Improving Kernel Autotuning with a Learned Cost Model
A `tiny` Study

Diya Lakhani [1] **Paul Talma** [2]

1. Background: Optimizing Compilers for Machine Learning

Training[1] and running machine learning models requires vast amounts of parallel computation. Graphics Processing Units (GPUs) possess vast amounts of parallel computing power. The current success of machine learning methods owes much to the happy marriage between the compute requirements of ML models and the compute capabilities of GPUs.

Programming GPUs is notoriously difficult, however. To achieve high performance, software engineers must tailor their code to specific hardware features of their target device. Doing so requires deep knowledge of hardware architecture and mastery of the corresponding programming model. Moreover, the resulting code is highly device-specific, so that the work must be repeated for each targeted device. For example, the optimal loop unrolling factor in a matrix multiplication kernel depends on the width of the target GPU's shared memory bank, the amount of shared memory available, and the number of registers per processing unit, all of which may differ across hardware generations and vendors.

Machine learning compilers aim to reduce the software engineering burden of writing high-performance parallel programs (also called *kernels*). These compilers take as input a relatively abstract model specification, usually a graph describing computations to be performed and dependencies between them. They then automatically produce performant code implementing the model. Most ML compilers can emit code for a variety of hardware backends. ML compilers therefore fulfill the traditional functions of a compiler: abstraction, performance, and portability.

To emit efficient code, ML compilers perform a variety of optimizations. *Graph-level* optimizations target the computation graph. The most important graph-level optimization is *fusion*, which partitions the graph into *kernels*. Each kernel defines a GPU program; running the whole model consists in running a number of kernels in sequence. *Kernel-level* optimizations take a single kernel as input and return an optimized version of the kernel. Kernels are usually organized as a nest of loops that iterate over the axes of the tensors involved in the computation. The performance of a kernel depends on the structure of its loop nest and the details of the hardware it runs on. For example, a kernel computing the matrix product $C = AB$ must compute the dot product of the ith row of A with the jth column of B for each element C_{ij} of the output matrix. Doing so involves iterating over three axes: the output rows, the output columns, and a reduction axis for the dot product. The order in which these axes are iterated has a significant impact on the performance of the kernel. The "naive" loop ordering,

```
for i in range(I):
    for j in range(J):
        for k in range(K):
            C[i, j] = A[i, k] * B[k, j]
```

tends to be slower than the following loop, in which the last two iteration axes have been exchanged:

```
for i in range(I):
    for k in range(K):
        for j in range(J):
            C[i, j] = A[i, k] * B[k, j]
```

The latter loop ordering ensures that all tensors are accessed in row-major order. Due to caching behavior, the row-major pattern results in fewer accesses to the GPU's main memory, improving performance.

While this illustrative optimization is well-known, other optimizations are more difficult to find. In addition, different optimizations exhibit complex interdependencies. For example, while loop unrolling and vectorization each usually improve kernel performance, applying both at once may degrade performance (by increasing register pressure and hence decreasing occupancy).

ML compilers discover performant kernels by automatically searching over the space of possible loop optimizations. A brute-force strategy for finding an optimal kernel would generate each possible kernel and measure its runtime, choosing

[1]Samueli School of Engineering, University of California, Los Angeles [2]Department of Philosophy, University of California, Los Angeles.

[1]Project code available here: https://github.com/paul-talma/tinygrad_beam_search.git

the kernel with lowest runtime. This strategy is infeasible in practice: the search space for a given kernel may contain billions of possible optimized kernels. Compiling and running each would be prohibitive. ML compilers must find some way of pruning the search space.

2. Prior Work

There are broadly two ways of rendering the search problem tractable: restrict the search space a priori, or selectively explore the space based on runtime information. Pytorch and XLA take the first route, using heuristics to restrict the search space to a tractable number of candidates. Apache TVM, Ansor, and tinygrad take the second, using methods from guided search to avoid searching the whole optimization space.

Pytorch's compiler, TorchInductor, performs exhaustive search over a relatively restricted set of kernel candidates provided by various backends (such as Triton [4] or cuBLAS/cuDNN [2]). This avoids a combinatorial explosion as each backend provides a limited number of implementations with a small handful of parameters each. Optionally, the winning candidate can be further fine-tuned through parameter-wise tuning. This greedy approach likewise avoids sampling the full Cartesian product of the optimization axes.

XLA, another popular ML compiler, likewise performs exhaustive search on a hand-curated list of kernel candidates, with optional refinements of the best candidates [3].

While these approaches achieve good performance, they are difficult to maintain, relying as they do on the specialized knowledge of compiler engineers.

An alternative approach is presented by Apache TVM and Ansor [1, 5]. Both use a learned cost model to efficiently search the space of kernel optimizations. The cost model is trained to identify faster kernels on the basis of features such as loop nest depth, loop ordering, operations performed inside the loop, device characteristics, and so on. Using a model to estimate a kernel's runtime is much faster than running the kernel. Thus, the cost model allows TVM and Ansor to quickly "screen" a large number of potential kernels, retaining and timing only the most promising few. Both frameworks use the cost model as part of an evolutionary search strategy, which performs random transformations on promising candidates, though the details differ. Since the cost model is trained on readily available data, it automatically adapts to new architectures and hardware characteristics.

Since tinygrad will be central to our approach, we describe it more fully in the next section.

3. tinygrad

For a sequence of Tensor operations with at least one realized Tensor, the tinygrad scheduler produces a sequence of ASTs, where each AST corresponds to a kernel that must be executed. The beam search algorithm will be applied to each AST.

The objective of the beam search is to identify a set of optimizations, if any, that minimize the kernel's execution time. For this process, tinygrad defines a search space consisting of 193 possible optimization transformations, summarized in Table 1.

Table 1. tinygrad Beam Search Optimizations

Optimization	**Factors (or Argument)**	**Axes**
`UNROLL`	0, 4, 7	`range(5)`
`UPCAST`	0,2,3,4,5,7	`range(8)`
`LOCAL`	2,3,4,8, ... 29	`range(6)`
`GROUPTOP`	13, ... 256	`range(3)`
`GROUP`	0,4,8,16	`range(3)`
`PADTO`	32	`range(7)`
`TC`	-	`range(9)`
`SWAP`	pairwise	`range(5)`
`THREAD`	2, ... 64	`range(3)`

`UPCAST` performs vectorization by moving elements into an inner dimension. `UNROLL` unrolls loops to reduce loop overhead. `LOCAL` maps computation to local/shared memory. `GROUP` and `GROUPTOP` partition reduction work across thread groups. `PADTO` pads dimensions to improve alignment. `TC` applies tensor-core-specific transformations. `SWAP` permutes loop axes. `THREAD` maps computation onto hardware threads.

Each optimization in tinygrad is defined by a factor or argument and an axes value. For instance, for `UNROLL`, the argument would be the unroll factor for the loop and the axes is to specify which loop in the kernel we are unrolling. For `TC`, the argument contains tensor-core optimization parameters, while the axis specifies the kernel dimension to which the optimization is applied.

At the first iteration of the beam search, our candidate set only consists of the base AST kernel produced by the scheduler. All applicable optimizations are independently applied to this base AST to produce a list of candidate kernels. Each candidate is then compiled and executed, and the top k candidates are selected based on execution time, where k denotes the beam width. Thus, each iteration of beam search (after the first) compiles and runs at most $k \times |O|$ kernels, where O is the set of legal optimizations.

In order to avoid paying the full cost of executing each kernel, tinygrad adopts some strategies to filter kernels and

estimate execution times. Firstly, candidates whose total operations count is $1000\times$ the current least number of total operations observed are removed from consideration. Second, runtime evaluation is performed using reduced launch dimensions, effectively executing a smaller instance of the same kernel. The measured runtime is subsequently scaled to estimate the execution time of the full kernel. The top k candidates identified during this evaluation stage are retained, and all available optimizations are applied to each of them in the next beam search iteration. The algorithm terminates when none of the newly generated candidates demonstrates an improvement over the runtime of the current best-performing kernel. We found that beam search tended to terminate after 3 to 7 iterations.

Although, tinygrad has strategies in place to avoid paying the full cost of execution, the time required to compile and execute each candidate, even on a small scale, can accumulate as beam depth increases and more legal optimizations are uncovered at each layer. Additionally, according to our experiments, for kernels like `matmul` and `conv`, fewer than 1% of the candidates get filtered out by the flops count check.

Our project aims to improve the speed of the tinygrad autotuner, reducing compilation times and enabling novel search strategies.

To make this better, we select a small set candidates pre-compilation using a cost-model. The cost-model will take in a list of all the candidate kernels pre-compilation and rank them by execution time. We compile this subset of candidates, measure the runtimes, and select the top-k for the next iteration.

4. Methodology

4.1. Overview

Inspired by TVM and Ansor, we design a cost model to evaluate tinygrad kernels and use this model to guide beam search. Instead of executing and measuring each candidate kernel at each beam search iteration, we use the cost model to rank the candidate kernels by estimated execution time. We retain the top $m \times k$ of kernels, as ranked by our model, where m is a pruning factor (in our experiments, $m = 4$). We then compile and run these kernels to get accurate runtime measurements, and retain the top k kernels, as ranked by measured runtime, for the next iteration of beam search. In our experiments, $k = 3$.

4.2. Data

The data to train the model was extracted from the existing tinygrad autotuning pipeline. The existing pipeline already explores a significant number of optimizations and records their execution times. We captured a total of $51,124$ (kernel, runtime) pairs across 56 distinct high-level operations (e.g. `conv_3x3`, `softmax`, etc.). All relevant information about a kernel is encoded in the corresponding AST. The data are hierarchically organized, first by kernel and second by beam depth. This is because the model ranks proposed kernels by estimated runtime. Ranking a matrix multiplication kernel against a softmax kernel would teach the model very little. Likewise, we group the data by beam depth lest we compare proposed kernels with one optimization to ones with several optimizations applied. Doing so would lead the model to learn that more optimizations is always better, at the expense of learning which optimizations to apply.

Once the raw data were collected, we extracted a number of features from the ASTs to serve as inputs to the model. Such features included the number of axes of each type (reduction, iteration, etc.), the datatype of the input tensors, the type of arithmetic operation (ADD, FMA, etc.).

To avoid contaminating training and model selection, we split the data into training, validation, and testing sets by kernel type. We reserved data from the following kernels for testing:

`matmul_256`	`elem_relu`
`matmul_1024`	`elem_fused`
`matmul_2048`	`reduce_sum`
`matmul_4096`	`attention`
`matmul_8192`	`matmul_3072`
`matmul_rect`	`layernorm_large`
`conv_3x3`	`gelu_fused`
`conv_5x5`	

Note that the `attention` kernel is in fact several kernels (matmul, softmax, etc.).

4.3. Model

Since the primary goal of our project is to enable faster beam search, choosing a model with low inference latency was important. We therefore followed TVM and Ansor in choosing a gradient-boosted tree model. Specifically, we used the Light GBM implementation of gradient-boosted trees trained on the lambdarank objective.

To evaluate our model, we monkeypatched tinygrad's beam search module as described above: instead of compiling and running each candidate kernel, we call the model on the set of candidates and run only the top $m \times k$. We then measured two key metrics:

- beam search time (s)
- kernel runtime (μs)

Our goal is to substantially reduce beam search time while sacrificing as little as possible in kernel runtime.

We mentioned above that tinygrad's current autotuner does not measure the true runtime of a kernel but instead uses an estimate based on the measured runtime of a kernel with reduced launch dimensions. We implemented our pipeline with two settings: the first uses the true measured runtime of a kernel, while the second uses a similar estimate as tinygrad. Measuring full runtimes is an option for us since we need to compile and run far fewer kernels than tinygrad's current optimizer. Nonetheless, we were curious to see whether using the runtime estimates would cut the search time significantly, and whether it would impact the quality of the kernels we discovered. We therefore report results for both settings.

5. Results

Our results are presented in Table 2. The main takeaways are as follows:

- Beam search always beats heuristics: the baseline beam search always produces a faster kernel than the heuristics do, by up to a factor of 3. This result vindicates a data-driven approach to kernel selection.
- Our model significantly speeds up the beam search: using estimated runtimes during the search, our model halves the beam search time. Using full measured runtimes results in slightly slower search, though the gains remain substantial.
- The model needs full measured runtimes: using estimated runtimes, the model consistently makes poor decisions, only matching or improving on the baseline performance on three out of the eight tested ops, and fails egregiously on matmul_4096. By contrast, using full measurements, the model appears to match or exceed the performance of the baseline search (though the difference may not be statistically significant).

Overall, our experiment appears successful: our strategy improves search times significantly at no cost to kernel quality.

Looking at some of the results in more detail, we find that our model achieves more significant speedups on ops that take longer to search. Thus, while the average speedup is 46.5%, the gains are greater than this number suggests. For example, we reduce the search time for conv_3x3 from 704.0 to 68.4 seconds, a 90.3% speedup. We note that in one case (matmul_4096), our model-guided search took longer than the baseline beam search.

6. Limitations and Future Work

The principal limitation of our results is their low statistical power. Due to time constraints, we were not able to run many experiments. Anecdotally, we observed moderate variations in runtimes and search times across runs. Thus, we caution against placing a lot of weight on our results. Further work should repeat our experiments several times to achieve better statistical power.

Another limitation of our work is that we had to exclude tensor core optimizations from the entire process. This was due to severe timing inconsistencies for tensor core operations on our local (Apple M2) machines and the unavailability of tensor cores on the tetracosa GPU. We were thus unable to collect reliable data on tensor core runtimes. This limitation is significant, as tensor core optimizations are among the most powerful for matmul-style operations. However, we see no reason why our framework would fail to extend to such cases.

Another area for future work is to leverage our ability to search the optimization space more rapidly, in order to search a wider region of the space. We could increase the beam width, although early experiments with the baseline beam search indicated that increasing beam width beyond $k = 3$ has almost no effect on kernel performance. Alternatively, we could change the search algorithm. A particularly interesting option would be to use beam search with an iteration depth of $d > 1$. This would involve applying d optimizations simultaneously in each iteration of the search. While this greatly increases the complexity of the search space (each iteration must now consider about $k \times |O|^d$ optimizations, instead of $k \times |O|$), it would allow us to discover pairs (or d-tuples) of optimizations such that neither works very well alone, but which offer good performance gains when applied together.

References

[1] T. Chen, T. Moreau, L. Zheng, M. Cowan, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. 2018.

[2] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. URL http://arxiv.org/abs/1410.0759.

[3] A. Sabne. Xla : Compiling machine learning for peak performance, 2020.

[4] P. Tillet, H. T. Kung, and D. Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019,

	Heur	Baseline (B)		Model-Full (F)		Model-Est (E)		Quality Δ	
Kernel	(µs)	srch (s)	kern (µs)	srch (s)	kern (µs)	srch (s)	kern (µs)	B→F	B→E
matmul_1024	487	225.3	309	123.3	324	109.9	339	+4.6%	+9.6%
matmul_4096	25 825	170.5	14 318	216.9	13 835	19.9	232 839	-3.4%	+1526.2%
conv_3x3	202	704.0	142	68.4	126	52.0	168	-11.5%	+18.0%
conv_5x5	–	315.8	126	82.1	127	82.3	135	+0.8%	+7.3%
attention	62	167.2	61	20.9	62	66.1	52	+1.7%	-15.0%
elem_relu	319	19.3	155	9.7	152	22.1	156	-2.0%	+0.7%
layernorm_1024	67	24.7	53	15.6	49	5.8	61	-7.7%	+15.3%
reduce_rows_large	296	15.1	95	12.8	93	19.6	95	-2.2%	+0.0%
Search Δ				-46.5%		-49.9%			
Quality Δ								-2.5%	+195.3%

Table 2. Beam search validation on 8 kernels (beam width = 3, prune factor = 4). **Heur**: hand-coded heuristic, no beam search. **B**: vanilla beam search. **F**: model-guided, full timing (`allow_test_size=False`). **E**: model-guided, estimated timing (`allow_test_size=True`). Search time in seconds; kernel time in µs, re-timed at full size. $\Delta = (\text{model} - \text{B})/\text{B}$; teal = improvement over B, red = regression.

pages 10–19. Association for Computing Machinery. ISBN 978-1-4503-6719-6. doi: 10.1145/3315508.3329973. URL https://dl.acm.org/doi/10.1145/3315508.3329973.

[5] L. Zheng and C. H. Yu. Ansor: Generating High-Performance Tensor Programs for Deep Learning.