Ansor: Generating High-Performance Tensor Programs for Deep Learning

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, Ion Stoica

R244 Presentation

Juntong Deng

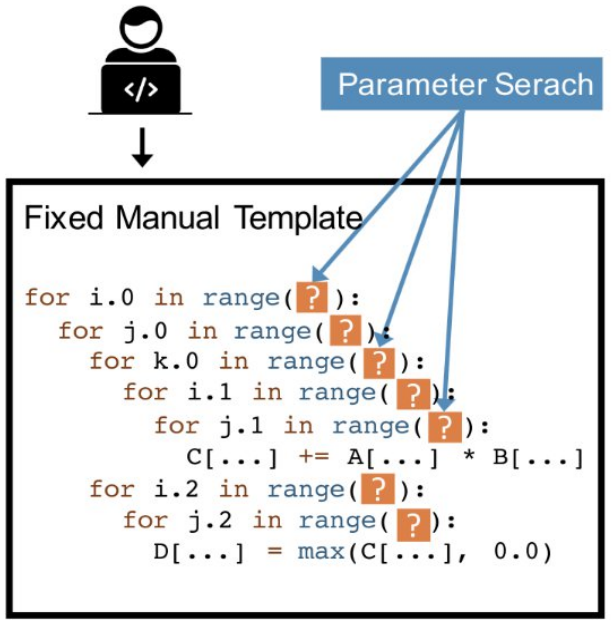
jd2125@cam.ac.uk

Motivation — High-performance Tensor Programs

- High-performance tensor programs are crucial for efficiently executing deep learning models
- Deep learning models are being deployed on a variety of hardware platforms (CPUs, GPUs, TPUs, FPGAs, ASICs, etc.)
- It is difficult to obtain high-performance tensor programs for different operators across various hardware platforms
- It typically requires a significant amount of engineering work to develop hardware-platform-specific optimized code
- We need automated methods to find (generate) high-performance tensor programs

Previous Work — Template-guided search

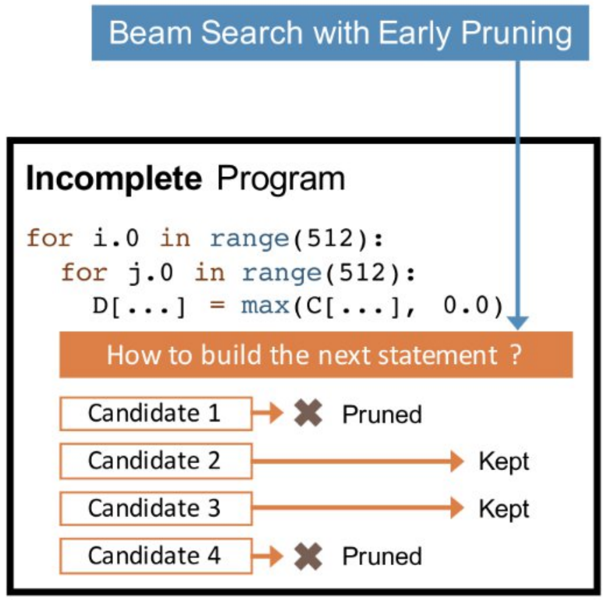
- The search space is defined by templates manually written by users
- Example: TVM [5] (An Automated End-to-End Optimizing Compiler for Deep Learning)
- Templates define tensor program structures with adjustable parameters
- The compiler searches for the best values of these parameters for the specific input shape configuration and specific hardware target
- Developing these templates requires substantial efforts
- TVM repository contains over 15K lines of code for these templates
- Constructing quality templates requires expertise in both tensor operators and hardware
- Only cover limited program structures (manually enumerating all optimization choices is prohibitive)



```
for i.0 in range(?):
  for j.0 in range(?):
    for k.0 in range(?):
      for i.1 in range(?):
        for j.1 in range(?):
          C[...] += A[...] * B[...]
    for i.2 in range(?):
      for j.2 in range(?):
        D[...] = max(C[...], 0.0)
```

Previous Work — Sequential construction based search

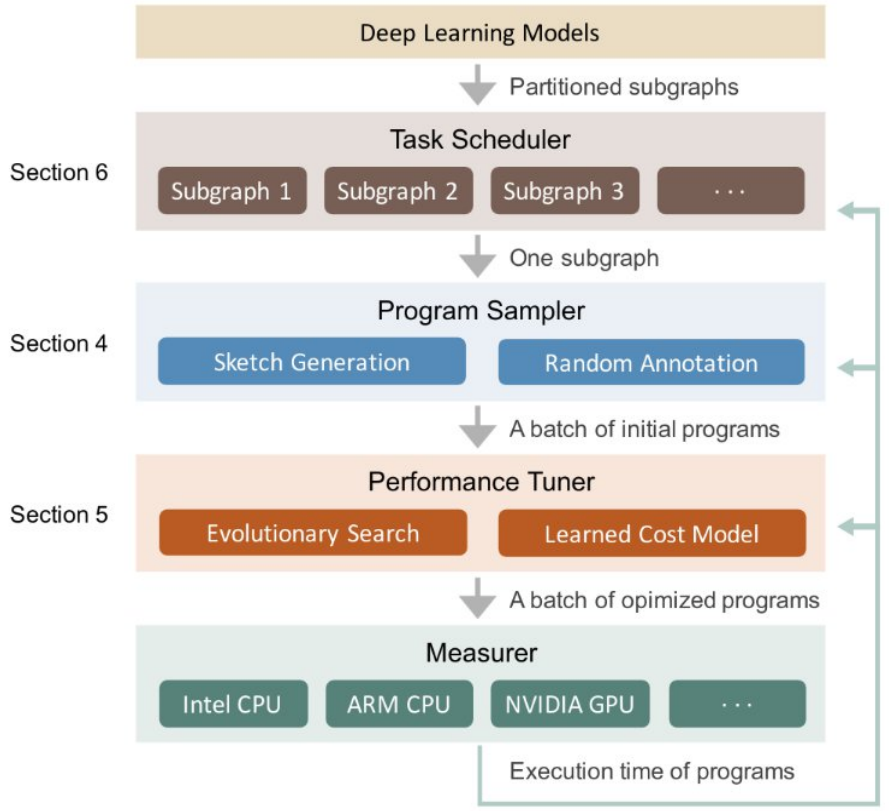
- The search space is defined by decomposing the program into a fixed sequence of decisions
- Example: Halide Auto-Scheduler [1]
- The compiler constructs a tensor program by sequentially unfolding all nodes in the computation graph and making decisions at each node, using algorithms such as beam search to search for good decisions
- When making decisions, the program is incomplete, and cost models trained on complete programs cannot accurately predict performance
- The fixed order of sequential decisions limits the design of the search space
- Sequential construction based search is not scalable



```
for i.0 in range(512):
  for j.0 in range(512):
    D[...] = max(C[...], 0.0)
```

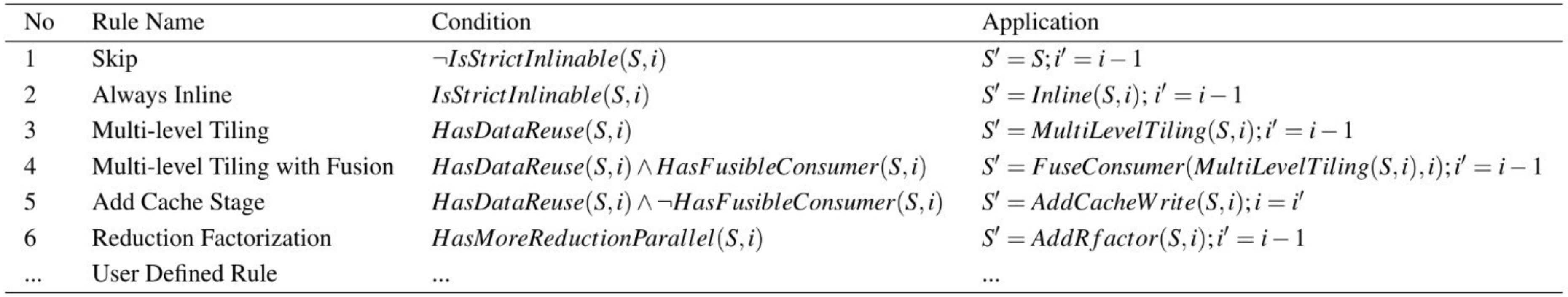
Ansor — Design Overview

- Ansor, a framework for automated tensor program generation
- The input of Ansor is a set of to be optimized DNNs
- Program Sampler
  - Constructs a large search space and samples diverse programs from it
- Performance Tuner
  - Fine-tunes the performance of sampled programs
- Task Scheduler
  - Allocates time resources for optimizing multiple subgraphs in the DNNs



Program Sampler — Sketch Generation

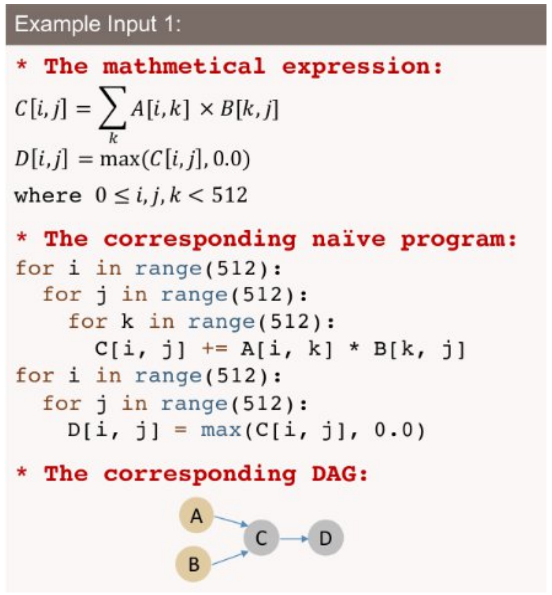
- To sample programs that can cover a large search space, Ansor define a hierarchical search space with two levels: **sketch** and **annotation**
- The high-level structure of a program is defined as sketches
- Ansor generates sketches by recursively applying a few derivation rules
- Ansor allows users to register new derivation rules and integrate them seamlessly with existing rules to adapt to emerging algorithms and hardware



No	Rule Name	Condition	Application
1	Skip	$\neg IsStrictInlinable(S,i)$	$S' = S; i' = i-1$
2	Always Inline	$IsStrictInlinable(S,i)$	$S' = Inline(S,i);\ i' = i-1$
3	Multi-level Tiling	$HasDataReuse(S,i)$	$S' = MultiLevelTiling(S,i); i' = i-1$
4	Multi-level Tiling with Fusion	$HasDataReuse(S,i) \land HasFusibleConsumer(S,i)$	$S' = FuseConsumer(MultiLevelTiling(S,i),i); i' = i-1$
5	Add Cache Stage	$HasDataReuse(S,i) \land \neg HasFusibleConsumer(S,i)$	$S' = AddCacheWrite(S,i); i = i'$
6	Reduction Factorization	$HasMoreReductionParallel(S,i)$	$S' = AddRfactor(S,i); i' = i-1$
...	User Defined Rule	...	...

Program Sampler — Random Annotation

- The sketches generated are incomplete programs because they only have tile structures
- Billions of low-level choices (e.g., tile size, parallel, unroll annotations) as annotations
- Ansor randomly annotates these sketches to get complete programs for fine-tuning and evaluation
- Ansor allows users to give simple hints in the computation definition to adjust the annotation policy


Example Input 1:

* The mathmetical expression:

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$

$$D[i,j] = \max(C[i,j], 0.0)$$

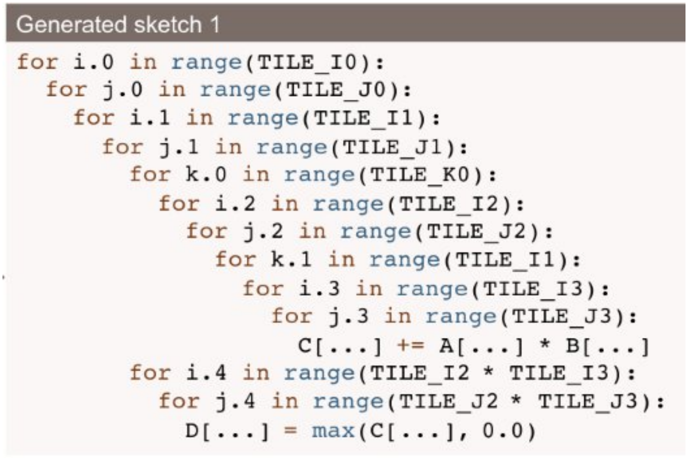
where $0 \le i,j,k < 512$

* The corresponding naïve program:

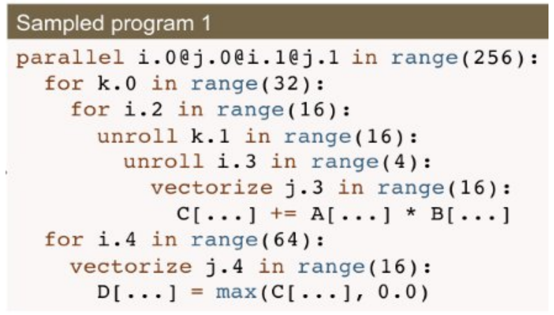
```
for i in range(512):
  for j in range(512):
    for k in range(512):
      C[i, j] += A[i, k] * B[k, j]
for i in range(512):
  for j in range(512):
    D[i, j] = max(C[i, j], 0.0)
```

* The corresponding DAG:

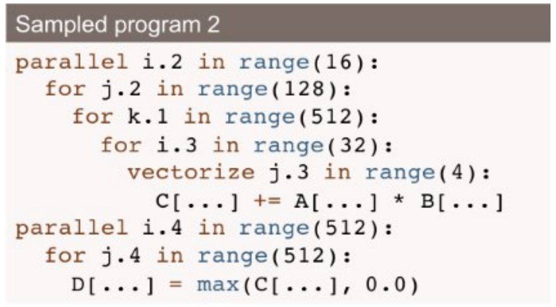
$$\begin{tikzcd} A \arrow[dr] & & \\ & C \arrow[r] & D \\ B \arrow[ur] & & \end{tikzcd}$$


Generated sketch 1

```
for i.0 in range(TILE_I0):
  for j.0 in range(TILE_J0):
    for i.1 in range(TILE_I1):
      for j.1 in range(TILE_J1):
        for k.0 in range(TILE_K0):
          for i.2 in range(TILE_I2):
            for j.2 in range(TILE_J2):
              for k.1 in range(TILE_I1):
                for i.3 in range(TILE_I3):
                  for j.3 in range(TILE_J3):
                    C[...] += A[...] * B[...]
        for i.4 in range(TILE_I2 * TILE_I3):
          for j.4 in range(TILE_J2 * TILE_J3):
            D[...] = max(C[...], 0.0)
```


Sampled program 1

```
parallel i.0@j.0@i.1@j.1 in range(256):
  for k.0 in range(32):
    for i.2 in range(16):
      unroll k.1 in range(16):
        unroll i.3 in range(4):
          vectorize j.3 in range(16):
            C[...] += A[...] * B[...]
  for i.4 in range(64):
    vectorize j.4 in range(16):
      D[...] = max(C[...], 0.0)
```


Sampled program 2

```
parallel i.2 in range(16):
  for j.2 in range(128):
    for k.1 in range(512):
      for i.3 in range(32):
        vectorize j.3 in range(4):
          C[...] += A[...] * B[...]
parallel i.4 in range(512):
  for j.4 in range(512):
    D[...] = max(C[...], 0.0)
```

Performance Tuner — Evolutionary Search

- The quality of the program sampled randomly by the program sampler cannot be guaranteed
- Ansor needs to fine-tune the performance of the sampled program through the performance tuner
- The performance tuner performs fine-tuning via **evolutionary search** and a **learned cost model**
- Initial population used in the evolutionary search:
  - randomly sampled programs
  - high-quality programs from the previous measurement
- Evolution operations in the evolutionary search:
  - Tile size mutation
  - Parallel, vectorization mutation
  - Node-based crossover

Performance Tuner — Learned Cost Model

- The learned cost model is used to predict the fitness of each program
- In Ansor, fitness is the throughput of programs
- The evolutionary search iteratively finds a small batch of promising programs based on the learned cost model, then measures their actual execution time costs on hardware
- The cost model is orders of magnitude faster than the actual measurement, allowing us to compare tens of thousands of programs in the search space in seconds
- The profiling data got from measurement is used to re-train the cost model to make it more accurate
- The evolutionary search gradually generates higher-quality programs for the target hardware

Task Scheduler

- For some subgraphs, spending time in tuning them does not improve the end-to-end DNN performance significantly
  - The subgraph is not a performance bottleneck
  - Tuning brings only minimal improvement in the subgraph’s performance
- Ansor needs to avoid wasting time tuning unimportant (low returns) subgraphs
- Ansor uses a gradient descent-based scheduling algorithm to efficiently optimize the objective function (predefined or user-provided)
- Ansor prioritizes subgraphs with high initial latency, optimistically guessing to quickly reduce its latency. If Ansor spends many iterations on the subgraph without observing a decrease in latency, Ansor leaves the subgraph

Evaluation

The authors evaluated tensor programs generated by Ansor on Intel CPU, ARM CPU, NVIDIA GPU

- Single Operator Benchmark
  - Common deep learning operators: C1D, C2D, C3D, GMM, GRP, etc.
  - Ansor outperforms existing search frameworks by 1.1 −32.7x
- Subgraph Benchmark
  - Two common subgraphs in DNNs: ConvLayer and TBG
  - Ansor outperforms manual libraries and other search frameworks by 1.1 −1.8x
- End-to-End Network Benchmark
  - The end-to-end inference execution time of several DNNs: ResNet-50, MobileNet-V2, etc.
  - Ansor performs the best or equally the best on 24 out of 25 cases

Conclusion & Key Contributions

- Proposed Ansor, a high-performance tensor program generation framework
- Proposed a hierarchical representation mechanism that can cover a large tensor program search space
- Proposed an evolutionary search with a learned cost model to fine-tune the performance of sampled tensor programs
- Proposed a scheduling algorithm based on gradient descent to prioritize important subgraphs
- Evaluated Ansor in common deep learning operators, subgraphs, and end-to-end networks
- Ansor outperforms state-of-the-art systems

Agree & Strength

- Covering large search spaces via hierarchical search space representation without relying on manually defined templates
- Efficiently fine-tune sampled programs via evolutionary search with learned cost model
- Avoid wasting time on low-return fine-tuning via gradient descent-based task scheduler
- Encouraging performance, larger search space, and generation of higher-performance tensor programs in a shorter time
- Ansor enables automatic extensions to new operators
- All Ansor source code is publicly available

Disagree & Weaknesses

- No sufficiently diverse hardware was evaluated. Only evaluated on Intel CPUs, ARM CPUs, and NVIDIA GPUs. Can Ansor work well on other GPUs (such as AMD GPUs) or TPUs?
- Only evaluated common deep learning operators such as C1D, GMM, etc, but did not consider user-defined operators. Can Ansor optimize user-defined operators?
- Only evaluated the ConvLayer and TBG subgraphs, which is very limited. Can the Ansor work well on other subgraphs?
- In the learned cost model, the fitness only considers throughput. In actual deployment, more factors need to be considered, such as memory consumption, energy consumption, etc.
- The scenario of multi-GPU or distributed environments has not been considered. For example, model parallelism, data parallelism, communication overhead, synchronization, etc.

Related Research

- This paper was influenced by previous works:
  - Automatic generation based on scheduling languages: Halide [7], TVM [5], etc.
  - Search-based compilation and auto-tuning: Stock[8], OpenTuner[2], etc.
  - Polyhedral compilation models: Tiramisu [3], TensorComprehensions [10], etc.
  - ...
- This paper (2nd generation auto tuner. 1st: AutoTVM [5]) influenced subsequent works:
  - MetaSchedule [9] (3rd generation auto tuner. Introduced modularity based on DSL)
  - ML2Tuner [4] (Potential 4th generation auto tuner. Multi-level machine learning-guided)
  - LoopTune [6] (Potential 4th generation auto tuner. Based on reinforcement learning)
  - ...

Impact & Possible Impact

- Impact:
  - This paper was accepted by OSDI 2020 [12]
  - This paper has been cited 578 times
  - Ansor integrated into Apache TVM as AutoScheduler [11]
- Possible Impact:
  - Other areas of compilation may be inspired to explore automated high-performance program generation, such as traditional compilers
  - Other areas that rely on manual templates may be inspired to move towards automation, such as system configuration
  - Application deployment may involve increasing automation optimization

Possible Discussion Questions

- How to achieve an automatic high-performance program search framework for traditional compilers (C++, Java, Rust)? What are the challenges?
- How to extend Ansor to support distributed environments? How to train a learned cost model suitable for distributed cluster environments at a low cost?
- How to extend Ansor to be aware of hardware features? How to make the learned cost model become accurate faster by utilizing hardware information?
- How to extend Ansor to support multi-objective optimization? (Not only throughput, but also memory consumption, program size, etc)
- Is it possible that random annotations and evolutionary search could introduce subtle, hard-to-detect errors in tensor programs?

Thanks

Thank you for listening

References

[1] 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. 2019. Learning to optimize halide with tree search and random programs. ACM Trans. Graph. 38, 4, Article 121 (July 2019), 12 pages. doi:10.1145/3306346.3322967

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. In Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (Edmonton, AB, Canada) (PACT '14). Association for Computing Machinery, New York, NY, USA, 303–316. doi:10.1145/2628071.2628092

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

[4] JooHyoung Cha, Munyoung Lee, Jinse Kwon, Jemin Lee, and Yongin Kwon. 2025. Multi-level Machine Learning-Guided Autotuning for Efficient Code Generation on a Deep Learning Accelerator. In Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (Seoul, Republic of Korea) (LCTES '25). Association for Computing Machinery, New York, NY, USA, 134–145. doi:10.1145/3735452.3735538

References

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (Carlsbad, CA, USA) (OSDI'18). USENIX Association, USA, 579–594.

[6] Dejan Grubisic, Bram Wasti, Chris Cummins, John Mellor-Crummey, and Aleksandar Zlateski. 2023. LoopTune: Optimizing Tensor Computations with Reinforcement Learning. arXiv:2309.01825 [cs.LG] https://arxiv.org/abs/2309.01825

[7] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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 (Seattle, Washington, USA) (PLDI '13). Association for Computing Machinery, New York, NY, USA, 519–530.doi:10.1145/2491956.2462176

[8] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (Houston, Texas, USA) (ASPLOS '13). Association for Computing Machinery, New York, NY, USA, 305–316. doi:10.1145/2451116.2451150

References

[9] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor program optimization with probabilistic programs. In Proceedings of the 36th International Conference on Neural Information Processing Systems (New Orleans, LA, USA) (NIPS ’22). Curran Associates Inc., Red Hook, NY, USA, Article 2593, 14 pages.

[10] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730 [cs.PL] https://arxiv.org/abs/1802.04730

[11] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, and Cody Hao Yu. 2021. Introducing TVM Auto-scheduler (a.k.a. Ansor). https://tvm.apache.org/2021/03/03/intro-auto-scheduler. Accessed on Nov 25, 2025.

[12] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: generating high-performance tensor programs for deep learning. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI’20). USENIX Association, USA, Article 49, 17 pages.