UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Discovering Performant Tensor Programs with Bayesian Optimization

Felix Jonathan Rocke

St Edmund's College

June 2024

Submitted in partial fulfillment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 59

Main chapters (excluding front-matter, references and appendix): 40 pages (pp 1–40)

Main chapters word count: 14876

Methodology used to generate the word count:

```
$ gs -q -dSAFER -sDEVICE=txtwrite -o - \
   -dFirstPage=9 -dLastPage=48 report-submission.pdf | \
egrep '[A-Za-z]{3}' | wc -w
14876
```

Declaration

I, Felix Jonathan Rocke of St Edmund's College, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this project report and the work described in it are my own work, unaided except as may be specified below, and that the project report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this project report I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my project report to be made available to the students and staff of the University.

Signed: 

Date: 2024-06-02

Abstract

Discovering Performant Tensor Programs with Bayesian Optimization

Machine Learning (ML) models require high-performance tensor programs for optimal inference latencies. Traditional compilation systems rely on hardware-specific libraries, which include handwritten high-performance operator primitives. However, due to the libraries' substantial engineering requirements, they often struggle to keep up with the hardware development cycles and the evolution of the AI landscape. *Deep Learning Compilers* aim to address this challenge by creating a search space of semantically equivalent programs, i.e., programs that produce the same output but differ in their internal structure. The created space is then searched for a performant implementation. However, as the search space often includes billions of programs, an efficient search strategy is required to minimize the number of programs that must be benchmarked until an efficient implementation is discovered.

Bayesian Optimization (BO) is a sample-efficient sequential design strategy for the optimization of expensive to evaluate objective functions. BO has previously been used, e.g., for hyperparameter optimization problems and material discovery [27, 44], and has shown to be more sample-efficient than the *Genetics Algorithm Evolutionary Search* (ES) [24]. In this thesis, we implement BO as a novel search strategy into the compiler Apache TVM [7] and its scheduling system MetaSchedule [43] and evaluate it against the compiler's state-of-the-art ES strategy.

In the evaluation, we find that our BO search strategy can discover efficient tensor programs with significantly fewer trials than TVM's ES strategy. When generating code for CPU targets, our search strategy delivers an up to 10% decrease in end-to-end latency when using the same number of trials, corresponding to a reduction of up to 68% in trials over the state-of-the-art. For GPU targets, the performance of the BO strategy is limited by the unique characteristics of the black-box objective function, which we analyze in detail.

Acknowledgements

This project would not have been possible without the excellent support and guidance of my supervisor, Eiko Yoneki, and co-supervisor, Guoliang He. I also want to extend my gratitude to the fantastic friends I have made over the last few months. Their support and camaraderie have been an immense source of inspiration. Finally, I want to thank my family for their unwavering support, which made this project possible.

Contents

List of Figures

Chapter 1

Introduction

With the widespread adaptation of machine learning (ML) systems across research, consumer, and industrial applications, the number and variety of devices on which ML models are being deployed has increased significantly. The target hardware ranges from highly specialized data center Tensor Processing Units (TPUs) to mobile and edge devices, which are increasingly important in the broader adoption of localized ML applications [19]. Ensuring optimal inference latency requires high-performance tensor programs tailored to the microarchitecture of the target hardware. However, providing optimized and performant tensor program implementations across a broad spectrum of operators and hardware platforms poses a substantial challenge. Many semiconductor manufacturers offer hand-optimized libraries, such as Intel with oneMKL [21] or NVIDIA with cuDNN [9], containing hardware-specific operator primitives fine-tuned for the respective microarchitectures. However, these libraries often struggle to keep pace with hardware development cycles and the rapid evolution of the AI landscape. Consequently, this has raised the need for a new compilation system capable of generating efficient operator implementations for a diverse set of hardware platforms.

The field of *Deep Learning Compilers* aims to address this challenge by learning the optimal code structure for the target hardware. One of the most popular compilers achieves this by creating a search space of semantically equivalent programs, i.e., programs that produce the same output but differ in their internal structure, for example, in their memory access patterns. As the internal structure decides a program's performance, the idea is to search the space for the most performant implementation. However, the number of potential programs is significant—often in the tens of billions—and evaluating the performance of a program is expensive since benchmarking them on the target hardware is necessary. Therefore, a search strategy capable of minimizing the number of empirical evaluations required until a performant program is identified becomes essential.

Current state-of-the-art search strategies revolve around the *Genetics Algorithm Evolutionary Search* (ES) with search durations ranging from hours to days [8, 55]. ES is a form

of random walk and takes inspiration from evolution by applying random mutations to a population of performant candidates. It builds on the idea that performant candidates are in proximity to other performant candidates. ES is widely used for hyperparameter optimization problems [53], which the formulated problem of discovering efficient tensor programs can be reduced to. Another highly effective parameter optimization strategy has shown to be *Bayesian Optimization* (BO) [4, 44]. BO is often considered the most sample-efficient search strategy regarding the number of evaluations required to find a performant solution [30, 44]. Unlike ES, which relies on random mutations, BO employs a more informed approach by considering previously evaluated points when selecting the next point to evaluate in the search space. BO has been shown to outperform ES regarding the number of evaluations required to identify high-performing solutions [24, 27] and is especially suitable for expensive to-evaluate functions.

We propose the use of BO to discover performant tensor programs in a cost-model-guided optimization. While BO has previously been used to fine-tune templated tensor programs [40, 51], it has not been used for their discovery in a several orders of magnitude larger search space. The properties of BO have the potential to outperform the conventional ES approach as a search strategy for this task, particularly in reducing the number of empirical program evaluations required. This work makes the following contributions:

1. A configurable BO-based search strategy capable of finding high-performance tensor programs built upon Apache TVM [7] and MetaSchedule [43]. Possible configurations include, e.g., acquisition functions, schedule selection policies, and memory eviction policies.

2. A comprehensive comparison between our BO search strategy and the current state-of-the-art ES strategy across different deep-learning models and two hardware targets. This analysis shows that BO requires up to 68% fewer trials to find a performant implementation on CPU targets. Overall, BO surpassed or matched ES across all CPU workloads. In the GPU domain, BO outperformed ES on one model while approaching similar performance on two others.

3. An investigation into the limitations of applying BO to a high noise, discrete, and fluctuating optimization problem in a high-cardinality search space, including an analysis of the characteristics of the black-box objective function—represented by a cost model's prediction of a program's throughput.

Chapter 2

Background

This chapter provides the essential background necessary to understand the core concepts of the deep learning compiler TVM, the generation of semantically equivalent programs, uninformed search strategies, and BO.

We will start by reviewing the high-level concepts TVM employs to lower a model from an ML framework to a diverse set of devices. Section 2.2 will give an overview of the principles used to generate semantically equivalent programs with TVM's newest scheduling system, MetaSchedule. The term *Scheduling* describes the process of transforming an initial program into an optimized program. Subsequently, we will review two uninformed hyperparameter optimization strategies before coming to BO in Section 2.4. Finally, we will provide an overview of related work in the field.

2.1 From ML Framework to Device

TVM is a compiler stack that performs multiple rounds of optimizations to compile a model for a hardware backend. The compilation process begins with the input of a graph-level Intermediate Representation (IR), such as ONNX [33] or TorchScript [36]. This input method allows TVM to work with the most popular ML frameworks, such as PyTorch [35] and TensorFlow [1].

The initial high-level graph IRs, obtained from the frameworks, are then lowered to a unified representation within TVM, which is Relay [39], or its newer counterpart Relax [23]. Both IRs make use of dataflow rewriting optimizations, such as operator fusion. They fuse multiple operators into a single, more performant operator. For example, convolution and batch normalization are often fused into one operation as it can significantly reduce the memory bandwidth requirements, leading to better performance [7].

The optimized graph-level representation is then lowered to TensorIR [12], which allows the specification of hardware-dependent optimizations, e.g., vectorization or memory access patterns. A search space of semantically equivalent programs is generated to explore

these optimizations. A search strategy can then be used to discover efficient programs within the space, which are benchmarked on the hardware. The benchmark results are subsequently used to adjust the search. The search strategies typically involve the use of a cost model, which can approximate a program's performance in a fraction of the time a hardware measurement would take. This allows for a by many orders of magnitude faster exploration of the search space since it significantly reduces the number of empirical hardware measurements required.

Once the search is complete and an efficient implementation has been discovered, it is lowered to the IR of the hardware target's compiler, e.g., LLVM [25], which compiles the program into an executable.

2.2 Program Generation

TVM enables the probabilistic generation of equivalent programs using TensorIR and the scheduling system MetaSchedule. This section introduces the core principles of program generation used in our BO-based search strategy. First, we will review the transformations enabling program rewrites before coming to their use in search space construction. Finally, we will review how we interact with sample instructions. Throughout these sections, we will introduce and explain the concepts needed to understand the interaction of the search strategy with the IR. For a more comprehensive discussion of TensorIR and MetaSchedule, see Feng et al. [12] and Shao et al. [43] respectively.

2.2.1 Transformations

Transformations are one of the core elements of program generation. They allow rewriting an initial program to a new semantically equivalent program. Depending on the program, dozens of different transformation kinds may be possible. MetaSchedule currently offers 30 different transformation primitives. For example, a possible transformation is called `Split`, which splits a loop into a loop nest, creating further optimization opportunities. The pattern with which the loop is extended is passed as a parameter to the transformation. Figure 2.1 shows how the for-loop of a vector addition is extended through a `Split` transformation before being parallelized and vectorized using the `Parallelize` and `Vectorize` primitives.

The `Split` transformation in Figure 2.1 is applied with the decision `[32, 8, 4]`. However, dozens of other valid loop extends exist, such as `[16, 16, 4]`. I.e., by changing the parameters of a single transformation instruction, we can create hundreds of semantically equivalent programs. As we are not limited to a single transformation per program, we may have 20 possible transformation primitives, each with tens to thousands of possible parameters. However, not all transformations are independent; therefore, some parameters are set according to decisions made by previous transformations.

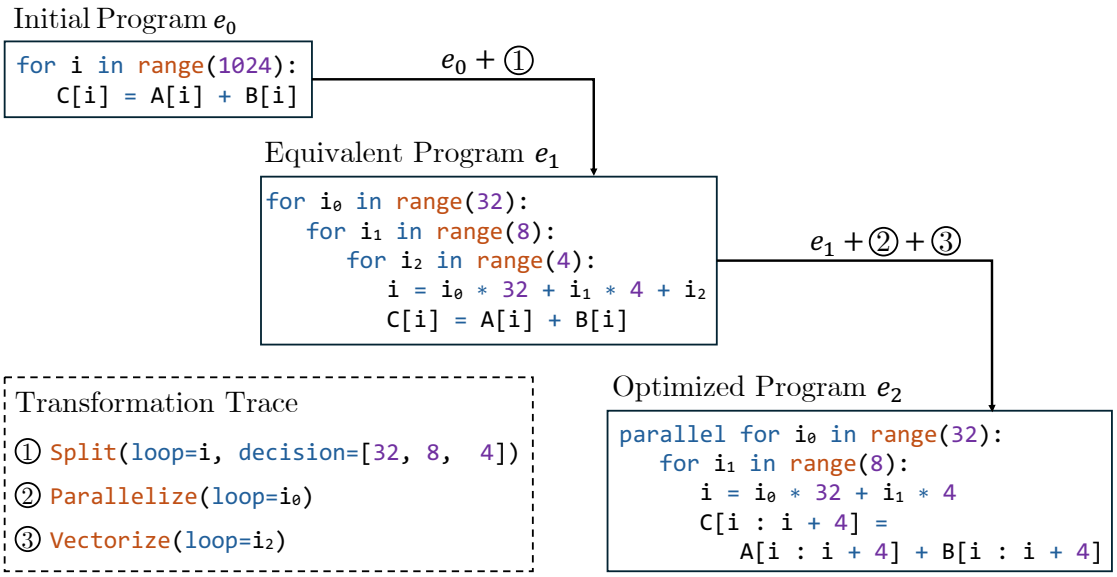


Figure 2.1: Rewriting programs using parameterized transformations. Applying transformation ① to the vector addition e_0 produces program e_1 where loop `i` is extended as parameterized in ①. Applying transformations ② and ③ rewrites the program to include parallelism and vectorization. Inspired by figure in Shao et al. [43].

2.2.2 Search Space Construction

Having gone over how to use parameterized transformations to rewrite programs, we will now review how to use them to create a search space. Optimizing a model as a single program, from input to output, would create an unmanageably large search space since the number of potential rewrite combinations would be too large. Therefore, the first step in the search space generation is partitioning a model into its fused and unique layers (sub-graphs), which we will refer to as *Tasks* or *Workloads*. Consequently, each task is optimized independently.

For each workload, we need to select a sequence of transformations that can be applied to the input workload e_0. This selection of transformations represents the construction of the *Design Spaces* and is of considerable significance since it limits the best program that can be found during the search. Depending on the hardware target, different kinds of transformations will be available. For example, transformations for multi-core parallelism may be used on CPUs, while transformations for thread binding may be applied on GPUs.

MetaSchedule selects transformations through the application of pre-defined rules. The rules decide which parts of the input program remain static, i.e., can not be transformed and which parts can be transformed. The rules produce an ordered sequence of transformations, which is called a *Trace*; compare the *Transformation Trace* in Figure 2.1. As applying one transformation may exclude the use of another, a list of traces is created that represents the feasible combinations of transformation instructions applicable to the program. As such, these initial traces are the design spaces of the workload.

Each design space trace has its own search space of possible parameter combinations for the transformations in the trace. I.e., given a design space trace, we can generate billions of semantically equivalent programs by picking parameter combinations from the search space of possible parameters. By applying the trace, with the selected parameters, to the initial program e_0, we rewrite the program according to the transformations and their parameters. Typically, we have 1 to 3 design spaces per workload, which each enable the generation of billions of programs through the selection of parameters for the transformations.

2.2.3 Sample Instructions

Sample Instructions are transformation primitives just like `Split`. However, they represent the probabilistic part of the transformation. The outputs of sample instructions are typically the input parameters for most other transformations. These instructions randomly generate a possible input value for subsequent transformations in the trace. Therefore, we could replay a trace, and the sample instructions in the trace would generate new random but valid parameters, leading to a new program.

In total, there are three sample instructions: `SamplePerfectTile`, `SampleCategorical`, and `SampleComputeLocation`. Together, they are the main transformations we interact with. While these instructions can sample their decisions, we can also set them to a specific value. Since these instructions are the input for many other transformations, altering their decisions enables the exploration of most of the search space. In the following paragraphs, we will review each sample instruction in more detail.

SamplePerfectTile samples a possible decision on how to extend a loop into a loop nest with the `Split` transformation. The instruction has three attributes and one decision field. The attributes are the loop to be tiled, the number of loops the original loop should be split into, and the largest factor of iterations in the innermost loop. The decision is a list containing the number of iterations of each loop in the loop nest. For example a valid `SamplePerfectTile` instruction could look like this: `SamplePerfectTile(loop=l1, n=3, max_innermost_factor=64, decision=[16, 4, 16])`. Here, loop `l1` is chosen to be divided into three nested loops, where the innermost loop can execute a maximum of 64 iterations per call. The integers in the decision field represent the iteration count for each nested loop, and their product (in this case, 1024) reflects the total iterations of the original loop before the transformation. Thus, the possible decisions the instruction can sample are permutations of the factors that multiply to 1024, with the last decision index not exceeding 64.

SampleCategorical is quite versatile as it selects a value from a list of candidates. Consequently, it is used as input for a variety of transformations. For example, it can be used to select the unroll depth of a loop-unroll transformation. The instruction has

two attributes, the first being a list of possible output values and the second being the probability of each value on the list. The decision is the index of the value chosen from the output list. A valid use of the instruction would be, e.g., `SampleCategorical(candidates=[0, 16], probs=[0.25, 0.75], decision=1)`. This means there is a 75% chance that the index chosen is 1, resulting in a return value of 16.

SampleComputeLocation is similar to `SamplePerfectTile` as it is also a loop transformation used to optimize a program's memory locality. The transformation is used to sample a location in the program's loop nests where two operators, for example, *Dense* and *ReLU*, should be fused. The possible decisions of this transformation depend on the decisions of previously applied transformations. Therefore, it is the last scheduling step.

2.3 Uninformed Hyperparameter Optimization Strategies

In the previous sections, we have described how we can create semantically equivalent programs using a selection of transformation parameters. Therefore, finding an efficient tensor program is reduced to a hyperparameter optimization problem.

The goal of an optimization strategy, in our problem space, is to find a transformation parameter combination τ from the domain of all possible transformations T that maximizes the throughput function $f : \mathrm{T} \to \mathbb{R}^+$ of an initial program e_0. Formally, we want to find:

$$\tau^* \in \arg\max_{\tau \in \mathrm{T}} f(\tau); \qquad f^* = \max_{\tau \in \mathrm{T}} f(\tau) = f(\tau^*) \tag{2.1}$$

There is a variety of established methods for the automatic solving of hyperparameter optimization problems. *Grid Search* (GS) can be an adequate solution for small search spaces since the cost of trying every possible combination is low. However, with increased search space size, GS quickly becomes too time-consuming. An alternative uninformed search strategy would be *Random Search* (RS), which will, on average, find a performant solution in a large search space in less time than GS. However, while RS can often find a performant solution relatively quickly, it does not adapt to the search space characteristics and, as a result, often lacks consistency in the number of evaluations required. Consequently, optimization strategies such as ES and BO are often favored due to their more dependable convergence characteristics.

The following two subsections will provide an overview of how the two uninformed search strategies, RS and ES, work, which advantages they offer, and what drawbacks they might have.

2.3.1 Random Search

RS is an uninformed search strategy that works by uniformly sampling parameters from the search space and evaluating their performance using the objective function. Consequently, it is relatively easy to implement. The simplicity of the search also means that it can be used in discrete and continuous domains without much adaptation. However, in search spaces where optimal solutions present strong locality and only occupy a fraction of the search space, RS will require a significant number of samples. Since the likelihood of finding performant candidates directly correlates with their distribution within the space. As a result, more complex search strategies, which can automatically prioritize performant subregions, are often preferred over RS.

2.3.2 Evolutionary Search

ES is a type of genetic algorithm inspired by the principles of biological evolution. Typical implementations have a population of candidates from which the best are selected using a fitness function. The selected candidates become the parents of the subsequent generation. Following this selection, random mutations are introduced to the parents, generating a new candidate population. ES is also considered an uninformed search since it does not take previous observations into account, when applying mutations. Instead, ES assumes a locality between good candidates, i.e., performant solutions are surrounded by other performant solutions. Therefore, incremental mutations of parameters will lead to better candidates near the parent generation.

With a large population, the candidates can be distributed across multiple regions within the search space. Depending on the selection strategy of the fitness function and the mutation rate, a balance between exploration and exploitation can be achieved. Similar to RS, the search strategy is quite robust in the sense that it needs little adaption for varying problem domains.

2.4 Bayesian Optimization

BO is a sequential and informed design strategy applicable to black-box optimization problems, where the function is only known through observation since no closed-form expression is required. It is typically used to optimize expensive to evaluate objective functions. As an informed search strategy, BO utilizes information learned in previous evaluations of the objective function to select a new point to probe. It has been shown that the number of points probed until convergence to an accurate solution is among the lowest using BO compared to other optimization techniques [29, 30, 44]. BO can be defined as a maximization and minimization strategy. In the following, we will consider the maximization case.

2.4.1 Algorithm

BO works by constructing a probabilistic model of the objective function, which is typically modeled as a *Gaussian Process* (GP). The model can then estimate the objective function's values for a fraction of the cost. Enabling the *Acquisition Function* (AF) to evaluate thousands of points on the probabilistic model before selecting one. We will discuss the different AFs in the next section. The point selected using the AF is then empirically evaluated on the objective function and used to update the GP to improve the model's accuracy for subsequent predictions. Algorithm 1 shows a simplified BO algorithm, which can be used to find a solution to Equation 2.1.

Algorithm 1 Bayesian Optimization

input: set of previously evaluated points $\mathcal{D}$ (possibly empty)

for $t = 1, 2, \ldots$ **do**
  $x \leftarrow AcquisitionFunction(\mathcal{D}_{t-1})$ ▷ Sample point to probe
  $y \leftarrow f(x)$ ▷ Evaluate objective function at x
  $\mathcal{D}_t \leftarrow \mathcal{D}_{t-1} \cup \{(x, y)\}$ ▷ Save result to dataset
end for

return $x^* = x$ where $(x, y) \in \mathcal{D}$ and $y = \max_{(x', y') \in \mathcal{D}} y'$

2.4.2 Acquisition Functions

As mentioned before, a variety of different AFs exist. They select the next evaluation point based on different objectives. The three AFs below are among the most popular in use today [5]:

Probability of Improvement (POI) was suggested by Kuschner et al. [22] and, as the name suggests, maximizes the probability that the next sampled point will be an improvement over the previous best. Since this strategy inherently favors exploitation over exploration, a trade-off parameter ξ is introduced to allow for adjustment between the two.

Expected Improvement (EI), as proposed by Močkus et al. [31], maximizes for the most significant improvement over the current known best. It differs from POI in the sense that it does not consider how likely an improvement is but rather how large the improvement could be. Similarly to POI, a trade-off parameter ξ is chosen to balance between exploration and exploitation.

Upper Confidence Bound (UCB) exploits the upper confidence bound to minimize regret over the course of the optimization [45]. UCB typically is the weighted sum of the GP's posterior mean and posterior standard deviation. The weight is applied to the posterior standard deviation and represents the trade-off parameter κ. A larger κ value leads to more exploration, whereas small factors favor exploitation.

As the three functions have their unique sampling behavior, deciding which to use is often not trivial. A more detailed exploration of the AFs, with their closed-form expressions, can be found in Brochu et al. [5].

2.4.3 Primary Advantages

As an informed search strategy, BO brings some advantages compared to other strategies, making it a better choice for some optimization problems. In the following enumeration, we list some of BO's advantages:

1. BO's main advantage is the previously mentioned sample efficiency, which means that BO tends to require significantly fewer function evaluations than, for example, RS and ES.

2. BO also allows the specification of an exploration-exploitation trade-off. Enabling adjustments to be made during the optimization. For example, the optimization can start with a high degree of exploration and increase the exploitation as confidence in the model grows. This is something other search strategies do not enable as easily.

3. BO can handle uncertainty in the observations by incorporating noise, often referred to as *jitter*, into the kernel, reducing the risk of overfitting to noisy data.

4. BO provides a theoretical convergence guarantee to the global maximum of continuous functions [29]. However, this guarantee is often limited by the computational requirements.

2.4.4 Discrete Bayesian Optimization

As described in the previous paragraph, BO has a theoretical convergence guarantee when the objective function is continuous. However, as we can see in Section 2.2.1, the transformation parameters are discrete, making the objective function discontinuous. With an objective function defined on a discrete domain, further steps must be taken to ensure good convergence performance. Since the GP and the AFs are defined on a continuous domain, the suggested points are invalid parameters for the transformation instructions. While there are approaches to, for example, turn the AF into a step-wise function to predict discrete values or to use a probability distribution of continuous parameters to maximize the AF on, they are often more costly to maximize [10, 28]. The traditional approach is to keep the AF continuous and round the suggested candidate parameters to the nearest discrete value. While this strategy allows the optimization of discrete objective functions, they may converge to suboptimal solutions in high-cardinality search spaces [10, 14, 28]. Another difficulty is that two real values may round to the same discrete value, making reevaluations possible, which are only helpful in high-noise environments.

2.4.5 Computational Complexity

While BO is sample-efficient, it may not be the most time-efficient strategy in every use case. For each point we add to the GP kernel, computational effort is required, and this effort is not constant. Adding an observation, typically done every iteration, requires the inversion of the covariance matrix. This inversion is costly, with a time complexity of $O(n^3)$ where n is the number of considered observations [42]. As a result, there is a trade-off point for specific optimization problems—which depends on how expensive the objective function is to evaluate—where an increase in the number of remembered observations becomes more expensive than making more but less informed decisions. Močkus et al. [30] called this the restricted memory case. It is important to note that this will degrade BO's sample efficiency, as suggestions will be less informed.

2.5 Related Work

The field of deep learning compilers started gaining traction in 2018 with the introduction of TVM, which was initially heavily based on Halide [38], a language and compiler for image processing workloads. TVM's introduction brought with it its first scheduling system, AutoTVM [8]. AutoTVM is template-based and requires domain experts to write high-level templates, which are subsequently fine-tuned. However, the need for templates led to significant engineering requirements, something the compiler aimed to resolve. Additionally, the template-based approach limited performance since it constrained the search space of possible programs. To address these limitations, the next-generation scheduling system ANSOR [55] was introduced, which allowed the generation of programs without templates. MetaSchedule is an evolution upon ANSOR and unifies deterministic transformations with probabilistic search space construction.

Integrating BO into TVM's scheduling has previously been done with AutoTVM [40, 51]. Other compilers have also used BO in their tuning stages [17, 50]. As AutoTVM relies on fine-tuning templated programs, the search space is restricted significantly and, as a result, multiple orders of magnitude smaller than MetaSchedule's. To our knowledge, using BO to discover tensor programs in a search space without template-based restrictions has not been done before.

Besides TVM, other deep learning compilers, such as XLA [41], and other optimization approaches have emerged. MLIR [26] is one of the technologies that has become increasingly popular within the deep learning community. Although MLIR itself is not a compiler, it serves as a foundational infrastructure for numerous compilers and IRs. For instance, OpenAI's Triton [48] language uses MLIR, enabling simplified development of CUDA kernels for NVIDIA GPUs, and is the language behind PyTorch's TorchInductor [3] compiler. Triton also uses auto-tuning; however, it requires a fraction of the search space and tuning time of TVM while achieving performance on par with established oper-

ator libraries for specific hardware targets and workloads [48]. In addition to compilation, other optimization and compression strategies, such as distillation, pruning, and quantization, can significantly lower a model's inference latency with little effect on model accuracy.

Chapter 3

Bayesian Optimization Search Strategy

As discussed in Chapter 2, the problem of exploring a search space of billions of semantically equivalent tensor programs resembles a hyperparameter optimization problem. While MetaSchedule's modularity allows the implementation of a new search strategy without significant changes to the rest of the compiler, incorporating BO into MetaSchedule remains a non-trivial task. This begins with the complexity of TVM itself, which has multiple IRs, runtimes, and outside dependencies while, in many parts, lacking clear documentation compared to other compiler frameworks such as LLVM. Consequently, using the compiler successfully requires significant domain knowledge [52]. This difficulty extends to working on TVM; for example, validating the effectiveness of changes to the scheduling can easily take 144 CPU hours, slowing down development times significantly.

TVM is implemented with a Python frontend for fast prototyping and a C++ backend to maximize performance. We built the search strategy primarily in Python but expanded the C++ bindings with features we could only implement in the backend. In total, we added approximately 3,000 lines of Python and C++ code to TVM to support our BO search strategy, making the following contributions:

1. Implemented BO as a novel search strategy into MetaSchedule, which currently offers two RS and one ES implementation.

2. Expanded TVM's C++ bindings to include features such as static analysis for annotation values and annotation editing from the frontend.

In this chapter, we will go over the search strategy implementation and the design decisions made. We will start with an overview of the strategy's data flow in Section 3.1, followed by an examination of MetaSchedule's cost model. In Section 3.3, we review the optimizer before coming to schedule validation, optimizer memory restriction, schedule selection, and duplicate avoidance. Finally, we will summarize the strategy's possible configurations.

3.1 Strategy Overview

Before going into the finer details of the strategy's implementation, we want to give an overview of the general structure. As seen in Figure 3.1, the search begins with the input of the design spaces for a given workload. As mentioned in Section 2.2.2, design spaces are traces containing the possible sequences of transformation primitives applicable to an input program.

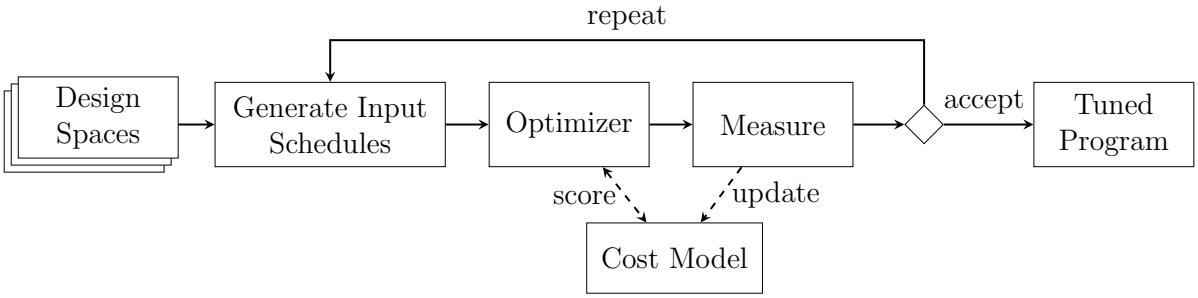


Figure 3.1: Overview Search Strategy.

In the subsequent step, *Schedules* (programs) are created from the traces by selecting parameters for the transformations. This selection is made through two methods: uniformly sampling parameters and picking those with high-cost model scores and choosing from a database of the most performant empirically measured parameter combinations. The balance of selecting measured and unmeasured programs for the creation of input schedules is controlled by a ε-greedy policy [47] with the likelihood of picking a random schedule set to 20%.

The optimizer uses these input schedules to decide the design spaces to search and to gather initial parameter combinations. This explains why we select the most performant schedules as input, as we expect other performant candidates to be in the same space. We use the mix of unmeasured and measured candidates to lower the likelihood of over-exploiting a single design space.

The optimizer can subsequently explore and exploit the search space of each input schedule's design space. To do this, the optimizer suggests a parameter combination and applies it to the transformations in the input schedule's trace. From the edited trace, we can create a new schedule by applying it to the original program e_0. This new schedule can then be scored by the objective function, which is the cost model's throughput prediction. The prediction is registered with the BO optimizer and used in the subsequent parameter suggestions to inform the search.

After a defined number of input schedules and parameter suggestions, a selection of the created schedules is built into executables and benchmarked on the target hardware. The measurement results are used to retrain the cost model for future search phases. With the measurement trials complete and the cost model updated, we can either continue the search or return the best program discovered.

3.2 Cost Model

The learning cost model is an essential part of the search strategy's design, as it allows for quick evaluation of thousands of possible schedules without the need for time-intensive benchmarks on the hardware. The model is based on XGBoost [6], which is a scalable implementation of gradient tree boosting and was initially introduced to TVM with the template-based scheduling system AutoTVM but has also been implemented for ANSOR and MetaSchedule. The XGB cost model is the standard fitness function for ES across TVM's scheduling systems. In our BO search strategy, we utilize the same cost model as ES for the objective function, i.e., we optimize against the cost model since hardware measurements are still too expensive, even with BO's sample efficiency.

The model is trained on features extracted from the generated programs and their corresponding measurement results. Therefore, the cost model evolves and becomes more accurate throughout the search process. Possible features include, for example, the length of the innermost vectorized loop or the number of floating point operations [55]. After training, the model can approximate a schedule's normalized throughput by evaluating a program's features. These predictions, however, are not always accurate and can be biased toward programs closely related to the training data. As the prediction only takes a fraction of the time an accurate hardware measurement would take, the cost model enables orders of magnitude faster search space exploration.

The cost model determines which programs are measured on the hardware based on the scores it assigns, thereby indirectly selecting its own training data. Therefore, it is essential to implement a strategy that maintains diversity in the cost model's training data to prevent overfitting. Accordingly, not all schedules are selected solely based on their scores. Similarly to the balance between measured and unmeasured input schedules for the optimizer, we achieve a balance of optimized and uniformly sampled programs in the measurement set by using an ε-greedy policy with the likelihood of selecting a random schedule set to 20%. This policy ensures that the training data represents the search space and that we do not overfit the cost model.

3.3 Optimizer Overview

Having gone over the overall data flow of the search strategy and the workings of the cost model, we can now have a closer look at the *Optimizer*, compare Figure 3.1, and discuss the implementation in more detail.

The optimizer is structured into three phases, each collectively responsible for tuning all relevant transformation parameters in a schedule's trace. The first and most critical phase is the *BO-Phase*, where `SamplePerfectTile` and `SampleCategorical` transformation parameters are selected. The subsequent two phases fine-tune the parallel annota-

tions and `SampleComputeLocation` instructions in schedules found during the BO-Phase. Both phases require static analysis of the *IRModule's* (TensorIR representation of the transformed program) tiling structure to identify possible parameters. Therefore, their transformations' decisions can only be set after the tiling is decided in the BO-Phase. GPU scheduling only uses the BO-Phase since the two instructions tuned in the other phases do not exist in GPU design spaces. In the following sections, we will review each of the three phases in detail.

3.3.1 BO-Phase

In the BO-Phase, we select the decisions of the two sample instructions `SamplePerfectTile` and `SampleCategorical`. As mentioned in Section 2.2.3, `SamplePerfectTile` instructions are responsible for the tiling structure of the schedule's loops, whereas `SampleCategorical` instructions are the inputs of other transformations, such as the unrolling factor or thread binding. Together, these two instruction kinds represent the primary parameters we need to optimize. A typical trace contains approximately 4 to 8 tunable instructions, resulting in a 4 to 8-dimensional optimization problem with billions of parameter combinations. The BO-Phase is structured as visualized in Figure 3.2. In the following paragraphs, we will walk through the steps and explain them further.

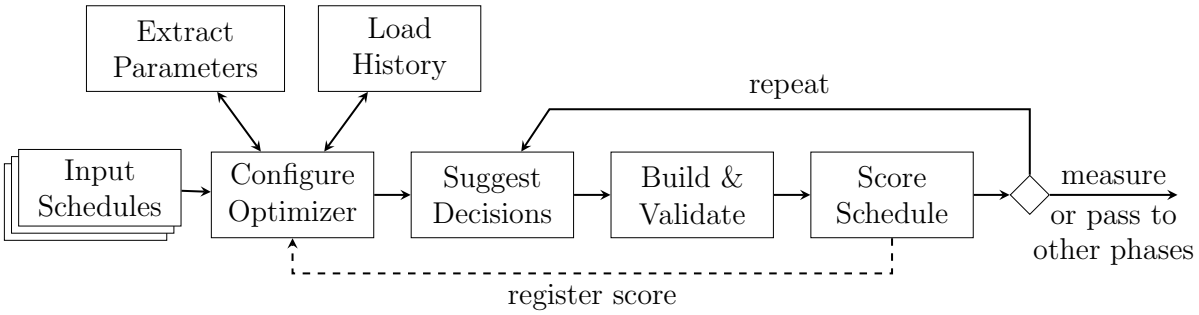


Figure 3.2: Overview BO-Phase.

Input Schedule

The optimization process begins with a selection of input schedules; compare Figure 3.2. These include the highest-performing previously measured schedules and some of the best uniformly sampled ones. Together, they represent a mix of programs with various high-performing parameter combinations from different design spaces.

These input schedules are subdivided into groups based on their design space and the decisions attached to the transformations, which cannot be optimized in the BO-Phase. On GPUs, the groups are equal to the design spaces, as all parameters are tunable in the BO-Phase. All schedules in the same group can use the same optimizer with the same memory of previously evaluated points, as their only difference is in the tunable parameters. To identify the group of a schedule, we serialize its trace, remove post-

processing data, delete all tunable parameter decisions, and create a hash, which is the group's unique identifier.

By using performant schedules as input, we can limit the number of groups and, consequently, the number of optimizers to the well-performing ones. Also, note that the parameters of the input schedules are registered with the optimizer, ensuring the optimizers always know the best-known parameters in the space; we will see in Section 3.5 why this is important.

Extracting Optimization Parameters

For each input schedule group, we need to identify the tunable transformations and their parameter boundaries. Boundaries are the start and end values of the interval from which the optimizer can suggest values. To find the optimization parameters, we analyze the transformations in the schedules' traces and extract details on the tunable transformations and their boundaries.

As we want to share the optimizer between schedules from the same group, we must create a unique identifier for each tunable transformation, which can be matched between their traces and registered with the BO optimizer. To achieve this, we construct a transformation tag, which incorporates the instruction kind, the output parameter names, and transformation attributes, e.g., `[v25,v26,v27,v28]_SamplePerfectTile_4_64` is the tag for a `SamplePerfectTile` instruction where `[v25,v26,v27,v28]` represents the output variables (containing the tiling pattern). The loop attributes, which are the depth of the loop nest and the number of total iterations, are in the last part of the tag, here `4_64`. With this tag, we can identify the equivalent transformation in other traces, i.e., we made transformations comparable between traces of the same workload and group.

Coming to the BO optimizer's suggestion bounds. For `SampleCategorical` instructions, the upper boundary is the length of the category list. Therefore, the optimizer will suggest an index for the list of candidates encoded in the instruction. A possible boundary could, e.g., be $(0, 3)$ if the instruction has four candidates in the list.

The boundaries of the `SamplePerfectTile` instructions are not as straightforward since the decisions must adhere to a strict constraint. As explained in Section 2.2.3, the decision parameter specifies how a loop is extended. For example, the decision `[32, 8, 4]` means that the loop is split into three nested loops with the iteration counts in the decision list. The product of the decision values is the number of iterations the loop had before it was split. Defining this constraint within the optimizer is impractical, as it could only reject points that violate the constraints after being sampled by the AF. This would add significant overhead since most sampled combinations would not satisfy the constraint. Additionally, the number of optimization dimensions would be too great if each factor in the list was a dimension since BO works best with less than 20 dimensions [13].

To address this, we adopted a strategy similar to that used for `SampleCategorical`. Rather than having the optimizer directly propose a perfect tiling pattern, we allow it to suggest an index to a list containing all possible tiling decisions for a loop. Thus, the optimizer's suggestion boundaries for `SamplePerfectTile` become the length of the tiling pattern list. The list is sorted to add some structure to the data since similar patterns share locality within the list. Also, note that we only compute the list once for each tiling constraint since we share it between instructions if they have the same restrictions.

Configure Optimizer

For each input schedule group, we initialize a BO optimizer using the Python library *BayesianOptimization* [32]. We chose this library since we wanted to avoid introducing significant additional dependencies to the compiler. After initializing the optimizer, we can register the extracted parameter boundaries. Subsequently, we check if a history of previously probed points is available for the group; if so, we can continue the optimization from the previous state.

The next step in the configuration is selecting the AF with the respective exploration and exploitation trade-off. An additional configuration we allow is a Sequential Domain Reduction (SDR) strategy. This narrows the search space throughout the optimization and may be helpful for workloads where the objective function is fluctuating or very noisy; see Stander et al. [46] for more on SDR.

Schedule Discovery

With the configured optimizer, we are ready to explore and exploit the search space. As mentioned before, the search starts with the optimizer suggesting a new set of decision parameters from which we build the respective schedule by editing the parameters of the transformations in the trace and applying the trace to the original program e_0. Some parameter combinations may produce a syntactically correct but semantically invalid schedule, e.g., due to changes in the execution sequence. Therefore, we need to post-process the schedule and validate the correctness; we will review schedule validation in more detail in Section 3.4.

After post-processing, the schedules are scored. Schedules that fail validation receive a score of 0, while successful schedules get a score between 0 and 1 from the cost model. The optimizer then uses these results to guide the search in the following iterations. Each input schedule has a maximum budget of successful and invalid suggestions. The optimizer will continue suggesting parameters until either one of the budgets is used. With one input schedule finished, the work on the next input schedule is started, which may involve continuing with the current optimizer if the schedules are from the same group. When the budget for all input schedules is used, a selection of the created schedules is returned; we will go into more detail on how schedules are selected in Section 3.6.

3.3.2 Parallel-Phase

Following the BO-Phase, the *Parallel-Phase* selects the appropriate parallel extent annotation for a block or loop in the IRModule. The annotations specify how deep the block or loop should be parallelized (threaded and vectorized). The possible annotation parameters depend on the tiling parameters chosen. Therefore, analyzing the IRModule becomes necessary, meaning BO can not suggest values for the parallel annotations. Consequently, we tune the annotations in a subsequent phase, which only operates on the best candidate programs found during the BO-Phase. Given the limited number of possible annotations, we can take inspiration from GS and evaluate every possible parallel extent combination for each input schedule. Editing an annotation is similar to editing a transformation. However, we had to extend the C++ bindings to make it work from the Python frontend.

3.3.3 Compute-Location-Phase

The *Compute-Location-Phase* is also only used on CPUs but is slightly more complex than the Parallel-Phase. Similarly to the Parallel-Phase `SampleComputeLocation` instructions depend on the selected tiling pattern and require static analysis of the produced IRModule to find the possible loop locations to fuse operations. As the number of possible compute location combinations is too great to test them all, we take some inspiration from RS to explore the decision space of the instruction kind. We first extract the `SampleComputeLocation` transformations and set a budget of the maximum number of parameters to try. Then, we randomly select a transformation from the list and start mutating the instruction's decision. We continue picking transformations and mutating their decision parameter until the budget is used.

3.3.4 Summary

Together, the three reviewed phases optimize all user-facing transformations in the trace. While the BO-Phase does the heavy lifting, optimizing the most critical transformations, the subsequent two phases still play an important role. These two phases utilize ideas from GS and RS depending on the search space size of the transformations' possible parameters. Besides combining multiple phases, which use different optimization strategies, we extended the frontends' functionality, introduced novel ideas to match transformations between traces, and balanced the exploration and exploitation of design spaces.

3.4 Schedule Validation

After applying a trace to an initial program, we must verify that the TensorIR programs are semantically equivalent, i.e., produce the same output. Therefore, post-processing validators are applied, which filter out non-equivalent schedules. Based on our experience, schedule failures at this stage are minimal to nonexistent when scheduling for a CPU.

Additionally, we need to verify the thread and memory hierarchies on GPUs and accelerators that support threading. This is done via the `VerifyGPUCode` post-processor, which checks, e.g., if the thread binding fulfills the constraints of the hardware backend. Additionally, for cooperative memory access to shared memory, the data must meet the upstream and downstream requirements of all threads within the same group. Finally, the execution scope needs to be verified; for example, TensorCore intrinsics must run at the warp level [12].

Schedule validation is of significant importance for GPU code generation. While the search space of possible GPU programs is significantly larger than that of CPUs, many areas within the space produce invalid programs. When randomly sampling, we observed more than 90% of GPU schedules failing validation on our hardware setup across different workloads, solidifying the need for an informed search strategy.

3.5 Restricting Optimizer Memory

As highlighted in Section 2.4, adding an observation to BO's GP has a time complexity of $O(n^3)$, where n is the number of previously probed points that inform the next suggestion. During a model's optimization, each workload undergoes multiple search and measurement iterations. In a typical search iteration, we evaluate roughly 500 to 1000 parameter combinations (10 - 20 per input schedule) using the cost model. If we continue with the same optimizer throughout all iterations of a workload, point suggestions will quickly become a significant bottleneck. Therefore, we restricted the number of previous evaluations the optimizer can use to inform its decisions. As evaluating a schedule with the cost model is relatively cheap, measuring more schedules with less informed decisions can be more time-efficient than spending more time on more informed decisions. Additionally, we are not necessarily searching for *the highest* scoring schedule but rather a selection of plausibly good schedules since the cost model's accuracy is limited. Besides these reasons, there are two additional justifications that show the importance of having an upper limit to the optimizer's memory.

The first is that we retrain the cost model with the results of each hardware measurement phase. Consequently, the scoring of the cost model evolves throughout the search. For example, a schedule previously scoring high may now score significantly lower since a new best was discovered or a previous prediction was inaccurate. Conversely, a schedule previously not considered worth measuring may now score significantly higher. If we kept all points in memory, the noise levels would quickly be so high that BO would struggle to make good suggestions. The second factor is that discrete BO is more likely to get stuck in local maxima. Restarting the optimization process gives the optimizer a new chance to explore the space. As a result, we give BO the chance to converge to a different maximum on the objective function.

We implemented two possible options for limiting the memory of the optimization. The first is a naive approach, in which the optimizer will be reset when it reaches a specified threshold, i.e., we forget all previously probed points and start the optimization from the beginning. The other approach follows a Least Recently Used (LRU) approach; here, we forget the oldest points when adding new points. This allows us to always make informed decisions while removing points that may only add noise. However, this strategy may get stuck in the same local maxima for an extended time due to a slower exploration of the space. Please note that, as we mentioned before, even if we reset the optimizer, the parameters of the input schedules, which represent the best-discovered points, are registered again to give some guidance. However, these points are so few that the uncertainty in the GP remains high, meaning exploration will take place regardless.

3.6 Schedule Selection

During the optimization, we create thousands of possible schedules, but we only want to measure a selection of them on the hardware. Therefore, we implemented two different strategies for selecting the best candidates. Our naive strategy chooses the best scoring schedule found in the search budget of each input schedule. However, if one input schedule does not produce any high-scoring schedules, we will measure one with a low score, which is a potential drawback, when at the same time, an input schedule that generates multiple high-scoring schedules can only submit a single result. Ultimately, the set of submitted schedules will have lower scores than necessary. Thus, we also implemented a configuration that maintains a heap with the best schedules found across all input schedules and phases.

3.7 Avoiding Duplicate Measurements

As the BO library we use does not natively support discrete BO, we follow the naive rounding approach introduced in Section 2.4.4. To avoid rounding continuous values to the same discrete point, we keep track of what we have explored in the current memory history and skip points we have seen before when they are suggested. However, points not in memory may be suggested again, which is not necessarily bad since, as we discussed before, their score may have changed.

While our implementation of discrete BO can prevent some duplicates, it does not guarantee that we will not select the same schedule multiple times for measurement. As there is no benefit in doing so, we need a method that ensures a schedule is only measured once. The solution we implemented creates a structural hash of every IRModule we submit for measurement. If that hash is already contained in the set of measured programs, we will not measure the schedule again.

3.8 Possible Configurations

In the previous sections, we reviewed the implementation of the BO search strategy and examined the possible configurations we implemented. The main goal behind these different configurations is to better understand the search strategy's requirements by evaluating its interaction with the cost model and search space. This section summarizes the available settings and highlights the primary configurations evaluated in Chapter 4.

The first configuration to consider is the acquisition function and the exploration-exploitation trade-off factor. We offer the choice between POI, EI, and UCB for the AF. The trade-off factor can be selected in correspondence to the AF definition.

Primary Configurations

Configuration	**Policies**	
	Memory Eviction	**Schedule Selection**
BO Naive	Reset	Best per Input
BO + Heap	Reset	Best Overall
BO + LRU	LRU	Best per Input
BO + Heap + LRU	LRU	Best Overall

Table 3.1: Main configurations based on policy combinations.

Another key setting involves limiting the optimizer's memory by selecting the maximum number of points it uses to inform its decisions. As detailed in Table 3.1, we provide two memory eviction policies to manage this limit. The first resets the memory, i.e., we start the optimization from the beginning and forget all previously probed points. The second policy uses an LRU approach, meaning we delete the oldest points in memory when adding new points.

The next setting decides how programs are selected from the discovered schedules for benchmarking on the hardware. We implemented two possible policies for the selection; compare 3.1. One option is to select the best program discovered for each input schedule. The other is to choose the best programs discovered across all inputs using a Heap.

The final setting we implemented is a domain reduction strategy, which narrows the search space throughout the optimization; compare Section 3.3.1.

Chapter 4

Evaluation

In this chapter, we evaluate the BO-based search strategy's performance by comparing it against MetaSchedule's state-of-the-art ES strategy across multiple deep learning models and two hardware targets. The evaluation presented in this chapter required thousands of CPU and GPU hours, during which we collected hundreds of tracing and log files. Analyzing this data to draw conclusions and identify root causes required significant effort. Throughout this evaluation, we aim to examine the following objectives:

1. Whether BO is a viable search strategy for the discovery of high-performance tensor programs.
   (a) Whether BO is more sample-efficient than the ES strategy.
   (b) Whether BO is more time-efficient than the ES strategy.
2. Whether BO is effective across hardware targets and models without significant modification or configuration changes.

The upcoming sections are organized as follows. First, we will review the hardware configuration and benchmarking settings. Then, we will explain the selection of deep learning models used for the evaluation. Afterward, we will explore the results in depth, evaluate the search strategy's performance in relation to the formulated objectives, and address any limitations.

4.1 Evaluation Configuration

This section details the hardware setup, TVM configuration, and search strategy settings to ensure reproducible evaluation results. To support this goal, we included our extended version of TVM and the benchmarking scripts with this submission. To cleanly highlight our contributions, we included a patch file called `our_contribution.patch`, which shows the changes and additions we made to the compiler.

4.1.1 Hardware Configuration

All benchmarks used in the evaluation were conducted on the following two systems. The first system is used to benchmark the performance of the strategies when targeting a CPU. The second system is used to evaluate the search quality when targeting a GPU. An overview of the systems' hardware and dependencies is available in Table 4.1.

System Configurations

	System 1 (CPU)	**System 2 (GPU)**
CPU	Apple M3 Max (ARMv8.6-A) (16 Cores, 64 GiB RAM)	AMD EPYC 7763 (AMD Zen 3) (32 Cores, 250 GiB RAM)
GPU		NVIDIA A100-SXM-80GB (NVIDIA Ampere)
Deps	LLVM 15.0.7 openBLAS 0.3.21	CUDA 11.4 openBLAS 0.3.21

Table 4.1: The hardware and dependencies (deps) of the benchmarking platforms.

For the CPU evaluation, we will target an Apple M3 Max CPU with 16 cores. On the GPU side, we will use an NVIDIA A100 as the target device. The search phase is a CPU-bound problem regardless of the target device since the BO library performs most of its computations using NumPy and SciPy. Therefore, the choice of BLAS (Basic Linear Algebra Subprograms) library, which we link against, is important for the search duration. We conducted small-scale tests and found OpenBLAS [34] to outperform competing libraries for our workloads and hardware.

Together, the two systems are representative of state-of-the-art ARM CPUs and server-class GPUs. Therefore, we believe that the hardware selection reflects the compiler's typical target hardware.

4.1.2 TVM Configuration

When configuring TVM and the benchmarks, several settings need to be chosen. The first is the number of tuning threads, which decides how many threads are used in the parallel segments of the tuning strategies. ES has been designed to be highly parallel and is exclusively written in C++. The BO strategy, in comparison, is primarily written in single-threaded Python. However, the BO library, which is responsible for most of the compute-intensive work, uses highly parallel BLAS implementations. As shown in Table 4.2, we selected the number of CPU cores as the number of tuning threads for the benchmarks. Both search strategies typically have a CPU utilization of around 1300% on the CPU system during their search phases.

TVM uses a target string to identify and select the hardware features for which it should generate optimized code. While most CPU target strings must be manually created,

TVM Configuration

System	Tuning Threads	Target String
(CPU)	16	`"llvm -num-cores 16 -mcpu=apple-latest -mtriple=arm64-apple-macos"`
(GPU)	32	`"nvidia/nvidia-a100"`

Table 4.2: The TVM tuning configuration.

popular GPU targets can utilize predefined settings by specifying the appropriate tag, as shown in Table 4.2. TVM offers two graph-level IRs. We have decided to use Relay for the evaluation and load the models' IRs via the Relay testing package and converted from TorchScript.

Unless otherwise stated, all benchmarks are based on a maximum trial limit of 6000 hardware measurements. We have selected this number as it is long enough to make performance gains through random selections highly unlikely, allowing the search strategies to show their effectiveness. Producing high-quality benchmarks has been a significant challenge; therefore, we included additional documentation in Appendix A.

4.1.3 Search Strategy Configuration

As stated before, we evaluate the BO search strategy by comparing it against MetaSchedule's state-of-the-art ES implementation, which we run with the default settings.

For BO, we determined the non-user-facing parameters via a small hand-tuned grid search to find a selection of parameters that perform well on CPU and GPU across different models. As shown in Section 3.8, we also offer a variety of user-facing parameters, which we also determined with a small grid search. Unless otherwise stated, all BO benchmarks used UCB with $\kappa = 0.1$ as their AF and a memory limit of 250 observations. The κ value is exploration-friendly as the GP will have significant uncertainty with only 250 observations. We also tested POI and EI but did not observe significant performance differences on CPU or GPU.

The compiled models are benchmarked by measuring the end-to-end inference time of a model 10x for 1000 iterations and calculating the averages. For each model, we conduct baseline measurements using MetaSchedule's ES strategy. The subsequent BO benchmark results are normalized against ES to show the comparative performance gain or loss. Each result presented is an average of three compilation runs since the probabilistic nature of the search can lead to slight performance differences.

In the subsequent sections, we will evaluate the four configurations introduced in Table 3.1 with these base settings. Together, they cover the most promising configurations and will help us gain insights into which attributes are essential for an efficient search.

4.2 Selection of Deep Learning Models

To test the search strategy's effectiveness, we compile three different models for each hardware target. Please note that we use a batch size of 1 for all models and a sequence length of 128 tokens for the transformer models since these are commonly used settings when benchmarking inference times of deep learning models, making our results comparable.

For the CPU benchmarks, we have selected MobileNet [18], BERT [11], and ResNet-50 [16] since we wanted a mixture of model types in the evaluation to show that the strategy works on different operators and model architectures. The vision model MobileNet is meant for deployment on mobile and edge devices, whereas ResNet-50 has higher hardware requirements. As current ML workloads increasingly focus on transformer models, we decided to include BERT in the evaluation.

Regarding the GPU benchmarks, we wanted to take the same selection of models. However, we had to switch ResNet-50 for GPT-2 [37] since the TVM version we used for the search strategy has a bug in one of the operators, preventing it from generating CUDA code for our target system. However, GPT-2 is a good replacement since most inference workloads on server GPUs will likely be transformer models. Therefore, using GPT-2 may be more representative of the current landscape.

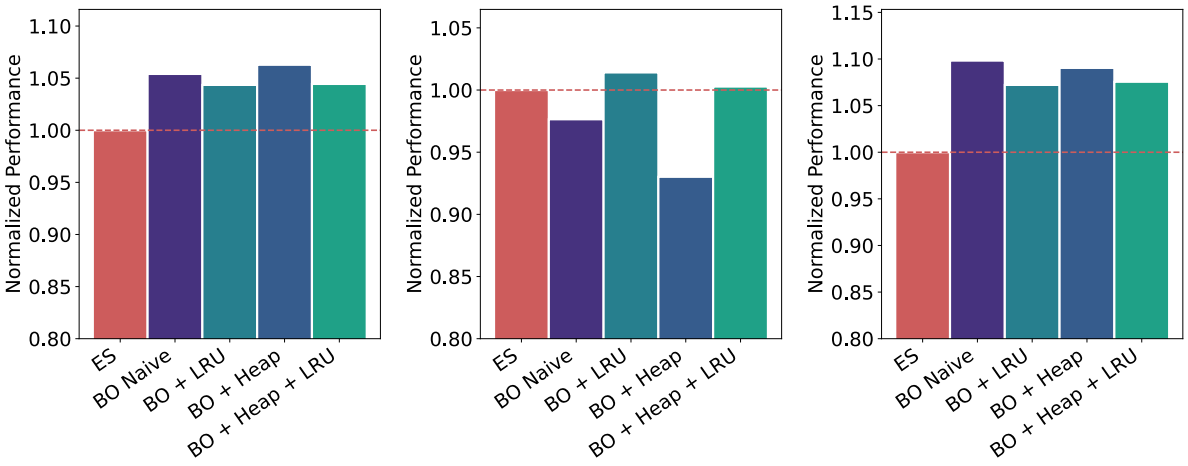
4.3 Overview Results

In the following sections, we will give an overview of the benchmarking results and evaluate possible causes of observed differences between ES and BO. As we consider the target hardware, it is important to note that GPUs are the most common target for large-scale model training; however, as TVM compiles models for inference, CPUs also play a significant role. This importance is further underscored by recent trends focusing on bringing ML models to mobile and edge devices. Additionally, recent research has shown that TVM is one of the best compilers for CPU targets [3]. We will start the evaluation by analyzing the CPU results before coming to the GPU evaluation.

4.3.1 CPU Results

To begin the evaluation of the CPU benchmarks, we will review Figure 4.1, which visualizes the normalized end-to-end latency gain or loss of the BO search strategy compared to the performance of MetaSchedule's ES strategy.

The first model we will review is MobileNet. As visible in Figure 4.1a, BO outperforms ES in all four configurations. In the best case, BO outperformed ES by 6.2% using the BO + Heap configuration, closely followed by the Naive approach. Both strategies using the LRU configuration to enforce the memory limit performed slightly worse. We will discuss the possible explanations for this later.



(a) MobileNet (b) BERT (c) ResNet-50

Figure 4.1: CPU benchmarks comparing ES against the BO configurations across models.

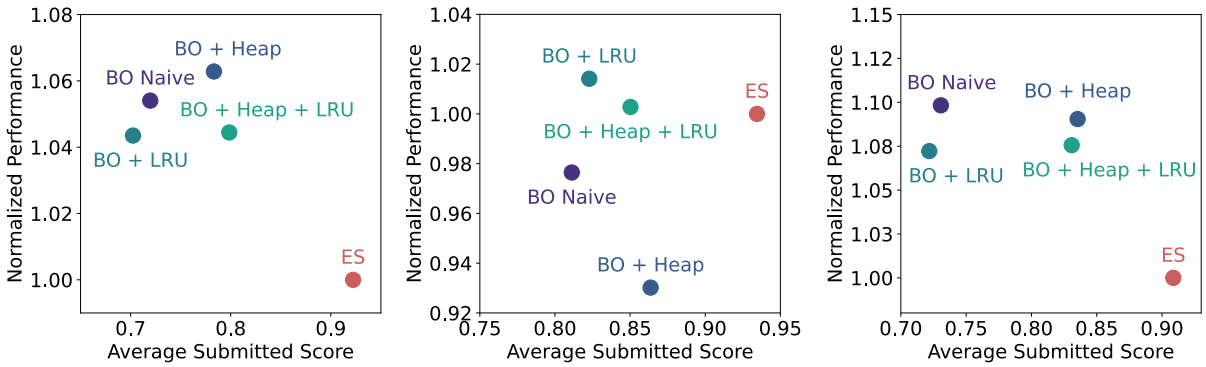
BO did not outperform ES for BERT but matched the performance; compare Figure 4.1b. This result is likely because more than 90% of all trials are being used on three matrix multiplication layers, which comprise most of the performance-critical tasks in the model. These layers have most likely converged close to an optimum after 6000 trials. This result indicates that BO converges to similarly performing programs as ES.

The BO search strategy caused the most significant performance increase over ES when compiling ResNet-50; compare Figure 4.1c. The model compiled with the Naive configuration was more than 10% faster than the one compiled with ES. The Heap configuration performed slightly worse than the Naive configuration. Overall, we saw similar behavior between ResNet-50 and MobileNet, where the configurations without the LRU setting performed best.

Score Analysis

To better understand how the search strategies interact with the cost model and the space itself, we will analyze the scores of the schedules we submit for measurement in more detail. Figure 4.2 visualizes the performance of the different BO strategies normalized against the performance of ES on the y-axis. The x-axis represents the average score submitted. Under the assumption of an ideal cost model, which perfectly predicts a program's performance, we would expect a direct correlation between higher scores and superior performance. However, the scatter plots in the three subfigures indicate that this is not the case. Therefore, we will examine possible reasons further and explain the observed behavior.

Before having a closer look at Figure 4.2, it is important to contextualize the cost model scoring further. As mentioned before, the cost model returns the predicted normalized throughput of a schedule. Consequently, the best-discovered program has a predicted score of 1.0. Almost all other programs will score below 1.0 since the cost model often



(a) MobileNet (b) BERT (c) ResNet-50

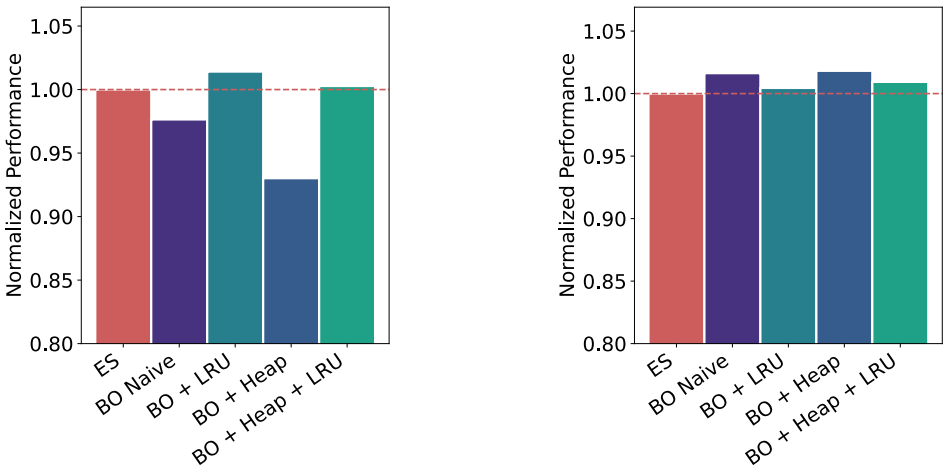
Figure 4.2: The average score submitted versus the normalized performance achieved.

fails to identify programs better than the current best. The objective is, nonetheless, to identify schedules with high scores. They are typically found in programs closely related to the current best, as their throughput is similar. As a result, we may measure similar programs, which maximize the submitted score average but may only lead to the discovery of marginally better programs within a local maximum. Overall, constantly averaging scores close to 1.0 can indicate a high measure of exploitation. This also has the side effect that the cost model can accurately score schedules within the exploited area of the search space but has significant uncertainty in other areas.

Coming to MobileNet, while the BO scores are significantly lower than ES, the performance exceeds that of ES; compare Figure 4.2a. Regarding the average score submitted, we can see a distinct difference between the BO configurations with and without the Heap setting. The configurations with the Heap selection returned scores, which were, on average, 0.08 points higher. The second observation is that both LRU configurations performed slightly worse than their counterparts, while the submitted scores are similar. These two trends are also visible for ResNet-50; compare Figure 4.2c.

From the similar scores of the configurations with LRU turned on and those without, we can see that restarting the optimizer does not significantly affect the average submitted score. However, the LRU setting negatively affects the performance of MobileNet and ResNet-50. This is likely due to spending more time in unfavorable areas of the search space, i.e., the LRU setting exploits local maxima. Since the optimizer never restarts, it explores the search space less, which seems to be especially important for MobileNet and ResNet-50.

The Root Mean Square Error (RMSE) of the cost model predictions measures the average deviation of the predictions from the actual values. The RMSE of the LRU configurations supports the conclusion that the setting's worse performance is caused by less exploration. The configurations with the LRU setting have an RMSE of around 0.175, while those without the setting have an RMSE of roughly 0.20. The difference suggests that the cost model can better score the programs discovered with the LRU strategy. This improved accuracy



(a) $\kappa = 0.1$. (b) $\kappa = 0.01$.

Figure 4.3: BERT performance when compiled with different exploitation factors.

is likely due to the increased exploitation of a specific subsection of the search space, leading to more training data for one area. Due to the higher cost model accuracy but similar cost model scores, further exploitation of the subregion becomes less advantageous than exploration of other regions in the search space. It is important to note that an RMSE of 0.175 is still relatively high, indicating that the cost model's accuracy remains limited for the given number of trials.

Having reviewed MobileNet and ResNet-50, we can now examine BERT, which differs from the other two models, as we can only match the performance of ES. Additionally, the two LRU configurations achieve better performance than the ones without. This may indicate that BERT requires more exploitation than the previous two models or is close to convergence. In this case, a stronger focus on exploitation, which can be achieved by turning on LRU, can be beneficial since the general locations of the optima have likely been discovered, and slight gains may only be possible by exploiting those areas. To further confirm this, we recompiled BERT but adjusted the exploration-exploitation trade-off factor κ. Previously, we compiled with a κ of 0.1; now, we compiled it with 0.01. As a result, we are more likely to probe points next to the best previous points, increasing the exploitation of the search.

Figure 4.3 compares the performance between the two settings for κ. The increased focus on exploitation with $\kappa = 0.01$ allows all configurations to match the performance of ES. The observed behavior shows how important the selection of hyperparameters is for the search quality. However, it is challenging to reason which parameter combination will perform best beforehand.

Trial Requirements

After comparing the performance of the two search strategies with 6000 trials, we want to investigate how many more trials are required by ES to reach BO's latency. We chose

the BO configuration with the lowest average latency for each model. To find the number of trials where ES matches BO's performance, we performed a binary search over the number of trials. We define a match in performance as the lowest number of trials where the ES average of three separate compilation runs is equal to or better than the average of the BO runs. Additional methodology is available in Appendix A.1.

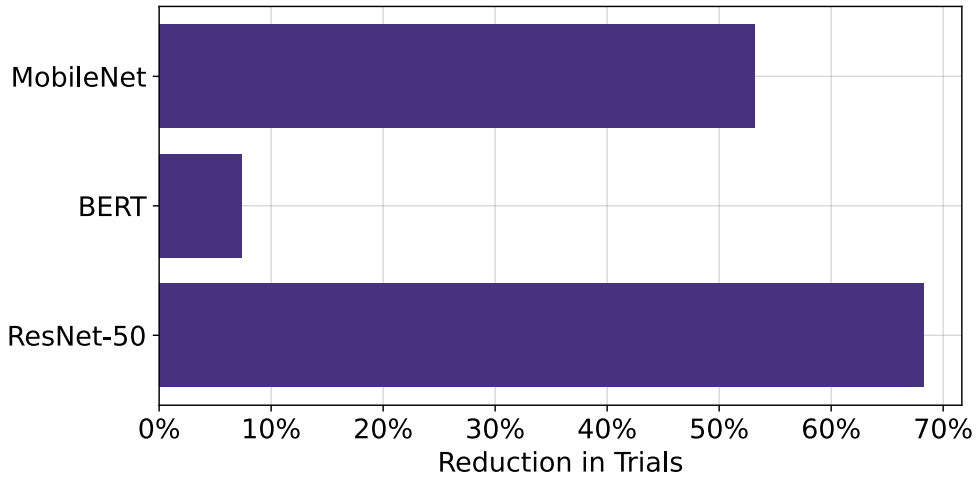


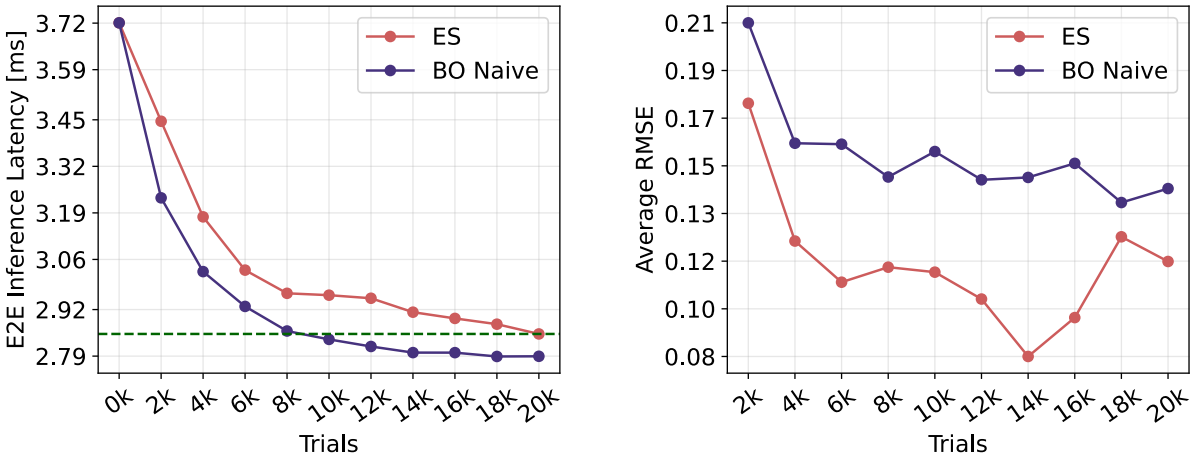
Figure 4.4: Reduction in required trials of the best BO configuration compared to ES.

Figure 4.4 shows that BO reduces the number of required trials for MobileNet and ResNet-50 by more than 50% and 60%, respectively. This is significant, as their performance at 6000 trials is only 6% and 10% apart. However, a gap of this size after 6000 trials is considerable because most of the easy-to-make performance gains have been discovered at that point. For example, MobileNet, compiled with BO, improved by 0.82 ms in the first 6000 trials while only improving by 0.13 ms in the subsequent 14000 trials.

For BERT, the difference in required trials is less significant. However, this makes sense as BO and ES follow an equally exploitation-heavy strategy, making their convergence behavior similar. Consequently, we can determine that BO requires significantly fewer trials when the workloads in the model are suitable for a more exploration-heavy search.

Convergence Analysis

The results we previously examined were recorded with 6000 trials, and we saw the benefit of exploration. Therefore, we wanted to examine the performance of a longer compilation run to show that the observed performance is not limited to the early stages of the optimization. To do so, we compile MobileNet with 20000 trials, as it is long enough to indicate convergence characteristics and is viable with the limited computing resources available. Figure 4.5 shows the mean end-to-end inference latency on the y-axis and the number of trials used on the x-axis. The graphs are based on a single compile run, representative of the performance after 20000 trials, which was benchmarked multiple times throughout the search to show the latency's evolution.



(a) The end-to-end (E2E) inference latency throughout the optimization.

(b) The average RMSE of the cost model predictions.

Figure 4.5: Evolution of latency and cost model RMSE when compiling with 20,000 trials.

From Figure 4.5a, we can tell that TVM's untuned base implementation (see 0 trials) achieves a latency of 3.72 ms, which could be lowered to around 2.79 ms using the Naive BO strategy and 2.85 ms using ES, after 20,000 trials. This shows that the BO strategy is ahead of ES, even for extended optimization runs. The final ES latency is indicated with a horizontal green dashed line in the figure and intersects with the BO strategy at around 9000 trials. From this, we can tell that the BO strategy required about 11,0000 or 55% fewer trials to reach a similar performance to ES.

One characteristic, which may give insights into the root cause, is visualized in Figure 4.5b and shows the average RMSE of the XGB cost model predictions. From the figure, it becomes clear that ES consistently achieves a lower RMSE, likely due to more exploitation, enabling it to make more accurate predictions within the area of the search space it is exploiting. BO's error, on the other hand, is significantly higher due to the broader exploration and wider variety of the programs selected, exploiting the uncertainty in the cost model. While 20,000 trials may not be enough to show the overall trends in cost model accuracy, it is noticeable that BO's cost model error decreases slowly but linearly, while the ES RMSE seems to be more erratic. This is likely due to the cost model overfitting in one area and spiking after discovering a new performant area. Therefore, we believe that BO's stronger emphasis on exploration leads to a more balanced cost model over the search space.

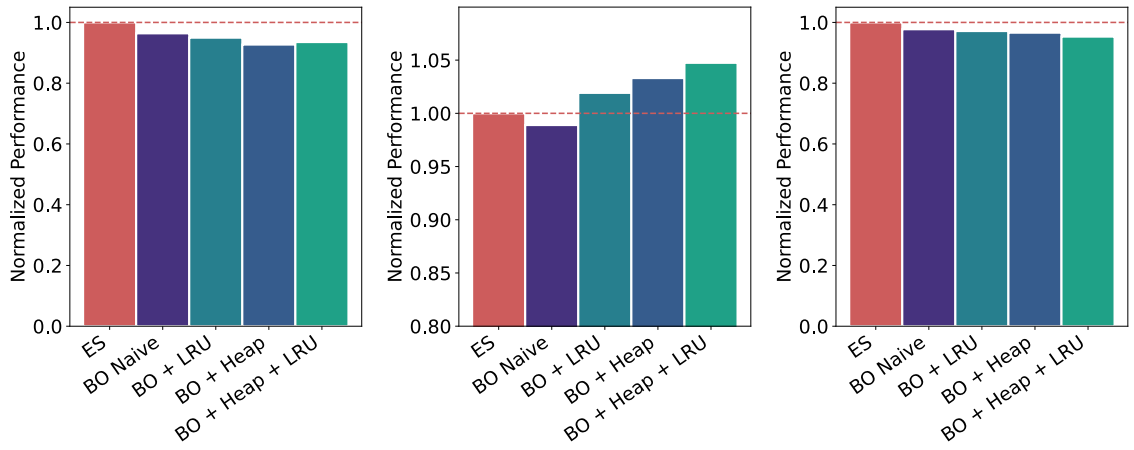
Summary CPU Results

In this section, we have seen that our BO-based search strategy can reliably identify performant programs within the search space, achieving performance exceeding that of the ES strategy by around 6% to 10% on MobileNet and ResNet-50, respectively. For BERT, we were able to match ES with some configurations; only with an adjustment in the hyperparameters were we able to match ES across all configurations. The analysis also

showed that while a 6% to 10% decrease in mean end-to-end inference latency appears low, ES needs significantly more trials, with BO reaching ES performance levels with up to 68% fewer trials. Overall, we saw that BO's exploration is vital to achieving strong performance when optimizing code for CPU targets since ES's more exploitation-focused approach is more likely to get stuck in local maxima.

4.3.2 GPU Results

Having examined the CPU results, we can now evaluate the performance achieved when targeting a GPU. Figure 4.6 shows the performance of the three selected models when compiled with the same settings and configurations as in the CPU benchmarks. It becomes clear that the results here are not as favorable for the BO strategy as before. On MobileNet, BO can match the performance of ES; on BERT, we can significantly outperform ES; and on GPT-2, BO falls behind ES.



(a) MobileNet (b) BERT (c) GPT-2

Figure 4.6: GPU benchmarks comparing ES against the BO configurations across models.

We will start by reviewing BERT's results more closely. As visible in Figure 4.6b, the Naive configuration performs around 1.1% worse than ES, whereas the other configurations can outperform ES. This result becomes reasonable when we consider that vast portions of the search space are invalid. By resetting the optimizer's memory, information on their location is lost and has to be rediscovered. In comparison, the LRU configurations will always have a mix of valid and invalid points in memory. Thus reducing the overall rate of invalid points probed and allowing more evaluations to be spent on finding performant programs.

The BO + Heap + LRU configuration performs better than the Naive configuration since the average submitted score is significantly higher, as shown in Table 4.3. The distance of around 0.2 between the configurations' scores is significant and translates to the achieved performance. Even with possible inaccuracies in the cost model scoring, the programs discovered by the Naive configuration are unlikely to be better than the current best.

BERT Task Scores & Latency

Task Name	BO Naive		BO + Heap + LRU		EvoSearch	
	Avg. Score	Latency	Avg. Score	Latency	Avg. Score	Latency
`fused_nn_batch_matmul_2`	0.63	26 µs	0.84	**22** µs	**0.97**	27 µs
`fused_nn_batch_matmul_3`	0.68	66 µs	0.88	**61** µs	**0.97**	62 µs
`fused_nn_batch_matmul_4`	0.57	94 µs	0.84	**78** µs	**0.97**	94 µs

Table 4.3: Average task tuning score and the resulting latency. The best scores and latencies are highlighted in bold.

When comparing the BO + Heap + LRU configuration to ES, we observe a significant difference in the scores and latencies; see Table 4.3. Although BO's scores are lower than those of ES, the latency is better as we discover performant regions more quickly due to the higher exploration rate. The overall scores of BO and ES are similar to the CPU scores, with ES averaging 0.93 and BO + Heap + LRU averaging 0.82. It is important to note that task latency is unweighted, representing the latency of a single pass through the layer; however, `fused_nn_batch_matmul_2`, for example, is used 48 times in the model.

As illustrated in Figure 4.6, MobileNet and GPT-2 perform slightly worse with BO than ES. This discrepancy is again linked to the scores. Most configurations for these models have scores below 0.70. Low scores indicate that the discovered programs are far from the performant regions in the search space, making it unlikely they will outperform the current best, even with potential inaccuracies in the cost model. However, while the scores were low across all configurations, some of the lower scores outperformed the BO + Heap + LRU configuration, which, for both models, achieved the highest scores. As can be seen, MobileNet and GPT-2 both displayed almost a reversed pattern to BERT. Nonetheless, we remain confident that BO's slightly worse performance for the two models is primarily due to BO's average score being significantly below ES's average submitted score.

Overall, we observed that for all models except BERT, we could not consistently find high-scoring programs, making it challenging to match or outperform ES. The reasons behind BO's low scores during GPU scheduling will be explored in the next section.

4.4 Limitations

Having seen BO's strong performance when scheduling CPU code and its limited efficiency when scheduling for GPUs, we want to use this section to explore the strategy's limitations in more detail. We will begin by analyzing the factors contributing to BO's effectiveness for CPU scheduling and how they change when generating code for a GPU. This includes a detailed analysis of the characteristics of the objective function, represented by the cost model's throughput prediction, and the overall qualities of the search space. Following this, we will discuss the limitations stemming from the variety of configurations and the effects of BO's computational complexity on the search duration.

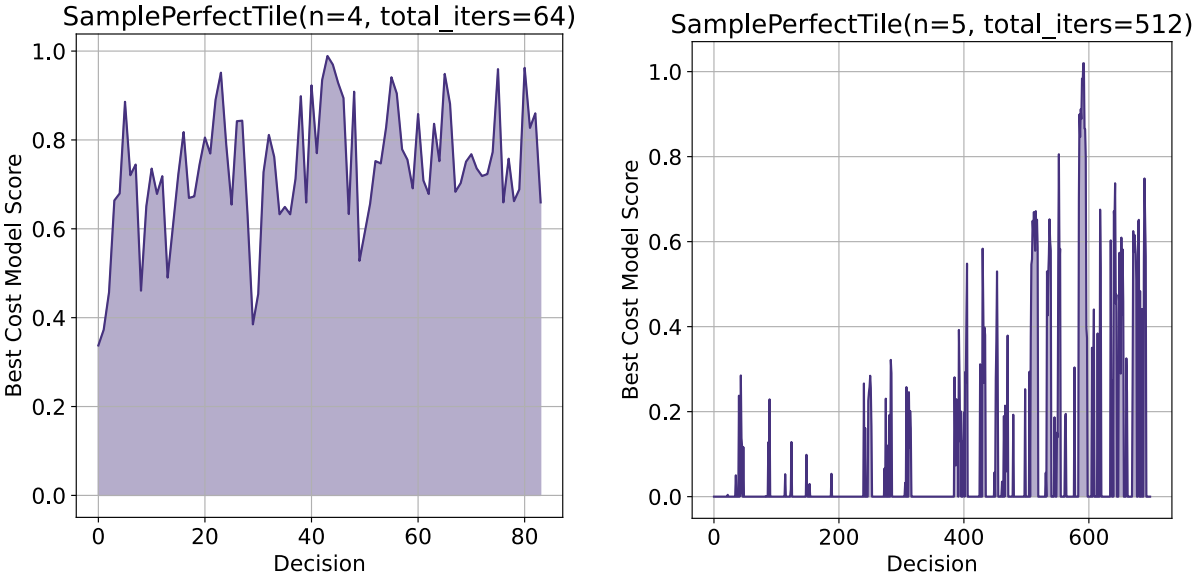
4.4.1 CPU and GPU Search Space Characteristics

In this section, we will discuss the factors that limit BO's effectiveness when searching for performant GPU programs. To better understand how CPU and GPU scheduling differs, we need to examine the characteristics of their respective search spaces in more detail. In our evaluation, we observed two primary differences between the CPU and GPU space:

1. The GPU search space is significantly larger than the CPU space. For example, MobileNet's search space has an average of 3.7×10^7 possible parameter combinations per task on CPU but roughly 4.1×10^9 on GPU.
2. A significant portion of the possible parameter combinations lead to invalid programs. For example, when uniformly sampling parameters, the percentage of invalid GPU schedules for MobileNet is around 95%. On CPU, that number is close to 0%.

Search Space Example

Figure 4.7 illustrates the effects of these two changes on the search space. The figures show the enumerated possible parameters of a `SamplePerfectTile` instruction on the x-axis and the best-achieved cost model score, after 7500 parameter combination evaluations, on the y-axis. The selected transformation has the highest parameter count in MobileNet's biggest latency contributing layer, `fused_nn_conv2d_add_nn_relu_14`. Please note that the visualized sample instruction represents only one of the 7 dimensions in the GPU or 8 dimensions in the CPU space.



(a) CPU (b) GPU

Figure 4.7: Visualization of the best scores in one transformation's search space.

From Figure 4.7a, we can tell that every possible transformation parameter in the CPU space can create a valid program. It also becomes clear that no decision on this instruction will guarantee a performance close to 0, regardless of the other dimensions' parameters.

When we compare this to Figure 4.7b, which shows the GPU space of the transformation, we can notice the two discussed differences. First, the space is substantially larger; the CPU transformation has 84 possible decisions, while the GPU transformation has 700. Secondly, we can see that significant portions of the search space do not lead to valid programs, or no valid programs were discovered for many parameters after 7500 evaluations. Due to these invalid programs, the graphed cost model score for the transformation displays an oscillatory pattern. The fluctuations of the CPU's cost model graph are, in comparison, less significant. Further, visualizations and analysis of other models can be found in Appendix B.

Challenges

If we consider the two graphs a rough representation of one objective function dimension, we can begin to understand the factors that degrade BO's effectiveness in the GPU search space. The first factor is that with the settings used in the benchmarks, the GP can only use 250 observations to build a probabilistic model of the 7-8 dimensional search space. Therefore, the lower the cardinality of a transformation's search space, the more accurate the GP's probabilistic model will be.

Besides the search space size, the characteristics of the objective function also play a role in the search quality. The oscillatory characteristics of the GPU target space are hard to optimize for BO as the GP kernel assumes a continuous and smooth objective function. The characteristics here, however, resemble a discontinuous, non-differentiable function, which is very challenging for the GP to model. This is further complicated by the narrowness of the performant regions, as a high-performing point in the space can be followed by a point that fails validation. Considering that the GP's model is smooth, it is likely that the performant regions in the GP are initially modeled wider than they are. As a result, even with a stronger focus on exploitation, the likelihood of picking a parameter outside a performant region remains high. The likelihood of picking an invalid program can, nonetheless, be lowered significantly; while it was around 95% when uniformly sampling from MobileNet's space, it drops to 43% with BO.

Therefore, the challenge is to exploit the small, performant regions inside the search space with a surrogate model that struggles to accurately represent the objective function. We aimed to make the search more efficient by increasing the exploitation of the search by setting UCB's trade-off factor to 0.01 and 0.0; we also evaluated the use of an SDR strategy and increased the memory of remembered points to 400 and 500 points. However, none of these settings or their combination increased the performance significantly.

Summary

The more substantial exploration factor, which helped BO outperform ES for CPU scheduling, is counterproductive for GPU scheduling. The locality of performant and

valid programs is significant, making exploitation incredibly important. The combination of limited optimizer memory in a large space where the objective function oscillates makes it challenging for BO to suggest performant points.

The points BO suggests for evaluation are typically near the parameters of the best programs found, which we registered via the input schedules. However, while they are in proximity, every dimension of the parameter combination has been slightly changed. This indicates the GP's wider modeling of performant regions, which, combined with the high degree of locality, often leads to non-optimal results. In comparison, ES picks a candidate schedule and mutates one parameter while keeping all other parameters fixed. This is done with hundreds of candidates for multiple generations. As a result, ES can more effectively exploit the narrow, performant regions of the GPU's objective function.

When we began work on this thesis, the characteristics of the objective function were unclear. While we did expect the function to have some fluctuations due to the discrete search space and the probability that small changes can significantly influence a program's throughput, we did not expect the GPU's objective function to be this challenging. Initially, we presumed that the GPU space would mirror the CPU space, with the only difference being its size and a low percentage of invalid programs. Our analysis shows that these assumptions were inaccurate and that the GPU objective function is significantly more complicated. This thesis identifies the concrete challenges a search strategy must address, making it possible for future research to be more targeted.

4.4.2 Parameters

Throughout the evaluation chapter, the settings and configurations of the search strategy played a significant role. For example, when BO did not match the performance of ES with all configurations when compiling with $\kappa = 0.1$, we lowered it to 0.01, which helped BO to match ES across all configurations. However, ideally, there would be a parameter combination that works optimally across models and hardware targets. As we did not identify such a strategy, we recommend using a UCB with $\kappa = 0.1$, a memory limit of 250, with the BO + Heap configuration since it is substantially faster than the configurations with the LRU setting. For future work, it would be interesting to implement a strategy that adjusts the trade-off factor depending on the workspace characteristics and increases exploitation towards the end of the search.

4.4.3 Search Duration

BO is highly efficient in the number of trials required when targeting a CPU, as we have seen in Figure 4.4. However, the high time complexity of BO is a potential limitation that we want to evaluate by considering the tuning duration. Table 4.4 shows the search duration of each model's best BO configuration required for 6000 trials. It also shows the

percentage of time spent in the optimization phases of the overall search time. For ES, we include the duration for 6000 trials and the duration until it matches BO's performance.

The durations it took to run 6000 trials make it clear that BO requires significantly more time; e.g., BERT took 190 min to compile with BO and 80 min with ES. Of those 190 min, 58% are spent in the optimizer, representing the three search phases. The remaining 42% are distributed between building the selected schedules for hardware measurements, the actual measurements, retraining the cost model, and other smaller tasks. As we mentioned, BO is sample-efficient. However, compared to ES, we have to perform significantly more work to find each sample.

CPU Search Duration

Model	BO Strategy			ES Strategy		Search
	Duration	Search Phases	Configuration	Duration	Duration Same Perf.	Speedup
MobileNet	**228 min**	55%	BO + Heap	107 min	230 min	**1.01x**
BERT	190 min	58%	BO + Heap[1]	80 min	**85 min**	**0.45x**
ResNet-50	**362 min**	58%	BO Naive	126 min	412 min	**1.14x**

Table 4.4: Search durations of the best BO configurations compared to ES for 6000 trials and the time ES takes to match BO's performance on the CPU system. The durations in bold represent the fastest times to reach the latency of BO after 6000 trials.

When comparing the time until ES reaches BO's average latency, we see that their duration is almost the same for MobileNet, and BO is around 50 min faster for ResNet-50. However, BO is significantly slower on BERT as ES only requires 500 additional trials to match the performance.

Overall, BO's duration can match or undercut the duration of ES for the models it performs well on. It takes significantly longer for the ones it can only match the performance on. Due to this limitation, we have considered the following possible optimizations for future work that could help reduce the search durations of the BO strategy:

1. Rewrite the *hot path* of the search strategy in C++. Dozens of separate calls to the C++ bindings are currently made in the hot path. Each call has an inherent latency since the Python datatypes have to be converted to their C++ counterparts. Additionally, C++ generally has better performance than Python.

2. The most compute-intensive part of BO is the inversion of the GP's covariance matrix. The BO library we use only offers CPU support, so this operation is expensive. Therefore, implementing a BO library with GPU acceleration may lead to considerable performance improvements.

3. Another possible optimization would be the use of batch BO, as it allows simultaneous suggestion of parameter combinations [15]. This may lead to a better performance in combination with the previous two optimizations.

[1] with $\kappa = 0.01$

Chapter 5

Conclusion

Looking back at the AI landscape since the beginning of TVM's development, we can see that it has changed significantly. Over the last few years, the number of model architectures in use has reduced as development has shifted to transformer models and their applications. Besides a decrease in model variety, the variety of server-class GPUs has also reduced as NVIDIA has solidified its position in the market. Consequently, NVIDIA's libraries are highly performant and well-maintained. Reducing the need for scheduling systems for server-class GPUs since performance rivaling that of TVM can be achieved with code generation phases, which take minutes, not hours. This is one of the reasons for TVM's Bring Your Own Codegen (BYOC) feature, enabling, e.g., the use of cuDNN for code generation while keeping TVM's graph-level optimizations.

Looking ahead, we are optimistic about the potential of TVM scheduling for compiling models for deployment on edge devices, mobile devices, and specialized accelerators. In these contexts, the availability of operator libraries is more limited, hardware diversity is greater, and there is a strong push to move models to the edge to save data center costs and address privacy concerns. For example, recent studies by researchers at Apple have focused on optimizing models for deployment to their mobile devices [2, 49].

Under this pretense, we will summarise the key results and findings of this thesis in the subsequent section. This will be followed by a section on future work, where we discuss possible further research directions and refinements.

5.1 Summary

In this thesis, we have implemented BO as a novel search strategy for TVM and investigated its effectiveness across different models and hardware. We started by reviewing the principles of creating equivalent programs through parameterized transformations and BO as a sequential design strategy. Followed by a detailed walkthrough of the implementation of the BO search strategy before conducting an extensive performance evaluation.

On CPU, BO demonstrated competitive or superior performance across all evaluated models. For example, BO achieved a roughly 10% lower end-to-end inference latency on ResNet-50 with the same number of trials as ES. This represents a reduction of 68% compared to the number of trials ES requires to achieve similar performance. However, BO's performance on GPU was less consistent, falling short of ES when compiling MobileNet and GPT-2, only surpassing ES on BERT. The discrepancy in behavior between the two hardware targets can be attributed to significant differences in the search space characteristics, making the objective function challenging to model on the GPU system.

At the outset of the evaluation in Chapter 4, we set key objectives with which we guided our evaluation. We will revisit them here and see if we were able to achieve the objectives:

1. We have shown that BO can be used to discover high-performance tensor programs.
   (a) We have shown that BO can be more sample-efficient than ES on CPU targets.
   (b) We have shown that BO can be more time-efficient for some models.

2. We have shown that BO's effectiveness is affected by the different characteristics of the objective function depending on models and hardware targets.

While BO's effectiveness is limited compared to ES when compiling for GPU targets, it excels for CPU compilation, which is of growing interest with the push to deploy ML applications to mobile and edge devices. This research concludes that BO presents a powerful alternative to ES as a search strategy for optimizing deep learning models for CPU targets. Here, BO significantly reduced the number of trials required and had a superior time efficiency for some models.

5.2 Future Work

Some of the future work arising from the search strategy's limitations has been discussed in Section 4.4. Nonetheless, we identify the following three avenues as particularly promising for further research that can build upon the foundations and ideas established in this thesis:

Scalability is a significant limitation of BO. The time complexity of updating the underlying GP must be considered when applying it to an optimization task. In our problem setting, we optimize against a cost model to identify plausibly good candidates. However, finding these plausible programs in the space becomes challenging as search spaces and their complexity scale. Further research may focus on improving the current implementation's performance and experimenting with alternatives to the GP kernel, such as Random Forests, which have been shown to handle high-dimensional and discrete spaces better while having a lower time complexity [20]. Reinforcement Learning may also be an effective approach capable of handling the complicated domain effectively [47].

Cost Model Accuracy is another challenge that can be considered for future work. As we have seen, the cost model can have a significant bias towards programs similar to the ones it has seen before. This behavior can reduce the meaningfulness of the scoring and lead optimizers to over-exploitation of local maxima. This has allowed the BO strategy, while producing fewer high-scoring programs than ES, to discover other promising regions in the search space. A recent development introduced a differentiable cost model enabling the use of gradient descent as a search strategy [54]. Additionally, researching cost models may help improve overall convergence characteristics by improving their accuracy.

Multi-Objective BO may be a promising further step. The current cost model and search strategies only optimize for throughput, i.e., for quick inference times. However, with the trend toward edge computing, it is also feasible to consider other optimization goals, such as energy consumption. Consequently, a multi-objective BO strategy could be a relevant contribution as it would allow balancing multiple optimization goals. However, it would require significant changes throughout the compiler's scheduling system.

Chapter 6

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. Llm in a flash: Efficient large language model inference with limited memory. *arXiv preprint arXiv:2312.11514* (2023).

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[4] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*. PMLR, 115–123.

[5] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM:

An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

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

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[10] Samuel Daulton, Xingchen Wan, David Eriksson, Maximilian Balandat, Michael A Osborne, and Eytan Bakshy. 2022. Bayesian optimization over discrete and mixed spaces via probabilistic reparameterization. *Advances in Neural Information Processing Systems* 35 (2022), 12760–12774.

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

[12] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 804–817.

[13] Peter I Frazier. 2018. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018).

[14] Eduardo C Garrido-Merchán and Daniel Hernández-Lobato. 2020. Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes. *Neurocomputing* 380 (2020), 20–35.

[15] Javier González, Zhenwen Dai, Philipp Hennig, and Neil Lawrence. 2016. Batch Bayesian optimization via local penalization. In *Artificial intelligence and statistics*. PMLR, 648–657.

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

[17] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2023. Baco: A fast and portable Bayesian compiler optimization framework. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 19–42.

[18] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[19] Haochen Hua, Yutong Li, Tonghe Wang, Nanqing Dong, Wei Li, and Junwei Cao. 2023. Edge computing with artificial intelligence: A machine learning perspective. *Comput. Surveys* 55, 9 (2023), 1–35.

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

[21] Intel. 2024. Intel oneAPI Math Kernel Library (oneMKL). `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`

[22] Harold J Kushner. 1964. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. (1964).

[23] Ruihang Lai, Junru Shao, Siyuan Feng, Steven S Lyubomirsky, Bohan Hou, Wuwei Lin, Zihao Ye, Hongyi Jin, Yuchen Jin, Jiawei Liu, et al. 2023. Relax: Composable Abstractions for End-to-End Dynamic Machine Learning. *arXiv preprint arXiv:2311.02103* (2023).

[24] Gongjin Lan, Jakub M Tomczak, Diederik M Roijers, and AE Eiben. 2022. Time efficiency in optimization with a bayesian-evolutionary algorithm. *Swarm and Evolutionary Computation* 69 (2022), 100970.

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

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

[27] Kai Yuan Andre Low, Eleonore Vissol-Gaudin, Yee Fun Lim, and Kedar Hippalgaonkar. 2023. Bayesian vs Evolutionary Optimisation in Exploring Pareto Fronts for Materials Discovery. *Authorea Preprints* (2023).

[28] Phuc Luong, Sunil Gupta, Dang Nguyen, Santu Rana, and Svetha Venkatesh. 2019. Bayesian optimization with discrete variables. In *AI 2019: Advances in Artificial Intelligence: 32nd Australasian Joint Conference, Adelaide, SA, Australia, December 2–5, 2019, Proceedings 32*. Springer, 473–484.

[29] Jonas Mockus. 1974. On Bayesian methods for seeking the extremum. In *Proceedings of the IFIP Technical Conference*. 400–404.

[30] Jonas Mockus. 1994. Application of Bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization* 4 (1994), 347–365.

[31] J Mockus, V Tiesis, and A Zilinskas. 1978. The application of Bayesian methods for seeking the extremum, vol. 2. *L Dixon and G Szego. Toward Global Optimization* 2 (1978).

[32] Fernando Nogueira. 2014–. Bayesian Optimization: Open source constrained global optimization tool for Python. `https://github.com/bayesian-optimization/BayesianOptimization`

[33] ONNX. 2019. Open Neural Network Exchange. `https://onnx.ai`

[34] OpenBLAS. 2024. OpenBLAS Library. `https://github.com/OpenMathLib/OpenBLAS`

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[36] PyTorch. 2023. TorchScript. `https://pytorch.org/docs/stable/jit.html`

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

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

[39] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*. 58–68.

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

[41] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[42] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas.

2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.

[43] 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. *Advances in Neural Information Processing Systems* 35 (2022), 35783–35796.

[44] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* 25 (2012).

[45] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. 2009. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995* (2009).

[46] Nielen Stander and Kenneth J Craig. 2002. On the robustness of a simple domain reduction scheme for simulation-based optimization. *Engineering Computations* 19, 4 (2002), 431–450.

[47] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[48] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. 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*. 10–19.

[49] Pavan Kumar Anasosalu Vasu, Hadi Pouransari, Fartash Faghri, Raviteja Vemulapalli, and Oncel Tuzel. 2023. MobileCLIP: Fast Image-Text Models through Multi-Modal Reinforced Training. *arXiv preprint arXiv:2311.17049* (2023).

[50] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. 2022. Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization. *Concurrency and Computation: Practice and Experience* 34, 20 (2022), e6683.

[51] Xingfu Wu, Praveen Paramasivam, and Valerie Taylor. 2023. Autotuning Apache TVM-based Scientific Applications Using Bayesian Optimization. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 29–35.

[52] Xiongfei Wu, Jinqiu Yang, Lei Ma, Yinxing Xue, and Jianjun Zhao. 2022. On the usage and development of deep learning compilers: an empirical study on TVM. *Empirical Software Engineering* 27, 7 (2022), 172.

[53] Steven R Young, Derek C Rose, Thomas P Karnowski, Seung-Hwan Lim, and

Robert M Patton. 2015. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the workshop on machine learning in high-performance computing environments*. 1–5.

[54] Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. 2024. Felix: Optimizing Tensor Programs with Gradient Descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 367–381.

[55] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879. `https://arxiv.org/abs/2006.06762`

Appendix A

Benchmarking

We have gone over the basics of the benchmarking setup in Section 4.1.2. However, we want to provide further insight into the methodology and experiments we conducted to ensure accurate measurements.

First, the entire search duration can be seen as one long benchmark since search periods are interleaved with measuring periods. For the measurement periods, we configured our setup to measure a potential program for at least 200 ms; for most layers, this is equivalent to a few hundred measurements. We also tried longer benchmark durations; however, this did not affect the search quality.

After each measurement, it would be beneficial to flush the cache to avoid subsequent measurements to have possible benefits from cache hits. However, TVM's cache flush system does not work on macOS, which can impact the search quality slightly. As the missing cache flushes impact BO and ES equally, the comparative results are not impacted by this.

Additionally, we tried to determine the effects of the Operating System scheduler on the measurement and benchmark quality. We experimented with setting the scheduling priority higher by decreasing the niceness of a process with the `nice(1)` utility. Through this, we determined that scheduling priority did not play a significant role in the search quality, which led us to run the benchmarks with the default priority.

It is important to note additional details for the M3 Mac setup. We were using macOS Sonoma 14.1, the machine was plugged into the power supply throughout the benchmark, the high-performance mode was turned on, and no other tasks were performed during the benchmark.

Our GPU system was running Rocky Linux 8 and was provided by the University of Cambridge's Research Computing Services. Each node in the cluster contains 2x AMD EPYC 7763 64-Core CPUs, 1000 GiB RAM, 4x NVIDIA A100-SXM-80GB GPUs, and a dual-rail Mellanox HDR200 InfiniBand interconnect. As we only required a single A100,

we requested a quarter node, giving us access to 32 cores with 250 GiB of RAM and one A100. These resources were exclusive to our job. However, as the node may have other processes running, jitter can be introduced due to motherboard bandwidth limitations. Since the project's resource requirements are significant, requesting an entire node was not feasible. Despite this, we believe that we have collected accurate measurements due to the consistency and repeatability of the results.

We used Conda environments to manage dependencies for the CPU and GPU systems. The respective environment files and CMake configurations are available upon request.

A.1 Matching BO and ES Performance

In Section 4.3.1, we evaluate how many trials ES requires to match the latency of BO at 6000 trials. Due to the high computational requirements, we used a binary search to limit the number of compilation runs required to find the number of trials. However, we also limited the search resolution to 1000 trials and 500 trials for BERT. As a result of the low sample count and significant standard deviation, it is hard to pinpoint the exact number of trials required. However, we narrowed it down to the approximate numbers displayed in Figures A.1 to A.3.

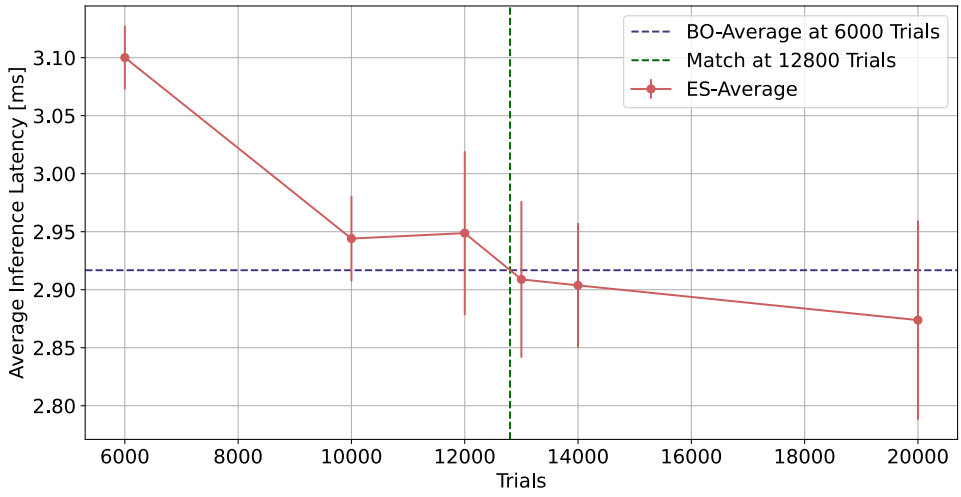


Figure A.1: MobileNet ES-BO Performance Match.

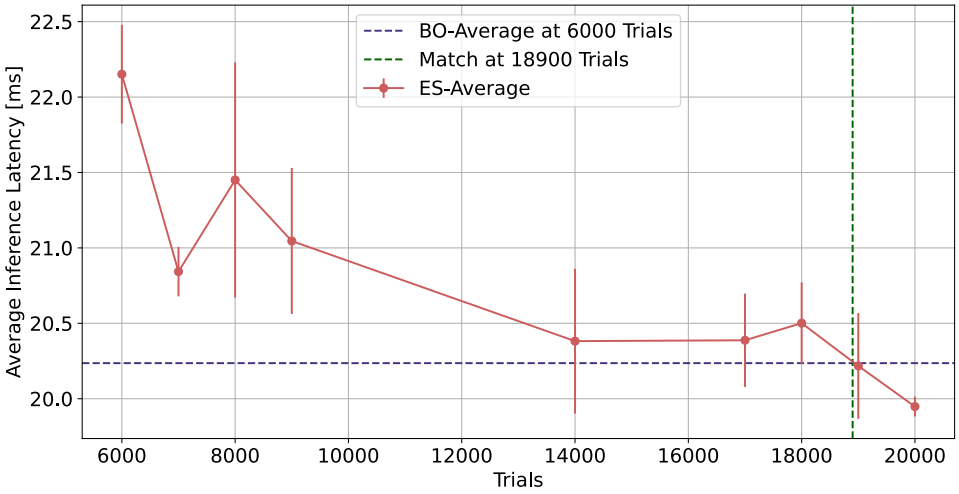


Figure A.2: ResNet-50 ES-BO Performance Match.

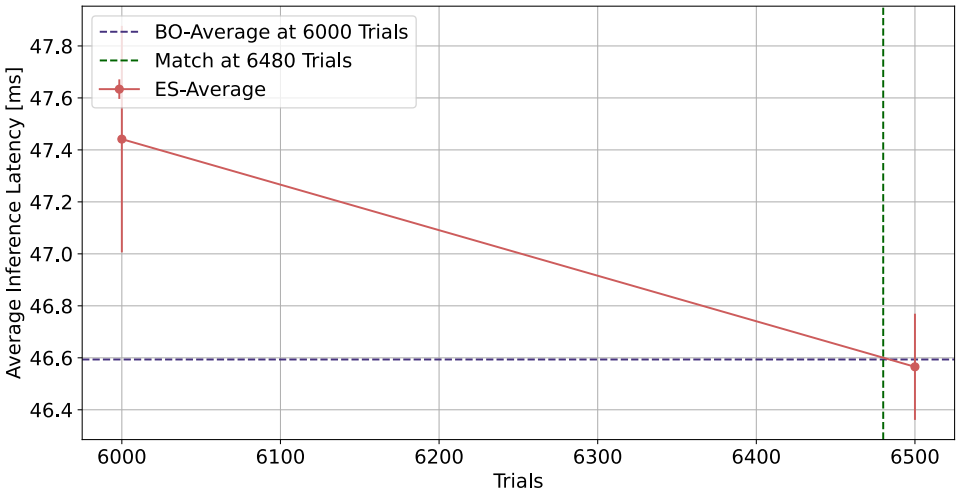


Figure A.3: BERT ES-BO Performance Match.

Appendix B

Additional Search Space Figures

The figures displayed in this chapter were created with the same Methodology as outlined in Section 4.4.1 for Figure 4.7.

When compiling BERT for a CPU target, BO struggled to match the performance of ES unless the exploitation of the search was increased; compare Section 4.3.1 and Figure 4.3. The transformation displayed in Figure B.1 comes from the task, `fused_nn_batch_matmul_4`, and shows an oscillating pattern, explaining why more exploitation can be helpful.

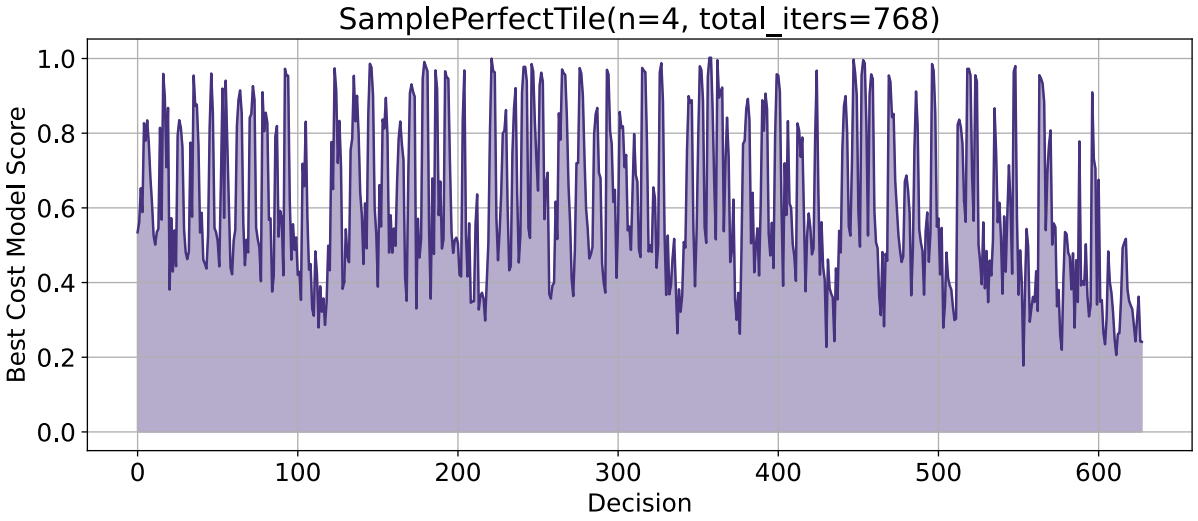


Figure B.1: CPU BERT Transformation Space Example.

Figure B.2 shows a transformation taken from the `fused_nn_dense_add_3` workload of GPT-2. It becomes clear how much of the search space is invalid and how small the areas with good performance are.

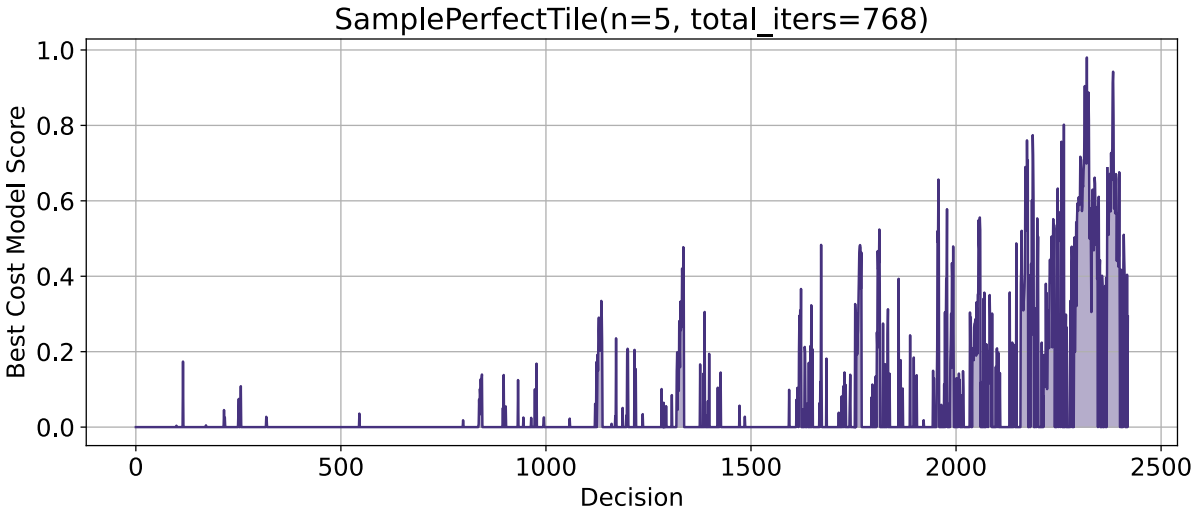


Figure B.2: GPU GPT-2 Transformation Space Example.

BERT was the only model on GPU that achieved strong performance compared to ES. Therefore, we had a closer look at its tasks. We picked the workload `fused_nn_batch_matmul_4` to observe closer. Interestingly, one of the transformations had an almost identical representation as Figure B.2; their other transformations were also similar except for one. This transformation can be seen in Figure B.3. Instead of the relatively wide performant regions, we can see in Figure B.3, the corresponding GPT-2 transformation (same extend and total loop iterations) has an oscillating pattern with narrow performant regions. This can potentially explain why BERT, compiled with BO, could outperform ES while GPT-2, compiled with BO, could not.

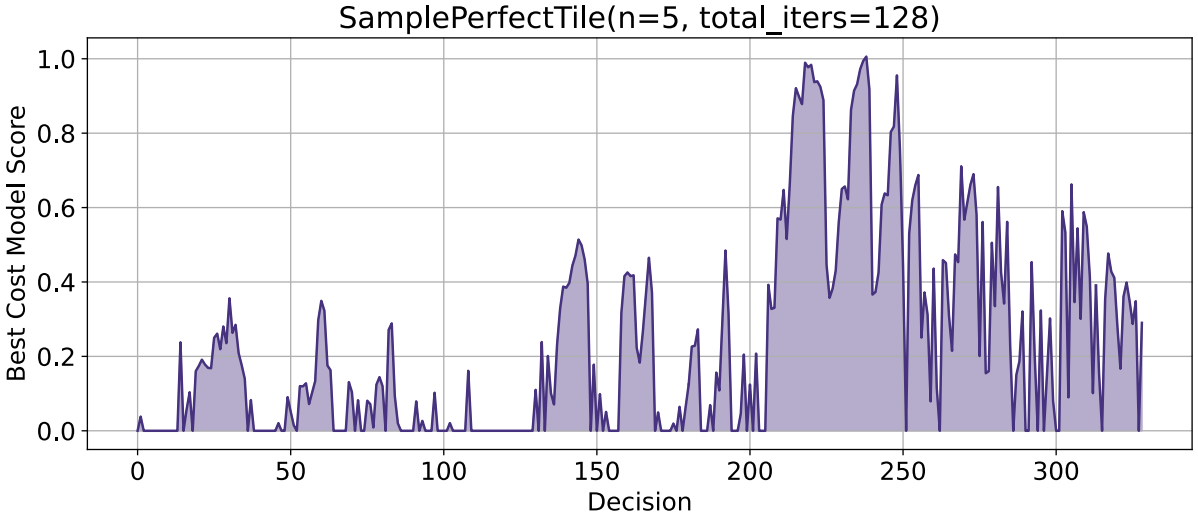


Figure B.3: GPU BERT Transformation Space Example.