Ansor: Generating High-Performance Tensor Programs for Deep Learning

Original Paper by Zheng et al.

Farin Lippmann
farin.lippmann@uni-jena.de
Friedrich Schiller University Jena
Jena, Thüringen, Germany

ABSTRACT

-+-+-+- TODO: WRITE ABSTRACT -+-+-+-

CCS CONCEPTS

• **Software and its engineering** → *Software notations and tools*; • **Computing methodologies**;

KEYWORDS

Deep Learning, Tensors, Optimization, High-Performance Computing, Auto-Tuning

ACM Reference Format:
Farin Lippmann. 2024. Ansor: Generating High-Performance Tensor Programs for Deep Learning: Original Paper by Zheng et al.. In *Proceedings of (Tensor Computations Seminar).* FSU Jena, Thüringen, Germany, 3 pages.

1 INTRODUCTION

Deep neural networks (DNNs) have many real-world applications with critical bounds for their inference latency (e.g. autonomous driving, real-time translation). With many different popular network and hardware architectures, optimizing performance with a hand-written low-level implementation is not feasible. Because of this, multiple attempts have been made to automate the creation of low-level implementations for tensor operations commonly used in deep neural networks.

These attempts use a search-based strategy, where a space of possible implementations of a given high-level description of a program is explored in order to find one with optimal performance. Existing solutions, however, have two drawbacks. They:

(1) limit the explorable search space by using predefined templates, which fix most of the program structure in place, or
(2) aggressively prune the search space during search.

Both exclude possibly more performant programs from being found.

The paper[2] by Zheng et al. introduces Ansor, a novel approach that manages to avoid these drawbacks by organizing the search process hierarchically, decoupling higher-level program structure and lower-level details. They sample complete programs from the search space, then optimize them further using an evolutionary search strategy, evaluating them using a learned cost model. To optimize the ratio of performance gain to search time, they dynamically prioritize the most performance-significant subgraphs of the to-be-optimized DNN.

They evaluated their implementation on deep learning benchmarks both against solutions that employ manual optimization and ones that also use search-based strategies. Their results show that Ansor often finds solutions outside of the search space of existing approaches and that Ansor searches more efficiently than them, despite its larger search space.

Note: All figures are taken from or based on figures from the original paper [2] by Zhen et al.

2 BACKGROUND

3 ANSOR'S DESIGN

Ansor works by optimizing one subgraph of a DNN at a time. These subgraphs are generated using the operator fusion algorithm from Relay [1]. Each subgraph is then passed through the major components of Ansor, which we discuss in this section:

(1) the program sampler
(2) the performance tuner
(3) the task scheduler

We introduce each of them and explain how they help to find efficient DNN subraph implementations in as short a time as possible. For a preview of Ansor's architecture, view Figure 1.

3.1 Program Sampling

Ansor's search for highly performant programs is split into two parts. First, a number of programs is randomly generated, in order to cover as much of the search space as possible. These programs are then optimized in a fine tuning step. Program sampling covers the first step of this process.

To randomly sample programs, Ansor first generates high-level program sketches, which determine the structure of the programs, but leave finer details undefined. These details, such as loop unrolling, parallelization, tile sizes, etc. are then filled in by random annotations. As such, program sampling consists of sketch generation and random annotation.

3.1.1 Sketch Generation. The program sampler gets DNN subgraphs as input. Each of these subgraphs is represented as a directed acyclic graph (DAG), with each node in the DAG corresponding to one step in the computation. The DAG may be viewed as representing the naive implementation of a subgraph, as can be seen in Figure 2.

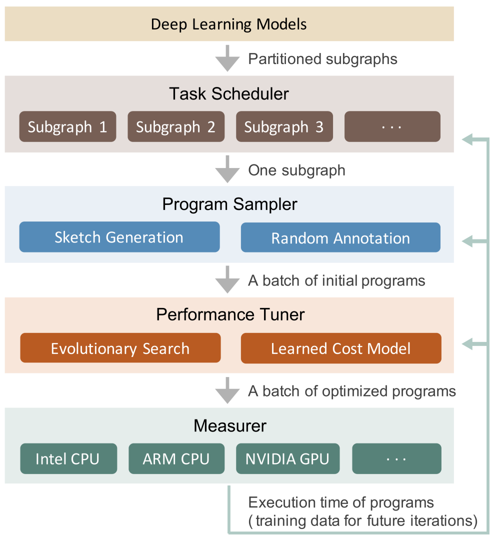

Figure 1: Design overview of Ansor. A DNN is partitioned into subgraphs. The task scheduler allocates time resources to the optimization of each subgraph. Optimization of one subgraph is done by passing it through the program sampler, performance tuner and measurer.

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 $0 \leq i,j,k < 512$

* **The corresponding naive 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:**

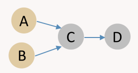

Figure 2: Example of input to the program sampler, viewed in its three equivalent representations (mathematical expression, naive program, DAG).

Ansor traverses this DAG representation backwards, beginning from the last node. At each node, it tries to apply a number of derivation rules that transform the DAG. Each rule that can be applied creates a new branch of possible sketches. An overview of the applicable rules for CPU optimization can be viewd in Table 1.

Using these derivation rules, Ansor generates all possible sketches (typically less than a dozen) from a given subgraph DAG. These define the programs structure, but leave details such as tile sizes and loop annotations open, as can be seen in Figure 3. These details get filled in by the next step of program sampling.

TODO: Maybe add an example here.

Table 1: Sketch derivation rules when optimizing for CPU. Each condition pertains to the current node.

No	Rule Name	Condition
1	Skip	Is not inlinable
2	Inline	Is inlinable
3	Multi-level Tiling	Has data reuse
4	Multi-level Tiling with Fusion	Has data reuse and a fusible consumer
5	Add Cache Node	Has data reuse and no fusible consumer
6	Factorize Reduction Loop	Has more reduction in parallel

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)
```

Figure 3: A sketch generated from the input in Figure 2. Note the variable tile sizes.

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)
```

Figure 4: Two fully functional programs that resulted from randomly annotating the sketch in Figure 3. All loops with length one have been left out.

3.1.2 Random Annotation. In this step, the sketches get completed to become fully functional programs. Tiles sizes are randomly (but validly) chosen, some outer loops are parallelized, some inner loops vectorized or unrolled. Where valid, nodes may be reordered. All annotations are chosen from a uniform distribution of all possible annotations. Figure 4 shows example programs after random annotation.

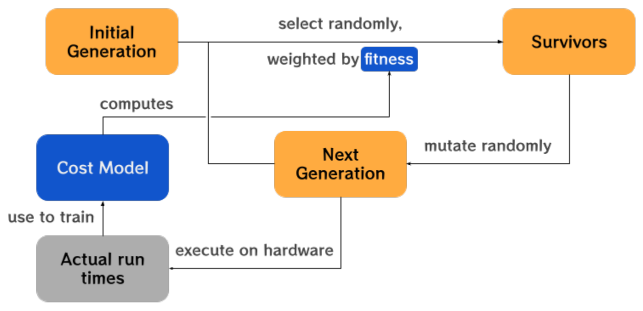

Figure 5: Caption

3.2 Performance Tuning

The programs sampled during the previous step cover the search space well, but are not guaranteed to be performant. The goal is now to use the samples as starting points for an iterative optimization procedure that fine tunes their performance.

To accomplish this, Ansor uses an evolutionary algorithm and a learned cost model.

3.2.1 Evolutionary Search.

3.2.2 Learned Cost Model.

3.3 Task Scheduling

4 RESULTS

5 CONCLUSION

REFERENCES

[1] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A High-Level Compiler for Deep Learning. arXiv:1904.08368 [cs.LG]

[2] 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.