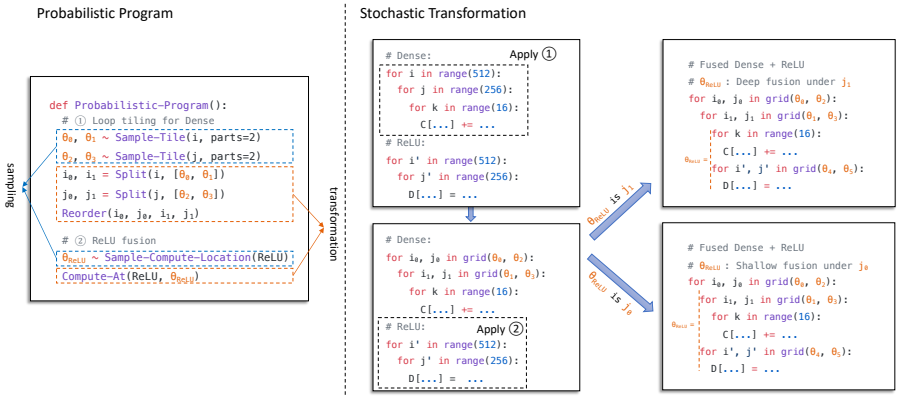


Figure 3: The MetaSchedule probabilistic language. The language contains two key elements: (1) sampling of random variables; (2) program transformation based on random variables. An example execution instance: Step ①: Draw tile sizes of and then organize the loops into a two-level tiling structure. Step ②: Decide where to fuse the `ReLU` operator.

On the other hand, it could be less practical for practitioners to determine the best combination of the parameter values in transformations. For instance, in Figure 2, it is usually efficient to use 512-bit vectorization over the inner loop when AVX-512 [2] vector instructions are available on Intel CPUs, or other vectorization lengths may lead to better performance, otherwise. Therefore, deep knowledge of the target hardware is mandatory to enumerate plausible parameter combinations to control the search space size while covering the optimal program.

To let practitioners efficiently define parameterized transformations without worrying about candidate values, MetaSchedule introduces *random variables* drawn from *analysis, sampling*. Parameterized by random variables, a transformation naturally becomes stochastic, and the underlying probabilistic space reflects the space of possible transformations.

As illustrated in Figure 3, when creating `Split` transformations to tile loop i and j in the `Dense` operator, the tile sizes are drawn by random variables θ_{0-3} defined from `Sample-Tile`. In this way, the `Split` transformation becomes stochastic. Similarly, we use `Sample-Compute-Location` to enumerate valid loops in `Dense` after splitting for `ReLU` to fuse its computation. In summary, 7 lines of MetaSchedule program covers a family of possible optimized tensor programs with stochastic transformations in its search space $S(e_0)$, where e_0 is `Dense-ReLU`.

Notably, unlike orthogonal grid space in hyperparameter tuning, MetaSchedule captures long-term structural and arithmetic dependency between random variables and the tensor program e_i being transformed. As demonstrated on Step ② in Figure 3, sampling distribution from `Sample-Compute-Location` depends on the latest tensor program e_5, whose structure depends on all previous random variables.

3.2 Modular Search Space Composition

Although the search space constructed by stochastic transformations proposed in the previous subsection is efficient and is capable of covering the optimal tensor program, it is hard for other developers to learn how the search space is constructed by reading a long sequence of transformations. It makes transformations designed for a workload hard to be reused by other workloads. Meanwhile, we observe that it is straightforward to group a sub-sequence of transformations for a particular fine-grained optimization. For example, some transformations implement multi-level tiling for better memory locality in compute-intensive operators like `Conv2d` and `Dense`; some other transformations are used to fold/inline elementwise operations such as activation functions into their predecessors or successors for better memory bandwidth efficiency.

To improve the usability and make MetaSchedule more practical, we introduce *transformation module*. Just like the convolutional module with `Conv2D`, `BiasAdd` and `ReLU` in ResNet, a transformation module in MetaSchedule is defined as either atomic stochastic transformation, or composition of

[2] A single X86 instruction that performs the computation to a 512 bits vector in one CPU cycle.

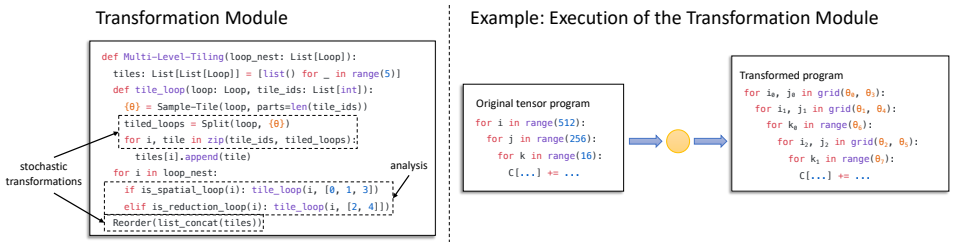


Figure 4: Transformation modules. A transformation module consists of tensor program analysis, sampling, and stochastic transformations. The figure uses `Multi-Level-Tiling` as an example. where analysis is done interactively to identify spatial (data parallel) and reduction loops, and then apply `Split` with the tiling factors drawn from `Sample-Tile`. A final `Reorder` organizes the loops into proper tiles.

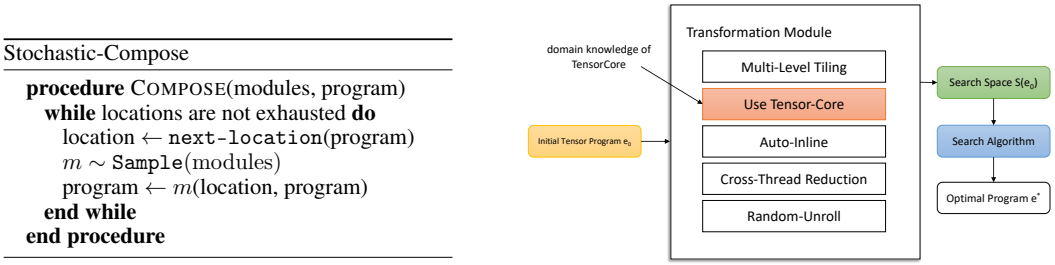


Figure 5: Left: An example algorithm to compose transformation modules. A sequence of transformation modules is composed together into a single transformation module. Right: Hierarchical composition of transformation modules gives generic search space. In this example, a hardware-specific module `Use-Tensor-Core` is composed together with other generic modules into a module that generates search space for any tensor program.

program analysis, sampling as well as smaller transformations. Each transformation module can have a meaningful name so that it can be easily adopted by many workloads to hierarchically construct a search space.

Figure 4 shows hierarchical composition of transformation modules. Specifically, `Multi-Level-Tiling` interleaves program analysis on loops and the stochastic tiling of the loop structure and organizes the original tensor program into a 5-level tiling structure. Notably, the transformation module is generic to tensor programs and thus could be applied to a variety of operators, including `conv1d`, `conv3d`, `matmul`, etc.

Figure 5 depicts an example of composing a search space with transformation modules. In this simple example, we select a set of transformation modules, which are implemented in advance by practitioners with prior domain knowledge, and apply them to every available location in the tensor program to form a search space. Consequently, the formed search space covers common optimizations on diverse hardware.

3.3 Relation to Existing Tensor Program Optimization Methods

In this subsection, we discuss prior approaches for automatic tensor program optimization and illustrate that many of them can be covered by the MetaSchedule framework.

Domain specific languages for program transformations used by prior frameworks [32, 5, 9, 37] allow developers to easily optimize a program manually. When there is no random variable sampling in the program, MetaSchedule reduces to a DSL for deterministic program transformations and achieves the same functionality.

Template-guided auto-tuning [10, 2, 25, 27] fully relies on developers to define a search space. In MetaSchedule, it means all random variables in a search space are defined ahead of the transformations, so there is no interaction between program analysis and follow-up random sampling choices conditioned on the program state.

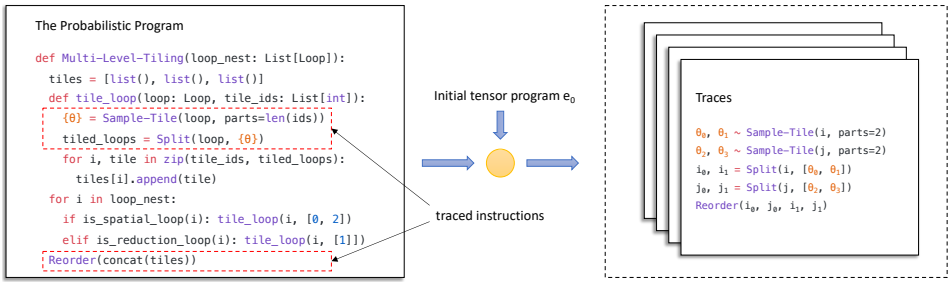


Figure 6: Execution tracing in MetaSchedule. The probabilistic language on the left defines the entire search space $S(e_0)$. Tracing the program execution across different runs leads to a set of linearized probabilistic programs on the right. Only sampling and transformation instructions are traced, while all other constructs and control flow in the host language is ignored.

Auto-scheduling [43, 45, 1, 19] requires developers to implement workload agnostic transformation rules. MetaSchedule achieves the same programmability and functionality through specific probabilistic transformation modules that correspond to the search space generation rules.

Notably, all approaches mentioned above have important use-cases in tensor program optimizations, depending on how much domain knowledge we want to incorporate for a particular scenario. By decoupling the search space construction from the search, we effectively build a single framework for all the use cases and enable further customization without surgical changes to the system.

4 Learning-driven Search

The last section provides a modular abstraction for search space. We still need to do an effective search to find an optimized program within the search space. This section provides a generic learning-driven framework to find an optimized program.

Objective formalization. For a given probabilistic program e_0, let us use τ to denote the transformations performed on e_0. τ can be sampled from a prior distribution specified by the probabilistic program. We define $g(e_0, \tau)$ to be the tensor program after applying transformation τ to e_0. Let $f(e)$ be the latency of the particular program e on the hardware environment. We define a posterior probability of an optimized program as:

$$P(\tau \mid e_0) \propto e^{-f(g(e_0,\tau))} \cdot P(\tau). \tag{1}$$

Intuitively, we want to assign a higher probability to the programs that perform well. Our final goal is to find $\tau^\star = \operatorname{argmax}_\tau P(\tau \mid e_0)$ that maximizes the posterior through maximum a posteriori estimation (MAP) estimation.

Execution tracing. To enable domain experts to express their knowledge via transformations modules productively, we embed MetaSchedule in Python. We introduced execution tracing to reduce the cost of repetitive re-execution of the Python program. Figure 6 demonstrates an example tracing process. During program execution, the system records all samplings and transformations while ignoring control flow and other constructs of the host language. The resulting trace is a sequence of MetaSchedule primitives with only sampling and transformation instructions, which could be re-executed as a normal MetaSchedule program. We can then continue to explore different sampling choices for a given collection of initial traces. Conceptually, this is equivalent to dividing up our support set and then sampling the program condition on the execution sequence of the program.

End-to-end search. Figure 7 shows the overall workflow of our learning-driven framework. The search algorithm first samples the MetaSchedule program to obtain a collection of traces. Then it continues to explore the space condition on the traces. Notably, there is a significantly higher cost measuring $f(e)$ directly on the hardware, so we also incorporated a proxy cost model $\hat{f}(e)$, which is updated throughout the process, similar to previous efforts on tensor program optimization [10, 43]. At each iteration, we adopt an evolutionary search algorithm that proposes a new variant of the trace by mutating the random variables, then accept or reject the proposal based on the cost model. While evolutionary search could be viewed as parallel chain MCMC, we also made our system modular enough to incorporate other ways to select the probabilistic choices, such as those through Bayesian optimization and reinforcement learning.

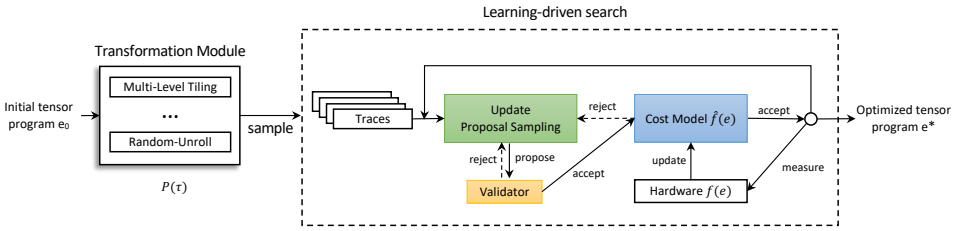


Figure 7: Learning-driven search. Based on the traces of MetaSchedule execution, candidate tensor programs are proposed by mutating sampling decisions in traces, among which the invalid ones are rejected by the validator. In every iteration, proposed candidates are accepted or rejected via annealed Metropolis-Hastings with a prediction from a learned cost model $\hat{f}$, while the cost model $\hat{f}$ is incrementally updated according to f, the measured latency of tensor programs on real hardware.

Cost model. Our approach allows extensive cost models, enabling us to supply those pre-trained from existing datasets [44]. We pick a tree-boosting-based cost model in $\hat{f}(\cdot)$ by default and leverage a common set of features that are used in previous works [43].

Trace validation. Importantly, invalid traces may show up as we propose updates. Such a scenario can happen when some of the random variable choices go beyond the physical hardware limit or a variable that induces changes to the execution sequence. Instead of enforcing a conservative proposal, we introduce a validator that validates the correctness of the trace. The trace validation allows us to move around the space more freely while still ensuring the correctness of the sample outcome to be on the right support set.

5 Related Work

Tensor Program Transformations are proposed by many prior works, such as Halide [32], TVM [9], Tiramisu [5] and TACO [23, 37]. Note that all the previous transformation languages are deterministic and cannot be directly used for search space construction, meaning that they have to introduce a separate programming model to express a search space. This paper makes a simple but powerful generalization to domain-specific probabilistic language. The resulting abstraction enables a unified approach to deterministic transformation and search space construction.

Black-box optimization has been adopted in high-performance computing libraries [16, 3]. Recent advances in automatic tensor program optimization brought a rich set of techniques to accelerate search through better cost modeling [10, 4, 34] and learning-based search [2, 25, 1, 19, 44], which could be incorporated into MetaSchedule search. Different variations of pre-defined search spaces have also been proposed that couple with the automatic tensor program optimization frameworks [10, 43, 1]. Polyhedral model [40, 6, 39] is one useful way to construct a rich pre-defined search space. This paper focuses on modular search space construction and provides orthogonal contributions to these prior works.

Probabilistic programming language is a powerful abstraction for incorporating domain knowledge and probabilistic inference. There are many general-purpose probabilistic languages, such as Church [17], Stan [8], Pyro [7], NumPyro [31], PyMC3 [35] and Edward [38]. This paper proposes a domain-specific probabilistic language for tensor program optimization with specializations such as tensor program analysis that would otherwise be opaque to the previous systems. Our learning-driven search can be viewed as an application of previous works [42, 33, 41, 46] that use tracing to divide programs into subprograms with fixed support. We focus on the MAP inference problem where the posterior depends on an unknown cost function. We solve the problem through a learned cost-model-driven evolutionary search over traces and with validation.

Automatic neural program synthesis [12, 18, 11] has seen large progress recently. Alphacode [26] builds a system that can output creative and sound solutions to problems that require deeper-level human understanding. These approaches generate abstract syntax trees (ASTs) that can be incorrect and use input-output pairs to filter out those erroring programs. Our compiler approach requires us to ensure the correctness of all transformations. However, some ideas like validation after creation might be reusable.

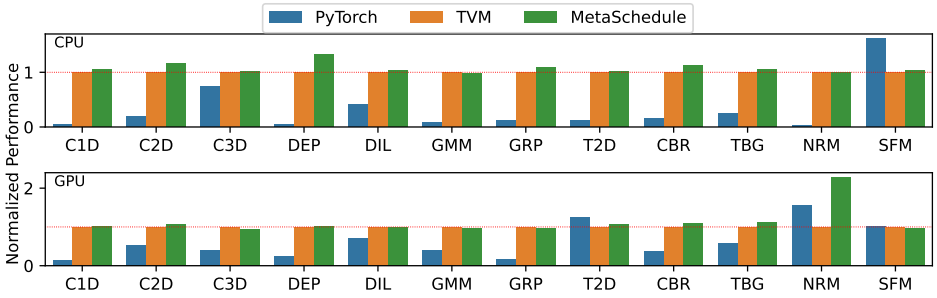


Figure 8: Operator- and subgraph-level performance. MetaSchedule always achieves similar or better performance compared with TVM (Ansor), and exceeds PyTorch significantly in most workloads whose operators are carefully hand-optimized.

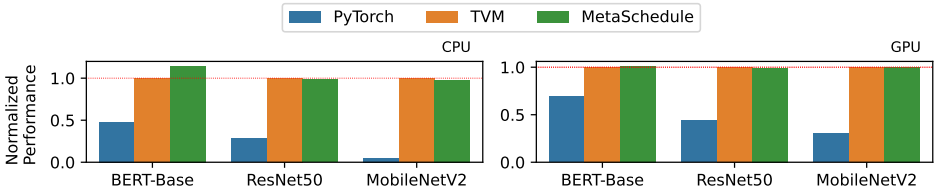


Figure 9: Optimizing end-to-end deep learning models. MetaSchedule reaches close or better performance compared to TVM (Ansor) on all the models.

6 Experiments

6.1 Expressiveness to Cover Common Optimization Techniques

This section aims to answer the following question: *Is MetaSchedule expressive enough to capture the search space of the state-of-the-art optimization techniques?* To answer this question, we evaluate our work on a diverse set of operators and subgraphs extracted from popular deep learning models, including variants of convolution, dense, and normalization.

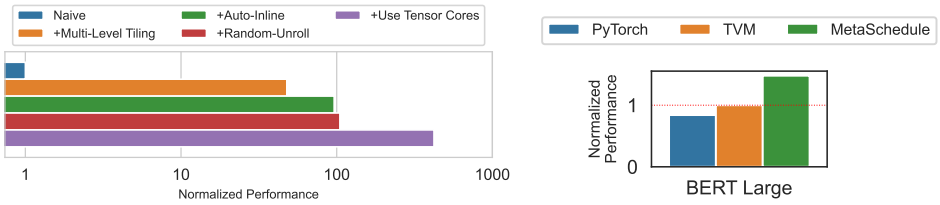
As baselines, PyTorch (v1.11.0) results are provided to compare performance with vendor libraries; TVM (commit: `8d4f4dd73f`), which incorporates AutoTVM [10] and Ansor [43], is used as the state-of-the-art tensor program optimization system, and we pick the best among the two in each respective setups. Full operators and hardware configurations are documented in Appendix A.2.

Figure 8 shows that, in all cases on CPU and GPU, MetaSchedule delivers performance comparable with or even better than TVM, from which we could infer that MetaSchedule could express optimization techniques comparable to TVM on diverse workloads. Additionally, in most of the cases, MetaSchedule outperforms PyTorch by a significant margin except for SFM, which is highly optimized manually in PyTorch.

6.2 Optimizing End-to-End Deep Learning Models

Operator performance does not always translate directly to full model optimization. Therefore, this section is dedicated to answering the following question: *Can MetaSchedule deliver competitive performance with state-of-the-art works for end-to-end models?*

Therefore, a series of experiments are conducted to compare MetaSchedule and TVM, including BERT-Base [14], ResNet-50 [20], and MobileNet-v2 [36] on both CPU and GPU. As shown in Figure 9, MetaSchedule performance is on parity with TVM, while surpassing PyTorch in all cases, which indicates that MetaSchedule framework delivers end-to-end performance. Additionally, tuning time is provided in Appendix A.5.



(a) Performance with different search spaces.

(b) BERT-Large Performance.

Figure 10: Left: Search space composition conducted on a representative subgraph of BERT called `fused-dense`. Gradually composing more transformation modules, the space covers more optimized programs. Right: Introduction of hardware-specific `Use-Tensor-Core` module, composed with existing search space, brought 48% speedup over TVM (AutoTVM).

6.3 Search Space Composition and Hardware-Specific Modules

Besides performance parity with existing work, in this section, we demonstrate the extra value of modular search space composition by answering the following question: *How convenient is it to compose transformation modules, and how does it translate to performance?*

We design an ablation study for transformation modules composition. As indicated in Figure 10a, by progressively enriching the search space, the performance of optimized tensor programs consistently increases. Composed with a hardware-specific module `Use-Tensor-Core`, MetaSchedule delivers significantly better performance compared with generic search space. The performance gain, brought by search space composition with customized rules, does translate to end-to-end model performance, as shown in Figure 10b. Specifically, on BERT-large workloads, MetaSchedule with `Use-Tensor-Core` delivers 48% speedup over TVM.

Notably, it took a graduate student only 2 days to craft the 82-line `Use-Tensor-Core` module in Python (see supplementary materials), which provides strong evidence of the convenience of customization and composition. More details are in Appendix A.4.

7 Conclusion

This paper presents MetaSchedule, a programming model to describe search space construction in tensor program optimization. Our method abstracts search space as a probabilistic language and enables flexible incorporation of domain knowledge by allowing practitioners to implement customized probabilistic programs. A learning-driven search algorithm is developed on top of the probabilistic language abstraction, which delivers competitive performance with state-of-the-art frameworks. In the future, we will explore and modularize declarative API for various hardware environments. Therefore, we will open-source our framework and hope it could enable broader collaboration between the machine learning deployment engineers and intelligent machine learning algorithms for tensor programs.

Acknowledgement

This work is supported in part by a gift from Oppo. We would like to thank Josh Fromm, Denise Kutnick and Sunghyun Park from OctoML for helpful discussion and support of the work.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[2] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743*, 2020.

[3] Collaboration ATLAS, Torsten Åkesson, Paula Eerola, Vincent Hedberg, Göran Jarlskog, Björn Lundberg, Ulf Mjörnmark, Oxana Smirnova, Sverker Almehed, et al. Atlas computing: technical design report. 2005.

[4] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems*, 3:181–193, 2021.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[7] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[8] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[11] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

[12] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34, 2021.

[13] 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.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] 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, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[16] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pages 1381–1384. IEEE, 1998.

[17] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[18] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695, 2020.

[19] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanović, and Ion Stoica. Protuner: tuning programs with monte carlo tree search. *arXiv preprint arXiv:2005.13685*, 2020.

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

[21] Yu Huang and Yue Chen. Autonomous driving with deep learning: A survey of state-of-art technologies. *arXiv preprint arXiv:2006.06091*, 2020.

[22] Intel. Intel® math kernel library for deep learning networks, 2017.

[23] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[24] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 4013–4021, 2016.

[25] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: Adaptive tensor program compilation made efficient. *Advances in Neural Information Processing Systems*, 33:14807–14819, 2020.

[26] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[27] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, 2019.

[28] Ze Liu, Jia Ning, Yue Cao, Yixuan Wei, Zheng Zhang, Stephen Lin, and Han Hu. Video swin transformer. *arXiv preprint arXiv:2106.13230*, 2021.

[29] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[30] Nvidia. Nvidia tensor cores, 2017.

[31] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

[32] 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. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[33] Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. C3: Lightweight incrementalized mcmc for probabilistic programs using continuations and callsite caching. In *Artificial Intelligence and Statistics*, pages 28–37. PMLR, 2016.

[34] Jaehun Ryu and Hyojin Sung. Metatune: Meta-learning based cost model for fast and efficient auto-tuning frameworks. *arXiv preprint arXiv:2102.04199*, 2021.

[35] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

[36] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[37] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[38] Dustin Tran, Matthew D. Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, Alexey Radul, Matthew Johnson, and Rif A. Saurous. Simple, distributed, and accelerated probabilistic programming. In *Neural Information Processing Systems*, 2018.

[39] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*, pages 185–201. Springer, 2006.

[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. *arXiv preprint arXiv:1802.04730*, 2018.

[41] David Wingate, Andreas Stuhlmüller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778. JMLR Workshop and Conference Proceedings, 2011.

[42] Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. Generating efficient mcmc kernels from probabilistic programs. In *Artificial Intelligence and Statistics*, pages 1068–1076. PMLR, 2014.

[43] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

[44] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[45] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[46] Yuan Zhou, Hongseok Yang, Yee Whye Teh, and Tom Rainforth. Divide, conquer, and combine: a new inference strategy for probabilistic programs with stochastic support. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 11534–11545. PMLR, 13–18 Jul 2020.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section gen_inst.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See section 7
   (c) Did you discuss any potential negative societal impacts of your work? [No] To the best of our knowledge, there is no potential negative societal impact of our work, given the only focus is to accelerate existing machine learning models.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A] Our work does not include theoretical results.
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] We will not include the URL for codebase for anonymity, and will release the link after the review process.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Operator configurations and hyperparameters for evolutionary search are shown in the Appendix.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] The variance of running time of tensor programs is negligible across several runs.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We use the codes of TVM and PyTorch which are both cited.
   (b) Did you mention the license of the assets? [No] TVM is under Apache 2.0 license. PyTorch is under Modified BSD license.
   (c) Did you include any new assets either in the supplemental material or as a URL? [No] We did not use any new assets.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] We did not use data obtained from other people.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] We did not use data contains personally identifiable information or offensive content.

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] We did not use any crowdsourcing or conduct any research with human subjects.
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Appendix

A.1 Environment Setup for Experiments

All CPU experiments are conducted on AWS C5.9xlarge instances with Intel Xeon Platinum 8124M CPUs. All GPU experiments are done with NVIDIA GeForce RTX 3070 graphics cards.

A.2 Workload Configurations in the Evaluation

- C1D (1D Convolution): batch=1, length=256, input channel=64, output channel=128, kernel size=3, stride=2, padding=1
- C2D (2D Convolution): batch=1, height=224, width=224 input channel=3, output channel=64, kernel size=7, stride=2, padding=3
- C3D (3D Convolution): batch=1, depth=16, height=224, width=224 input channel=3, output channel=64, kernel size=7, stride=2, padding=3
- DEP (Depthwise Convolution): batch=1, height=112, width=112 channel=32, kernel size=3, stride=1, padding=1
- DIL (Dilated Convolution): batch=1, height=224, width=224 input channel=3, output channel=64, kernel size=7, stride=2, padding=3, dilation=2
- GMM (Matrix Multiply): batch=1, N=M=K=128
- GRP (Group Convolution): batch=1, height=56, width=56 input channel=64, output channel=128, kernel size=3, stride=2, padding=1, groups=4
- T2D (Transposed 2D Convolution): batch=1, height=4, width=4 input channel=512, output channel=256, kernel size=4, stride=2, padding=1
- CBR (2D Convolution + Batch Norm + RuLU): batch=1, height=224, width=224 input channel=3, output channel=64, kernel size=7, stride=2, padding=3
- TBG (Transpose + Matrix Multiply): batch=1, seq=128, head=12, dim=64
- NRM (Norm): batch=1, m=256, n=256
- SFM (Softmax): batch=1, m=256, n=256

A.3 Use-Tensor-Core Search Space Definition Code

```
b0 = sch.get_block(name="T_dense", func_name="main")
b1 = sch.get_block(name="T_add", func_name="main")
b2 = sch.get_block(name="T_multiply", func_name="main")
b3 = sch.get_block(name="T_cast", func_name="main")
b4 = sch.get_block(name="T_erf", func_name="main")
b5 = sch.get_block(name="T_cast_1", func_name="main")
b6 = sch.get_block(name="T_multiply_1", func_name="main")
b7 = sch.get_block(name="T_add_1", func_name="main")
b8 = sch.get_block(name="T_multiply_2", func_name="main")
b9 = sch.get_block(name="root", func_name="main")
sch.annotate(block_or_loop=b0, ann_key="tiling_structure", ann_val="SSSRRSRS")
b10 = sch.reindex(block=b0, buffer_index=0, is_write_index=True)
b11 = sch.reindex(block=b0, buffer_index=0, is_write_index=False)
b12 = sch.reindex(block=b0, buffer_index=1, is_write_index=False)
sch.transform_block_layout(block=b10, index_map=lambda i_l, j_l, k_l: (i_l, j_l, k_l, ))
sch.transform_block_layout(block=b11, index_map=lambda i_l, j_l, k_l: (i_l, j_l, k_l, ))
sch.transform_block_layout(block=b12, index_map=lambda i_l, j_l, k_l: (i_l, j_l, k_l, ))
sch.transform_block_layout(block=b0, index_map=lambda i_l, j_l, k_l: (i_l, j_l, k_l, ))
sch.transform_layout(block=b0, buffer_index=0, buffer_index_type=write, index_map=lambda i_l, j_l: (i_l, j_l, ))
sch.transform_layout(block=b0, buffer_index=1, buffer_index_type=read, index_map=lambda j_l, k_l: (j_l, k_l, ))
sch.transform_layout(block=b0, buffer_index=0, buffer_index_type=read, index_map=lambda i_l, k_l: (i_l, k_l, ))
l13, l14, l15 = sch.get_loops(block=b0)
l16, l17 = sch.split(loop=l15, factors=[64, 16])
l18, l19 = sch.split(loop=l14, factors=[256, 16])
l20, l21 = sch.split(loop=l13, factors=[64, 16])
l22, l23, l24, l25, l26, l27 = sch.get_loops(block=b0)
sch.reorder(l24, l26, l21, l19, l17)
b28 = sch.blockize(loop=l21)
sch.annotate(block_or_loop=b0, ann_key="auto_tensorize", ann_val="wmma_sync")
sch.annotate(block_or_loop=b28, ann_key="auto_tensorize", ann_val="wmma_fill")
b29 = sch.get_block(name="root", func_name="main")
sch.annotate(block_or_loop=b29, ann_key="tensor_core_enabled", ann_val="1")
b30 = sch.get_block(name="root", func_name="main")
sch.annotate(block_or_loop=b30, ann_key="warp_execution", ann_val=1)
l31, l32, l33 = sch.get_loops(block=b28)
v34, v35, v36, v37, v38 = sch.sample_perfect_tile(loop=l31, n=5, max_innermost_factor=4, decision=[8, 1, 2, 2, 2])
l39, l40, l41, l42, l43 = sch.split(loop=l31, factors=[v34, v35, v36, v37, v38])
v44, v45, v46, v47, v48 = sch.sample_perfect_tile(loop=l32, n=5, max_innermost_factor=4, decision=[1, 32, 4, 2, 1])
l49, l50, l51, l52, l53 = sch.split(loop=l32, factors=[v44, v45, v46, v47, v48])
v54, v55, v56 = sch.sample_perfect_tile(loop=l33, n=3, max_innermost_factor=4, decision=[32, 2, 1])
l57, l58, l59 = sch.split(loop=l33, factors=[v54, v55, v56])
sch.reorder(l39, l49, l40, l50, l41, l51, l57, l58, l42, l52, l59, l43, l53)
l60 = sch.fuse(l39, l49)
sch.bind(loop=l60, thread_axis="blockIdx.x")
l61 = sch.fuse(l40, l50)
sch.bind(loop=l61, thread_axis="blockIdx.y")
l62 = sch.fuse(l41, l51)
sch.bind(loop=l62, thread_axis="threadIdx.y")
sch.annotate(block_or_loop=b28, ann_key="thread_extent_low_inclusive", ann_val=32)
sch.annotate(block_or_loop=b28, ann_key="thread_extent_high_inclusive", ann_val=1024)
b63 = sch.write_at(loop=l62, block=b28, write_buffer_index=0, storage_scope="wmma.accumulator")
sch.reverse_compute_inline(block=b10)
v64 = sch.sample_categorical(candidates=[4, 8, 16], probs=[0.0.33, 0.0.33, 0.0.33], decision=0)
sch.annotate(block_or_loop=b63, ann_key="vector_bytes", ann_val=v64)
b65 = sch.read_at(loop=l57, block=b28, read_buffer_index=0, storage_scope="shared.dyn")
v66 = sch.sample_categorical(candidates=[4, 8, 16], probs=[0.0.33, 0.0.33, 0.0.33], decision=2)
sch.annotate(block_or_loop=b65, ann_key="vector_bytes", ann_val=v66)
sch.annotate(block_or_loop=b65, ann_key="local_stage", ann_val=1)
sch.annotate(block_or_loop=b65, ann_key="double_buffer_scope", ann_val=0)
b67 = sch.read_at(loop=l57, block=b28, read_buffer_index=1, storage_scope="shared.dyn")
v68 = sch.sample_categorical(candidates=[4, 8, 16], probs=[0.0.33, 0.0.33, 0.0.33], decision=2)
sch.annotate(block_or_loop=b67, ann_key="vector_bytes", ann_val=v68)
sch.annotate(block_or_loop=b67, ann_key="local_stage", ann_val=1)
sch.annotate(block_or_loop=b67, ann_key="double_buffer_scope", ann_val=0)
b69 = sch.read_at(loop=l58, block=b28, read_buffer_index=0, storage_scope="wmma.matrix_a")
b70 = sch.read_at(loop=l58, block=b28, read_buffer_index=1, storage_scope="wmma.matrix_b")
sch.compute_inline(block=b11)
sch.compute_inline(block=b12)
sch.annotate(block_or_loop=l58, ann_key="software_pipeline_stage", ann_val=[0, 0, 1])
sch.annotate(block_or_loop=l58, ann_key="software_pipeline_order", ann_val=[0, 1, 2])
sch.annotate(block_or_loop=l57, ann_key="software_pipeline_stage", ann_val=[0, 0, 0, 0, 0, 1, 1])
sch.annotate(block_or_loop=l57, ann_key="software_pipeline_order", ann_val=[0, 3, 1, 4, 5, 2, 6])
sch.compute_inline(block=b7)
sch.compute_inline(block=b6)
sch.compute_inline(block=b5)
sch.compute_inline(block=b4)
sch.compute_inline(block=b3)
sch.compute_inline(block=b2)
sch.compute_inline(block=b1)
sch.reverse_compute_inline(block=b8)
v71 = sch.sample_categorical(candidates=[0, 16, 64, 512, 1024], probs=[0.2, 0.2, 0.2, 0.2, 0.2], decision=3)
sch.annotate(block_or_loop=b9, ann_key="unroll_explicit", ann_val=v71)
```

A.4 Search Space Construction and Customization for New Hardware

Our previous experience suggests that adapting to new hardware would require months of effort and deep expertise in the compilation stack, across code generation, transformations, and search across the legacy codebase.

Take TensorCore GPUs as an example. Unfortunately, Ansor in TVM does not support TensorCore, and it is highly non-trivial to extend its support to cover TensorCore without a dramatic revamp to its IR; Similarly, AutoTVM does not deliver comparable TensorCore performance without a dramatic re-design of its schedule tree, even though it supports a limited set of tensor intrinsics (e.g. WMMA). In both cases, with the assumption of upgrading the core design, we anticipate that it will take months of engineering effort, and more to first get familiar with the codebase.

MetaSchedule enables doing that without such coupled complexity and no prerequisite experience on code generation or existing transformations, where each transformation is modeled as an independent primitive in the probabilistic language. Incorporating TensorCore, as an independent program transformation, could be engineered within 2 days by a grad student, and then composed into the existing system without having to re-design or affect any existing functionality.

A.5 Tuning Time

MetaSchedule makes an orthogonal contribution as it is a probabilistic language for composable search space construction rather than speeding up tuning. To provide a fair comparison of tuning speed, we reproduced Ansor's search space in our MetaSchedule probabilistic language in Table 1

	TVM Ansor (min)	MetaSchedule (min)
ResNet-50	287.17	220.66
BERT-base	292.45	288.4644
MobileNet-v2	280.53	251.8831
GPT-2	270.97	214.1358
Inception-v1	295.766	290.3632

Table 1: Tuning time comparison.

A.6 End-to-End Integration Workflow with Deep Learning Framework

From frontend frameworks, for example, TensorFlow, PyTorch, or JAX, the tensor program to be optimized is generated from their computational graph. The generation process is generic to the shapes/ranks of tensors. The MetaSchedule program is generated from the tensor program obtained, based on which the search algorithm is performed.

A.7 Available Transformations Primitives

Transformation	Explanation
split	Split a loop into a sequence of consecutive loops
fuse	Fuse a sequence of consecutive loops into one
reorder	Reorder a sequence of loops
parallel	Parallelize a loop across CPU cores
vectorize	Vectorize a loop with SIMD
unroll	Unroll a loop
bind	Bind a loop to a GPU thread
cache-read	Create a block that reads a buffer region into a read cache
cache-write	Create a block that writes a buffer region into a write cache
compute-at	Move a producer block under the specific loop
compute-inline	Inline a block into its consumer(s)
rfactor	Factorize an associative reduction block by the specified loop
storage-align	Set alignment requirement for specific dimension of a buffer
set-scope	Set the storage scope of a buffer
add-unit-loop	Create a new unit loop on top of the specific block
re-index	Create a block that read/write a buffer region into a read/write cache with reindexing
reverse-compute-at	Move a consumer block under the specific loop
reverse-compute-inline	Inline a block into its only producer
decompose-reduction	Decompose a reduction block into two separate blocks
blockize	Convert the subtree rooted at a specific loop into a block
tensorize	Tensorize the computation enclosed by loop with the tensor intrin
annotate	Annotate a block/loop with a key value pair
unannotate	Unannotate a block/loop's
transform-layout	Apply a transformation to buffer layout, represented by an index map
transform-block-layout	Apply a transformation to block layout, represented by an index map
decompose-padding	Decompose a padding block into a block filling const pad values and a block writing in-bound values
sample-categorical	Sample an integer given the probability distribution
sample-perfect-tile	Sample the factors to perfect tile a specific loop
sample-compute-location	Sample a compute-at location of the given block

Table 2: All available transformation primitives.