Autotuning for Automatic Parallelization on Heterogeneous Systems

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Philip Pfaffe

Tag der mündlichen Prüfung: 24.07.2019

1. Referent: Prof. Dr. Walter F. Tichy
2. Referent: Prof. Dr. Michael Philippsen

Abstract

To meet the surging demand for high-speed computation in an era of stagnating increase in performance per processor, systems designers resort to aggregating many and even heterogeneous processors into single systems. Automatic parallelization tools relieve application developers of the tedious and error prone task of programming these heterogeneous systems. For these tools, there are two aspects to maximizing performance: Optimizing the execution on each parallel platform individually, and executing work on the available platforms cooperatively. To date, various approaches exist targeting either aspect. Automatic parallelization for simultaneous cooperative computation with optimized per-platform execution however remains an unsolved problem.

This thesis presents the `APHES` framework to close that gap. The framework combines automatic parallelization with a novel technique for input-sensitive online autotuning. Its first component, a parallelizing polyhedral compiler, transforms implicitly data-parallel program parts for multiple platforms. Targeted platforms then automatically cooperate to process the work. During compilation, the code is instrumented to interact with `libtuning`, our new autotuner and second component of the framework. Tuning the work distribution and per-platform execution maximizes overall performance. The autotuner enables always-on autotuning through a novel hybrid tuning method, combining a new efficient search technique and model-based prediction.

Experiments show that the `APHES` framework can solve the cooperative heterogeneous parallelization problem and that cooperative execution outperforms versions parallelized for a single platform. On benchmarks from the PolyBench benchmark suite, the `APHES`-transformed programs achieve a speedup of up to 6$\times$ compared to program versions generated by state-of-the-art single-platform parallelizers. The `libtuning` autotuner reduces the search time by up to 30% compared to state-of-the-art autotuning while still finding competitive configurations. Additionally, model-based prediction is is able to reduce 99% of the search overhead.

Ich versichere wahrheitsgemäß, dass ich die Dissertationsschrift mit dem Titel "Autotuning for Automatic Parallelization on Heterogeneous Systems" selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht, sowie die Regeln zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT) beachtet habe.

Philip Pfaffe Unterschleißheim, 22. Mai, 2020

Contents

Chapter 1

Introduction

The era of growing processor speed is over. Moore's law, which promised a doubling in performance every 18 to 24 months, has ended. This is a fact that must be accepted. But it is not an indicator of doom or deterioration of the industry! Instead it turns out to be an immense opportunity for system architects. Being at the end of the line of the development of more and more powerful CPUs has ignited the strive for developing the architectures of the future. Today, there are dozens of emerging parallel architectures, often optimized for specific tasks. A recent IEEE Spectrum article[1] quotes David Patterson estimating that there are "at least 45 hardware startups tackling the problem", making our time a "golden time for computer architecture". And these are not dreams of the future: Today's systems are highly parallel already, containing multiple general and special purpose parallel processors. This trend is not limited to the performance-hungry field of high-performance computing, but has reached low- to high-end consumer computers, down to the smartphone in everyone's pocket: Nearly every device is packed with multiple CPUs, GPUs for both graphics and computation acceleration, and specialized parallel compute units for various specific tasks.

The hardware landscape available to us within a single system has changed, and in the wake of this change an enormous challenge for the field of software engineering has risen. Effectively programming these highly parallel, heterogeneous systems is an intricate and time-consuming task even for highly trained and seasoned developers. Fortunately, various tools are available to ease the design, evaluation, and verification of parallel software. The list includes profilers, debug-

[1]Source: https://spectrum.ieee.org/view-from-the-valley/computing/hardware/david-patterson-says-its-time-for-new-computer-architectures-and-software-languages, September 2018. Last accessed: December 29th, 2018

gers and race detectors, recommender systems for parallelizable program parts, and automatic parallelizers. Fully automatic parallelization for heterogeneous systems is naturally the ultimate goal, because it reduces the required developer effort to zero.

To date, automatic parallelization approaches are mostly limited to single-platform targets, e.g., GPUs or CPUs. This is likely because targeting multiple platforms simultaneously is a notoriously hard problem. Simultaneously refers here to executing a program on different platforms *cooperatively* for distinct parts of the input, and subsequently fusing the results. We refer to this as *cooperative heterogeneous parallelization*. What makes this problem hard is the need to find an optimal configuration for the distribution of work among platforms and for platform-specific parameters at the same time. Examples for platform-specific parameters are the number of CPU or GPU threads and the mapping of program variables onto the GPU's memory hierarchy. The work distribution and the platform parameters are interdependent. Furthermore, the quality of a configuration for all the parameters is application, hardware, and most importantly input dependent. For example, consider a program performing matrix multiplications. For matrices with 4000×4000 elements, a GPU can offer a substantial acceleration. For 10×10 elements, however, the overhead we incur for GPU execution is sure to destroy all benefits and drastically slow down the program. While it is sensible to assume that performance-relevant application and hardware properties remain static during the lifetime of a program, the inputs must be assumed to be highly dynamic. For simplicity, we thus focus on input dependence in the following, but all following arguments can easily be extended to varying performance-relevant application and hardware properties by simply considering them a part of the input.

Given that inputs must be considered dynamic, configurations must consequently be selected at *application runtime*: Configuring a-priori could not account for varying inputs. This means, however, that the time required for selecting a configuration must be included in the cost of the parallelization. For example, if an optimal heterogeneous distribution for a parallelized program accelerates its execution by a factor of two, but determining that distribution takes as long as executing the program, we have obviously not gained any benefit. The time required to choose an optimal configuration thus determines the *performance of a configuring method*.

Fundamentally, for a given application, system, and input, there are two basic methods to produce an optimal configuration. These methods exhibit distinct performance characteristics. Obviously, the fastest way to produce a configuration is using an efficiently computable function that maps inputs to parameter values, which can be as simple as a precomputed lookup table. We call this method *predictive*, because it predicts a configuration for a given input based on a-priori knowledge. Knowledge can either be hand-coded, for example in the form of heuristics designed by the application developer, or learned, for example through machine learning. Alternatively, configurations can be explored *empirically* without a-priori knowledge. Empirical exploration means iteratively picking and observing configurations, which adds an interesting dimension to the performance criterion: On top of the time required to determine and apply the update to the parameters, we pay for configurations that are suboptimal. These might even yield program runtimes worse than in the original, unoptimized application. The extra overhead we incur from these configurations must be considered in the evaluation of the configuring method. In other words, we must consider the cumulative time spent sampling the program in relation to running the original program repeatedly. Assume for example the program is run ten times for a given input and empirical exploration determines a configuration which outperforms the original within ten steps. If, however, the exploration process evaluated configurations with worse performance than the original along the way, so that the *sum* of runtimes over the ten iterations is higher than ten times the original runtime, then it has effectively reduced the performance of the program. Consequently, when optimizing runtime, empirical exploration must not only find a configuration better than the original, it also must amortize quickly enough to provide a net benefit. The accumulated time spent sampling must be lower than the accumulated time for running the original program for the same number of sampling iterations. We call the time required to first reach that point the *amortization time.*

1.1 Problem Statement

The current state of the art in cooperative heterogeneous parallelization falls short in at least one of two crucial aspects:

1. Parallelization is usually limited to a given platform: Parallelizable code regions of a program are identified and then transformed to run on multiple

CPU threads, or (multiple) GPUs, or accelerators such as Intel's Xeon PHI cards. Although the heterogeneous system offers multiple different parallel platforms, the parallelized code executes only on a single platform while others are idle: No work is cooperatively shared across platforms. Only two approaches exist today that produce cooperatively executing code, but they are limited: Only the work distribution is optimized, not the per-platform parameters.

2. Since parallelization is done at compile time, all decisions need to be made heuristically. Heuristics are suboptimal in general: Multiple compilation decisions during parallelization are input dependent. Selecting platforms, determining the share of each platform in the cooperative execution, and choosing values for platform specific parameters all are key decisions to attain optimal performance. Some existing approaches solve this problem in similar use-cases with the help of models that are either hand-crafted by the compiler or application developer, or are learned from sample inputs. Although effective, the model design and training still require developer time and effort.

Fully automatic parallelization for optimized cooperative execution on heterogeneous systems that requires zero developer involvement is currently still an open problem. This dissertation aims to close that gap.

Based on these observations, we now formulate the requirements for a solution to the cooperative heterogeneous parallelization problem. Generating code for multiple platforms and for adaptive distribution of work across platforms requires a specialized compiler. The compiler may processes both implicitly and explicitly parallel languages and use any means to extract parallelism from sequential applications. In both cases however, it must be able to accurately determine all dynamic memory accesses made by the individual parallel units. If the heterogeneous platforms form a system with distributed memories (as is the case in most systems containing both CPUs and GPUs), data needs to be moved between the individual memories. Hazards arise if data movements are performed based on approximations of the accessed memory ranges.

Because the optimality of a configuration depends on inputs, a runtime component is necessary to supplement the parallelizing compiler, and to configure the transformed application. Parallelizing general applications implies several additional requirements: No assumptions can be made about program inputs or other

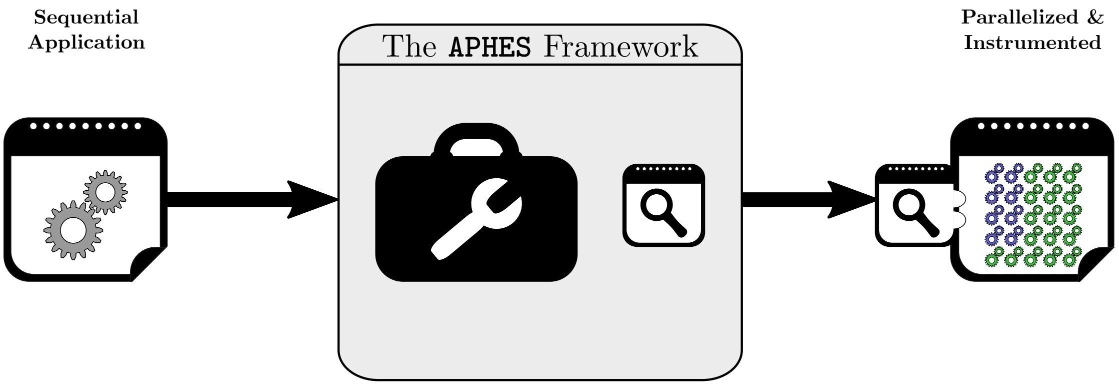


Figure 1.1: The APHES framework transforms sequential applications for work sharing among multiple platforms, e.g., a GPU and a CPU. The transformed application is instrumented to interact with a runtime tuning component.

application and hardware properties. This implies that both predictive and empirical configuring must learn from the real hardware and real inputs in the actual deployment context. Learning from real inputs further means learning at application runtime, which requires minimizing the amortization time of the tuning process. Minimizing the amortization time in turn requires avoiding search if possible. However, because the deployment context is unknown a-priori, training the prediction function still needs to obtain training samples during a production run. Choosing these samples randomly or from predictions of an early imprecise model bears the danger of increasing the amortization time. If search is used, optimizing the amortization time of the search is an additional requirement. This optimization means reducing the overall number of samples required to reach the (local) optimum and reducing the number of configurations sampled with a runtime worse than the original program's runtime. In summary, it is evident that whichever configuring method is used, great care must be taken to balance the drawbacks of either method: To be competitive, both must aim to sample as few poorly performing configurations as possible. For prediction this means selecting training samples carefully. Search is required to converge as quickly as possible.

1.2 Thesis Objectives

The overarching goal of this thesis is to provide a framework for automatic parallelization for heterogeneous platforms. The framework shall transform programs to simultaneously offload computations to multiple parallel target platforms. We will define the `APHES` framework, implementing a compiler for heterogeneous parallelization and a runtime component satisfying the requirements outlined above. The envisioned usage of the framework requires no application developer or user interaction and is depicted in Figure 1.1: The framework's compiler component consumes sequential applications and transforms them for work sharing among multiple heterogeneous platforms. The applications are instrumented by the compiler with the framework's runtime component, which optimizes the transformed applications at runtime in the actual deployment context. The core of the runtime component forms a novel online autotuner. Its primary goal is to provide input sensitive tuning while simultaneously being attentive to amortization time. Our key idea to achieve this is to combine predictive and efficient empirical configuring into a hybrid tuning method that aims to offer the best of both worlds.

We use the `APHES` framework to examine the following research theses:

Thesis $\mathbf{T_1}$ Optimized cooperative parallelization accelerates programs, exceeding the performance of single platform parallelization.

Thesis $\mathbf{T_2}$ The amortization time can be improved in comparison with state of the art autotuning.

Thesis $\mathbf{T_3}$ Predictive configuring achieves input sensitivity without sacrificing amortization time.

To evaluate these theses experimentally, we implemented the `APHES` framework prototypically. At the hand of established benchmarks composed of scientific programs, we test the theses by investigating the impact of autotuning on the result of the heterogeneous parallelization.

1.3 Scope

The `APHES` framework is an architecture for automatic parallelisation of sequential programs. As inputs, it takes programs in the LLVM intermediate language [LA04]. In order to simplify the required analyses, already parallelized programs are excluded from the scope. As the intermediate representation of compilers, the LLVM intermediate language constitutes a much lower level description of the program than the programming language it was written in. As such, some information that is present in the high-level frontend language is lost and needs to be recovered. Information that is lost is for example the structure of arrays, which is important to analyze memory accesses in the array. We accept these limitations to enable us to build upon a mature existing infrastructure for program analysis and transformation.

At the time of writing, parallelism is opaque within the LLVM intermediate language and only expressed as function calls into parallel libraries such as the OpenMP or pthreads runtime libraries. There is, however, work underway within the LLVM project to express parallelism at the language level, at which point the `APHES` framework can be easily extended to incorporate this information into its analysis stage.

We explore fully automatic cooperative parallelization of sequential programs. We focus the code transformations applied by the framework's compiler component to loop structures such as `while` and `for` loop constructs. Loops further must exhibit static control. This means that the number of loop iterations at runtime must be loop invariant, i.e., either constant or determined by a variable that is not modified within the loop body. `APHES` is targeted primarily at numerical programs which exhibit static control. The parallelism classes that are detected by `APHES` are data parallel loops and parallel reductions, which are the predominant patterns found in scientific programs. The heterogeneous parallel platforms `APHES` is intended to work with are CUDA for the GPU and OpenMP for the CPU.

The primary scientific contribution of this thesis is the hybrid autotuning approach and its application to cooperative parallelization. It is outside the scope of this thesis to invent a new method of extracting parallelism from a program: Decades of research have produced a catalogue of program analysis and parallelization techniques that are adequate to find parallelism and transform it for cooperative execution. The `APHES` framework reuses existing algorithms and implementations wherever possible.

1.4 Structure of this Dissertation

In the following chapter we discuss the technical foundations behind the design and implementation of the `APHES` framework and introduce terms and definitions. Subsequently we provide an overview of the `APHES` framework and its components, and show how application developers may use the tools we developed. In Chapter 4 we explore related prior art and assess existing solutions with respect to requirements that arise in cooperative parallelization. Then, we discuss the `APHES` framework in detail, focussing first on the autotuning component `libtuning` in Chapter 5 and the `aphes` compiler in Chapter 6. Chapter 7 presents an experimental evaluation of our approach with regard to the theses posed in Section 1.2 on the basis of our implementation of the `APHES` framework. In Chapter 8 we summarize the results, re-examine our theses with respect to the findings of our evaluation, and discuss possible future directions in research motivated by our findings.

Chapter 2

Fundamental Concepts

This chapter introduces the fundamental concepts upon which this thesis builds. In particular, we present the concepts of autotuning and automatic parallelization. We begin by defining the autotuning process, and present state of the art algorithms and techniques that form the basis for the runtime component of the `APHES` framework in Section 2.1. Second, we review compiler techniques and tools in Section 2.2. This includes an introduction to analyzing programs for parallelizable parts.

2.1 Online and Offline Autotuning

In today's research and practice the task of configuring program parameters is referred to as *autotuning*. In this section we first define the terms and concepts used in autotuning. Subsequently, we introduce two popular search algorithms for model- and derivative-free empirical tuning. We lastly provide an introduction to reinforcement learning, a special form of predictive tuning which forms the basis of the online learning mechanism used in our hybrid autotuner.

2.1.1 The Tuning Problem

Autotuning has been around since the 1990s, and was popularized by the ATLAS library [WD98]. The goal of autotuning is to optimize degrees of freedom in a program for some target function. Usually that target function is program runtime, but alternatives exist: In recent years, in particular the interest in optimizing

```
vector<int> add(vector<int> A, vector<int> B,
                unsigned NUM_THREADS) {
  vector<int> Result(A.size());

#pragma omp parallel for num_threads(NUM_THREADS)
  for (size_t I = 0; i < Result.size(); ++i)
    Result[I] = A[I] + B[I]

  return Result;
}
```

Figure 2.1: Vector addition using OpenMP

the energy efficiency of programs has grown.[1] The degrees of freedom, which are program variables, are called the *tunable parameters* of the program. Tunable parameters are program variables whose values do not influence the result of the program, but only the tuning objective, e.g., the runtime. Consider for example the OpenMP code snippet in Figure 2.1 implementing parallel vector addition. The code is simple and straightforward: Using the `omp parallel for` annotation, it executes the `for` loop in parallel using the requested 8 threads, adding vector elements. The number 8 here is a model example for a tunable parameter. Changing it does not change the semantics of the function, but clearly affects the performance. Without autotuning, the number 8 is magic: The program developer either needs to guess by rule of thumb, or needs to evaluate multiple values empirically and pick the best. Autotuning is a tool to automate this empirical process.

An autotuning process using a stylized `Tuner` tool for the example above might look like shown in Figure 2.2. Instead of encoding the thread count as a constant, we make it a global variable. We then repeatedly call the `add` function, measuring its execution time for different thread count values. The `Tuner` tool produces the sequence of thread count values to be tested, changing with the timing feedback.

Autotuning techniques involve both predictive and empirical approaches. Predictive tuning can be heuristical, meaning that parameters are configured using

[1] See for example the work of Jordan et al. [Jor+12], Balaprakash et al. [BGW13], or Bao et al. [Bao+16].

```
vector<int> add(vector<int> A, vector<int> B,
                unsigned NUM_THREADS); // As above, unchanged

void tune_add(vector<int> A, vector<int> B) {
  Tuner T;
  while(T.keepTuning()) {
    unsigned NUM_THREADS = T.getNumThreads();
    // execute add with arguments and measure execution time
    float Time = time_call(add, A, B, NUM_THREADS);
    T.feedback(Time);
  }
}
```

Figure 2.2: Example: Tuning the OpenMP thread count

rules hand-crafted by the application developer. Because manually writing rules is a time consuming and error-prone task, more sophisticated approaches use *model-based* prediction. For these, a model is built, mapping application, hardware, and input features to configurations. The model can be constructed either manually, or it can be learned from sample data. The advantage of model-based configuring is that configuring is a one-shot operation: To pick a parameter configuration, the application merely needs to query the model for the current application, hardware, and input. On the flip side, the quality of that configuration is only as good as the design of the model and its training. When a hand-crafted model is inaccurate or when the data seen during training does not reflect the real deployment context, its predictions are non-optimal. *Empirical autotuning* lifts the burden of model building and training from the application developer. Instead of providing a one-shot answer, empirical tuning iteratively samples configurations. The downside of the iteration is the time spent until a satisfying configuration is found. If the number of possible configurations is large, as it often is in practice, it can render an exhaustive exploration of configurations infeasible. Hence, practical applications of empirical tuning must resort to approximating the global search, which implies that the produced configuration may only be locally optimal. Therefore, the quality of the result of empirical tuning depends on the time spent tuning, but it can nevertheless reduce the necessary effort by the application developer.

Both forms of tuning can be either *white-box* or *black-box.* White-box tuning is tightly coupled with a specific application, and it decides between configurations based on application properties. Empirical white-box tuning systematically explores configurations based on rules defined by the application or tuner developer. The quality of these rules affects the quality of the optimization as heuristics and hand-crafted models generally do. Black-box tuning, in turn, only observes parameters, the measurement function values, and, in case of model-based tuning, input and application features. Tuning decisions are made purely based on these observations, which are free of the hazards of developer misconceptions. On the other hand, a black-box tuner might pick suboptimal configurations which could have been avoided using domain knowledge.

Independent of the actual application, the autotuning process follows always the same recurring structure that is shown in the example in Figure 2.2. Within a loop, which we call the *tuning loop*, a tuner first configures all tunable parameters with the next configuration to sample. Then it executes the piece of code that is to be optimized with the sample configuration. We refer to this piece of code as the *tuning kernel.* After executing the kernel, the observed measurements are fed back into the tuner for it to determine the configuration for the next iteration. The tuning loop is repeated until a termination criterion is met.

The tuning loop and its termination criterion occur in various forms in practice. Most often, they are not part of the program being tuned, but the whole program is executed in an autotuning environment for a set of example inputs. This approach is called *offline tuning.* The termination criterion is in that case most often either time passed or a minimum performance threshold. This means, tuning continues either for a given time, or until a target program performance is achieved. Alternatively, the tuning loop can be placed within the program, or it may even be a natural part of it. Since the tuning kernel is an important and performance critical part of the program, it will naturally be executed recurringly.[2] Autotuning can then piggyback on these naturally occurring tuning loops. In either case, this approach is referred to as *online tuning* since optimization happens in a production environment on real inputs.

While the tuner functions the same way in both offline and online tuning, the classes differ in the requirements they impose on the tuner's *tuning overhead.* The

[2]Otherwise, if the tuning kernel were not performance critical or executed only rarely, any form of performance tuning can offer little meaningful benefit.

tuning overhead comprises the time required to compute and apply a new configuration and the time spent sampling configurations. Offline tuning is an operation that is performed once, and the resulting configuration is used indefinitely. Thus, the tuning overhead is rarely of importance, as long as a satisfactory configuration is found. In online tuning on the other hand, the tuning overhead is visible to the program user. Thus, the tuner is required to minimize the overhead, potentially at the cost of the quality of the configuration it finds. While the sensitivity to tuning overhead puts online tuning at a disadvantage, it is able to work on real inputs and real hardware in an actual deployment context.

Next, we formalize the autotuning problem, following the formalism we introduced in our 2017 article [Pfa+17]. Formally speaking, autotuning is the task of optimizing a value function by modifying only tunable parameters of a program. A *tuning parameter* τ_j is a program variable j together with its sets of legal values. The set of all tuning parameters of a tuning kernel form the tuning kernel's *search space* T:

$$T = \tau_{j_1} \times \tau_{j_1} \times \ldots \times \tau_{j_J}.$$

Autotuning explores the search space to minimize a *measurement function* $m_K : T \to \mathbb{R}$, searching for its global minimum:

$$C_{opt,K} = \underset{C \in T}{\arg\min}\, m_K(C).$$

Therein, K defines the current dynamic *context*. The measurement function m_K maps configurations and the current dynamic context onto a tuning target, such as tuning kernel runtime or energy efficiency. The context is composed of the *application context* K_A and the *system context* K_S: $K = (K_A, K_S)$. The application context describes dynamic properties of the tuned application, such as the input size. The system context quantifies characteristics such as momentary system load or available hardware features, e.g., number of cores. For a given *configuration* $C \in T$, $m_K(C)$ is, for example, a measurement of the execution time of tuning kernel under the parameter configuration defined by C.

We categorize the tuning parameters τ_j according to Steven's topology [Ste46] into one of the four classes Nominal, Ordinal, Interval, and Ratio Parameters. This classification is summarized in Table 2.1. Although a parameter may fit in any one particular class among these four, the most relevant distinction must be made between a nominal and a non-nominal parameter. A nominal parameter is one that selects between labels or categories. These occur frequently when programs may

Table 2.1: Parameter Classes [Pfa+17].

Class	Distinguishing Property	Example
Nominal	Labels	Choice of algorithm
Ordinal	Order	Choice of buffer sizes from a set `small`, `medium`, `large`
Interval	Distance	Percentage of a maximum buffer size
Ratio	Natural Zero Equality of Ratios	Number of threads

choose between variants of code, such as between different algorithms, or between different execution platforms. Boolean parameters also fall into this category. The key difference between parameters belonging to the nominal class and those that do not is that within a nominal parameter space, there is no notion of distance or direction. This notion however is important, because it is mandatory for the vast majority of search algorithms used in empirical tuning. Reconsider the previous thread count tuning example in Figure 2.2. We execute the tuning kernel first with 2 threads and then with 4 threads, and for simplicity assume the 4 thread version is twice as fast. Based on this observation, a search algorithm may sensibly assume that more threads are better than fewer threads, and subsequently sample, e.g., an 8 thread version. While this version might just not be faster, for instance if there are less than eight hardware threads, the assumption is a valid hypothesis to test. If, on the other hand, we were not to optimize the thread count, but the selection between different implementations of `add`, the tuning parameter is nominal. That means, having measured the runtime of an `omp_add` and an `sse_add`, the search algorithm cannot make any meaningful assumptions about the runtime of a `cuda_add`, because there is no means to relate the three with each other[3]. Consequently, search on nominal parameters is inherently combinatorial.

2.1.2 Search Algorithms

Having introduced search spaces, tuning parameters, and parameter classes we will now discuss search algorithms. After decades of research, an enormous catalogue

[3]Assuming black-box tuning.

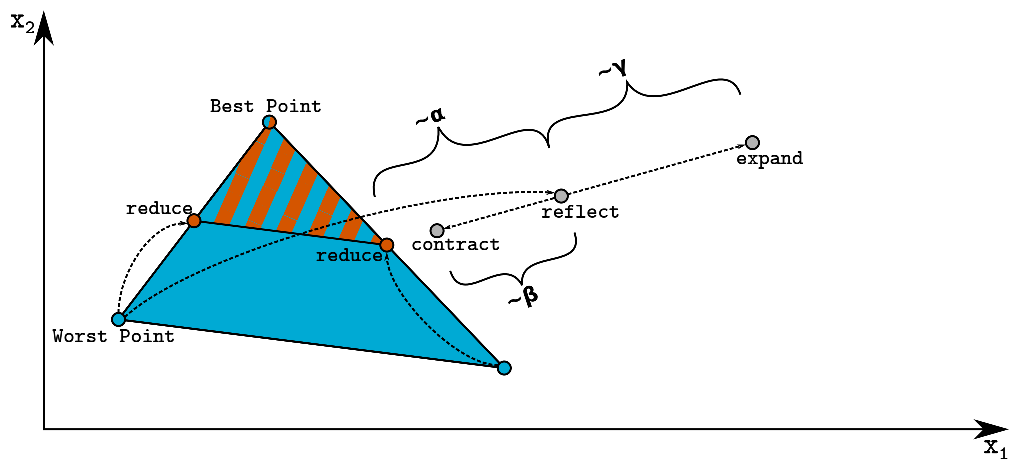


Figure 2.3: An illustration of the Nelder-Mead algorithm operations `reflect`, `expand`, `contract`, `reduce` sampling points starting from the blue 2D simplex. A reduction operation produces the orange result. The function values of the points are not shown.

of different search algorithms exists, which we will not review here in breadth. For a broader overview, we refer to Marthaler [Mar13] for a survey on numerical optimization methods and to Ashouri et al. [Ash+18] for a recent survey on machine learning methods, in particular in the context of compiler autotuning. In this section, we introduce the two algorithms on which the autotuner presented in this thesis is built upon, namely Nelder-Mead and ϵ-Greedysearch.

The Nelder-Mead method [NM65] is an extension of the simplex function minimization method originally by Spendley et al. [SHH62]. It maintains a simplex of points in n-dimensional real space, storing the simplex' $n+1$ point in sorted order with respect to the measurement function. Figure 2.3 illustrates the sampling behavior in the two-dimensional case, in which a simplex is a triangle. The initial simplex in the example is colored in blue and its worst and best points are marked. At every time step, the algorithm then executes one of the four operations `reflect`, `expand`, `contract`, or `reduce`:

`reflect`: Mirror the worst simplex point at the centroid of all but the worst point with mirroring coefficient $\alpha > 0$. The coefficient determines how far the new point is moved along the axis through the worst simplex point and the centroid. In Figure 2.3 this operation moves the "worst" blue point in the bottom left to the point labelled `reflect` along the line through the centroid.

In the 2D case the centroid is halfway between the two right-hand points of the blue triangle. After evaluating the measurement function for the new point, if its value lies between those of the best and the worst point, exchange the worst point for the reflected point and start over. Else, if the reflected point is a new minimum, perform an expansion, or else, if the reflected point is at least as bad as the worst tracked point, perform a contraction.

`expand`: The reflected point is a new minimum. Expand the simplex by moving the reflected point further along the same axis as in the `reflect` step using the expansion coefficient $\gamma > 1$. Figure 2.3 illustrates this step moving from the point labelled `reflect` to the point labelled `expand`. Exchange the worst simplex point for either the expanded or reflected point, whichever is better, and start over.

`contract`: The reflected point is at least as bad as the current worst point. Pick whichever of the two is better, and move it toward the centroid by contraction coefficient $0 < \beta < 1$. In Figure 2.3 we see this step as the move from the point labelled `reflect` to the point labelled `contract`. If this produces a better point, exchange it with the worst simplex point and start over. Otherwise, perform a reduction.

`reduce`: Move every simplex point halfway towards the current minimum. Every new point is evaluated to determine the new best and worst points. Figure 2.3 shows this operation moving from the bottom blue points to the bottom orange points, creating a new simplex shown as the orange triangle.

You et al. [YSD05] extended this original algorithm to better capture the requirements of empirical program optimization. Primarily, they deal with tuning parameters being discrete, such as the thread count parameter in the example above in Figure 2.2. They convert from the points in real space produced by Nelder-Mead to integer by floor rounding. Second, they support boundary and interior constraints, both of which frequently are necessary in program autotuning: The set of legal values is usually finite and often defined as closed intervals of values. Additionally, specific combinations of values from within the legal ranges can also produce illegal configurations. The search must be constrained to the boundaries of the value intervals and must avoid the illegal interior points. You et al. handle this through stationary penalty values. That means, if a point p produced by Nelder-Mead search either lies outside of the application specific bounds, such as a negative thread number, they simply return $m_K(p) = \infty$. They also notice the importance of defining how to start the search. If the points of the starting simplex turn out to lie on a (hyper-)plane, the search is limited to moves on the plane and can never explore in a direction perpendicular to it. To avoid this, the You et al. initialize the simplex so that it has a non-empty but random volume.

The second algorithm we present in this section is called ϵ-Greedy. The algorithm is frequently used in solutions to multi-armed bandit problems [SB98]. In this type of problem, an agent must repeatedly choose among a set of options with an unknown reward, which resembles playing a slot-machine with multiple arms and unknown payouts. Tuning nominal parameters can be seen as a multi-armed bandit problem, which motivated the choice of this particular algorithm. The ϵ-Greedy algorithm is remarkably simple: With a probability of ϵ, select a configuration randomly, otherwise greedily pick the best configuration found so far. Variants of the algorithm also sometimes decrease the ϵ over time to force the search to converge to a value [KLM96].

2.1.3 Reinforcement Learning

Reinforcement Learning (RL) is a class in the field of machine learning for solving control problems. In this section, we provide an overview of the general problem formulation, as well as a selection of solution methods. For a more in-depth introduction we refer to Sutton and Barto [SB98]. All definitions in this section are based on Sutton and Barto unless otherwise noted.

The RL control problem is usually formulated as a Markov Decision Process (MDP), in which an *agent* iteratively navigates an unknown *environment*. In every iteration, the agent observes its current *state* in the environment, and selects and executes an *action*. It makes this choice based on an action-selection *policy*. After carrying out the action, it receives a *reward* from the environment and transitions into the next state. The agent attempts to maximize its accumulated reward over time, and Reinforcement Learning methods are used to determine policies to achieve that. The state transition after executing an action need not be deterministic, i.e., an action takes the agent into the next state according to a probability distribution in general. The sequence of action-transition steps in the process can be either *infinite-horizon* or *episodic*. In the former, iterations go on forever, indefinitely continuing state transitions. In the latter, iteration happens in an episode until some termination state is reached, and is then restarted in an initial state. This case allows for a delayed distribution of rewards only at the end of each episode instead of after every action. The typical example for an episodic MDP with delayed rewards are games. Here, the agent plays a board or computer game against a human or machine opponent, and receives a score once the game is complete. TD-Gammon [Tes95] is a well-known instance of this, where a Temporal Difference learning (an RL method which we further describe below) agent learned to play Backgammon by training against world-class human players.

Formally, an MDP is the quadruple (Σ, A, Φ, ρ) where Σ and A are the sets of discrete states and actions, respectively. The action model $\Phi : \Sigma \times A \times \Sigma \to [0, 1]$ gives the probability $\Phi(s, a, s') = P(s'|s, a)$ of transitioning into state s' when performing the action a in state s. The reward function $\rho : \Sigma \times A \times \Sigma \to \mathbb{R}$ defines the reward received from the environment when going from state s to s' when performing action a. When applying RL to solve this MDP, Φ is typically not known, and is learned along the way.

After three decades of research, there is by now a substantial body of RL methods [Aru+17; SB98]. We focus on the two classical model free variants: Temporal Difference learning (TD) [Sut88] and Q-Learning [WD92]. Both operate in a similar manner, in that they attempt to maximize the expected future reward. Thus, they assign a *value* $V(s)$ to every state s. Temporal Difference learning computes this value as follows. After every step, it updates the value of the previous state s according to

$$V(s) = V(s) + \alpha \left[\rho(s, a, s') + \gamma V(s') - V(s)\right].$$

Here, α is called the learning rate, and $\gamma \in [0, 1]$ is a discount factor controlling the influence of expected future rewards. An agent setting $\gamma = 0$ is sometimes called *myopic*, in that it makes decisions purely based on immediate rewards and ignores any possible future gain.

TD is an instance from the class of so-called *on-policy* methods. On-policy methods learn a policy by using the policy. An on-policy agent can hence be considered to be always *exploring*. In particular this means the agent cannot be trained using a stochastic policy, e.g., randomly trying actions for a while, and then switching to a different policy, e.g. one greedily choosing the action most probably leading to the highest valued state. Learning about one policy (called the *target* policy) from data obtained through a different policy (called the *behavior* policy) is instead the discriminating feature of *off-policy* methods.

Watkins' Q-learning is an off-policy method for online learning. Instead of computing state-values as in TD, Q-learning approximates the optimal value function through state-action-values. Watkins noticed that the future development of the expected reward only depends on the current state and the chosen action [Wat89]. Thus, his state-action-value function $\mathcal{Q} : \Sigma \times A \to \mathbb{R}$ assigns each state and action pair the expected future reward. A trivial but optimal target policy can then be $\pi_t(s) = \arg\max_a \mathcal{Q}(s, a)$. To train $\mathcal{Q}$, Watkins defines the update after performing an action in time point t

$$\mathcal{Q}_{t+1}(s,a) = \mathcal{Q}_t(s,a) + \alpha \left[\rho(s,a,s') + \gamma \max_{a'} \mathcal{Q}_t(s',a') - \mathcal{Q}(s,a)\right]$$

The parameters α and γ are as above in TD learning. The behavior policy used during training is now another degree of freedom. One popular choice often used in both literature and practice is ϵ-Greedy, which chooses a random action with probability ϵ, and the best known action otherwise. Interestingly though, because Q-learning is off-policy, it can also consume data obtained through third party sources, for example a human controller, or data recorded in previous experiments.

The models and methods discussed so far in this section are subject to a strong restriction: Both action and state spaces are required to be discrete. This is an impactful limitation, because it excludes a wide range of control problems which have continuous states, unless the continuous inputs are proactively discretized. Particularly relevant examples are all problems that use time as input, which is continuous. But even if the problem being modeled has discrete states, there arises a practical difficulty when the state space is large. Tabular RL methods as described above need to keep the state value function or state-action-value function as a table in memory. Storing the complete table quickly becomes impractical. An alternative is to approximate the mapping as a function. Early results have shown this to be problematic, such as Baird's well known counterexample for which Q-learning with function approximation diverges [Bai95]. Recent advances in the theory of machine learning and stochastic gradient descent in particular have lead to several new gradient-based RL algorithms with much stronger convergence guarantees such as Greedy-GQ [Mae+10]. Greedy-GQ offers several benefits which make it an interesting candidate for online tuning scenarios: It is online, incremental, and efficient, involving memory and computation costs linear in the number of features. It is however limited to linear function approximation. The general framework of the algorithm is as introduced above for tabular Q-learning. Instead of the tabular $\mathcal{Q}(.,.)$, however, it computes $\mathcal{Q}_\theta(s,a) = \theta \cdot \varphi(s,a)$, where $\varphi(s,a) \in \mathbb{R}^d$ are non-linear features and $\theta \in \mathbb{R}^d$ are the parameters to be learned. Using Sutton's weight-doubling technique [SMS08], the iterative update for an ϵ-Greedy policy is defined as

$$\theta_{t+1} = \theta_t + \alpha_t \left[\delta_t(\theta_t)\varphi_t - \gamma \left(w_t \cdot \varphi_t\right) \hat{\varphi}(\theta_t)\right],$$

$$w_{t+1} = w_t + \beta_t \left[\delta_t(\theta_t) - \varphi_t \cdot w_t\right].$$

Here, α and γ are as before, and β is the learning rate for the second weights. The φ_t are abbreviations for $\varphi_t(s_t, a_t)$. The temporal difference error $\delta_{t+1}(\theta)$ is analogous to the TD update above. Lastly, $\hat{\varphi}$ is $\hat{\varphi}_{t+1}(\theta) = \varphi(s_{t+1}, a')$ for $a' = \arg\max_a \mathcal{Q}_\theta(s_{t+1}, a)$. Maei et al. prove the convergence of their Greedy-GQ algorithm still under strong conditions, but claim that it shows to be robust beyond their proof in practice. In particular, Greedy-GQ converges for Baird's counterexample.

2.2 Program Analysis and Transformation

In this section we provide an introduction to the compiler construction concepts and tools that form the basis of the cooperative parallelization within the `aphes` compiler. In particular, we introduce the LLVM framework and the analyses we used to extract parallelizable parts from a program. The latter includes both classical dependence testing techniques as well as the more formal polyhedral model.

2.2.1 The LLVM Framework

We implement `aphes` on top of the LLVM framework [LA04], an infrastructure for building compilers and programming language tools. The framework is the basis for several compilers and multiple languages, such as C++, Java, and Rust. An LLVM-based compiler is generally a pipeline of three components. The *front-end* translates the programming language into an intermediate representation, the LLVM IR. The *middle-end* then transforms this IR to optimize the program. Lastly, the *backend* first converts this IR to a second intermediate representation which more accurately reflects properties of the targeted register machine. Second, this machine IR is further optimized for the selected target platform and then lowered into binary code. Our compiler reuses both the LLVM frontend for the C-family of languages as well as the backends for the platforms we target. The

```
@S = private constant [13 x i8] c"Hello␣World\0A\00", align 1

declare dso_local i32 @printf(i8*, ...)

define dso_local i32 @main() {
entry:
  %1 = getelementptr [13 x i8], [13 x i8]* @S, i32 0, i32 0
  %2 = call i32 (i8*, ...) @printf(i8* %1)
  ret i32 0
}
```

Figure 2.4: LLVM intermediate representation of a "hello world" program.

frontend is part of the `clang` compiler. Additionally, we use program analyses and transformations from the LLVM middle-end, and will thus give a more detailed overview of the middle-end and the intermediate representation in the following.

The LLVM IR represents the source program in a RISC-assembly-like form. Figure 2.4 shows an example "Hello World" program in LLVM IR. The representation is Static Single Assignment (SSA) based, meaning that every variable, called an *SSA register* and written with a leading `%` in LLVM IR, is assigned only once statically. To express multiple assignments to a variable in the source language, the SSA form contains virtual *phi nodes* that select operands according to control flow. The largest unit of IR is the *module*, which describes a single compilation unit. The module contains *function* declarations and definitions, *type* definitions, *global variables* and *metadata*. In the example, `@S` is a global variable. Function declarations, such as for `@printf`, contain the function name, the return type, and the argument types. Function definitions, such as for `@main`, additionally define a list of *basic blocks*. Basic blocks are the nodes of the function's control flow graph, linked via branch edges, and contain the SSA *instructions*. The `@main` function contains a single basic block, labelled `entry`, which is terminated by a `return` instruction.

Program optimization in the LLVM middle-end is organized into passes, scoped to the different units of IR. A pass can be either an *analysis* or a *transformation*. An analysis is an operation that only reads IR and computes information, which is then cached for further use. As an example, consider points-to or alias analysis, which for every pair of pointer variables in a function computes whether they may ever point to the same object or not. A transformation is something that queries analysis results and mutates the IR, potentially invalidating previously cached analyses. An example for this is inlining, replacing a call to a function with a copy of its body. The middle-end applies sequences of transformations known as *pipelines* to the IR. As such, it alternates between performing multiple transformations on a function, or on a module, or on a loop, for instance.

In the remainder of this section, we describe two analyses implemented in LLVM in closer detail. These analyses, which are `AliasAnalysis` and `ScalarEvolution`, are those which our parallelization compiler relies most strongly on.

Alias analysis computes information to disambiguate memory accesses. Whenever memory is loaded or stored through two pointer variables, it decides whether it is possible that the two variables ever refer to the same bytes of memory. In general, a precise answer to that question is impossible, which follows from the halting problem. The analysis must make worst-case assumptions in that case. In all other cases it can respond with a “yes” or “no” answer. For example, distinct global variables or stack allocations cannot alias, neither can statically known different array indices on the same array. Beyond this simple reasoning, LLVM implements more complex, more precise, and more expensive analyses. Examples for this are the inclusion-based or unification-based analyses based on the context-free language reachability problem from the works of Zheng et al. and Zhang et al. [ZR08; Zha+13].

Unlike alias analysis, which attempts to describe the evolution and propagation of pointers throughout a function or program, `ScalarEvolution` analyzes the evolution of scalar variables across the iterations of a loop. It attempts to derive a closed form for the value contained in a variable, dependent on the iteration variables of containing loops. In essence, the analysis computes this closed form through symbolic evaluation of chains of recurrences, which is based on the work of Van Engelen and Bachmann et al. [Eng00; BWZ94]. With its help, passes can determine the (symbolic) value stored in a variable at any given loop iteration.

```
if (loadData()) {
  data = prepareData();
  consumeData(data);
}
```

Figure 2.5: Control Dependences: Execution of the `consumeData` task is conditional on the data produced by the `loadData` task.
Data Dependences: The `consumeData` task requires data produced by the `loadData` task.

2.2.2 Parallelism Detection, Dependence Graphs, and Interprocedural Analyses

In a parallel program, the fundamental components of parallel execution are *tasks* and *data*. Tasks read, modify, or create data. If there are multiple different tasks operating concurrently, this mode of execution is called *task parallelism*. If, on the other hand, the same task executes concurrently for different data, this is referred to as *data parallelism* [MRR12].

Parallelizing a sequential program means transforming it, so that formerly sequentially executed tasks now operate in parallel. Both task and data parallelism can be produced, both in isolation and in combination. A parallelizing compiler detects the potential to produce a parallel program. This can be done either by identifying distinct tasks that may run in parallel, or by finding a task that can be replicated to operate only on parts of the data. The parallel program preserves the original program semantics, but exhibits a different order of execution. For the necessary transformation to be sound, the compiler must prove that it does not violate *dependences* present in the original program. Dependences can be either *data dependences*, or *control dependences* [HP11]. A control dependence exists between two tasks, if the execution of the second depends on the result of the first. In Figure 2.5, e.g., `consumeData()` is control dependent on `loadData()` because of the conditional branch. The `consumeData()` task may thus only execute once the `loadData()` task is completed. A data dependence exists when a task requires data produced by another task. In the example, the `consumeData()` task is data dependent on the `prepareData()` task because it requires the prepared data.

There are four types of data dependences [GKT91; HP11]. Assume a program in which tasks A and B run in sequence, then the four dependence types are:

Input dependence (read-after-read): Task B reads a variable that task A also reads. This dependence is generally benign, and thus does not prevent parallel execution.

Output dependence (write-after-write): Task B writes a variable that task A also writes. In a parallel execution, this would non-deterministically alter the value any subsequent, non-concurrent tasks observe.

Flow dependence (read-after-write): Task B reads a variable that task A writes. Task B might see the old value in a parallel execution.

Anti dependence (write-after-read): Task B writes to a variable that task A reads. Task A thus might see any of the two values in a parallel execution.

Dependent tasks (other than Input dependent) cannot be soundly executed concurrently. Automatic parallelization thus requires finding tasks which are not ordered through either control or data dependences. To determine control dependences, compilers compute the post-dominance frontiers for a function [FOW87]. The post-dominance frontier is a function that maps basic blocks to their successors which are not post-dominators. A post-dominator of a basic block is an immediate or indirect successor that is part of every path from the basic block to a function exit. The post-dominance frontier is a static property of the control flow graph and is easy to compute. Data dependences, however, are much harder to determine. This is for two reasons. When accessing arrays through pointer variables, the compiler first must determine for two accessed variables whether these refer to the same array. Second, it must decide whether the index expressions of the accessed array elements ever assume an identical value, but the index expression may be computed from an arbitrarily complicated expression, or it might be loop variant or even unknown in case the index is loaded indirectly from other variables.

The first problem, whether pointers refer to arrays, is approached by *points-to* or *alias analyses*. Alias analyses make various trade-offs, balancing precision and speed [Hin01]: A flow-sensitive analysis for example considers control flow through a function. A context-sensitive analysis considers the calling context of a function. The precise analysis even has been shown to be NP-hard [LR91; Hor97].

```
1 for (int i = 1; i < N; ++i) {
2   float a = A[i + N];
3   float b = A[i + 2 * N - 1];
4   A[i + 2 * N] = a + b;
5 }
```

Figure 2.6: Example: A data dependent computation

Consequently, compilers need to make pessimistic assumptions about which arrays are accessed by a single instruction.

Having determined which arrays two memory accesses may refer to, dependence analysis can then be performed to solve the second problem. This analysis often uses a technique called *dependence testing*. The algorithm by Goff et al. [GKT91] is an instance of this technique. The general idea of the algorithm is to first run a sequence of simple and cheap tests that cover the majority of access patterns found in practical software. Only if those are inconclusive they defer to more general and expensive analyses such as Banerjee's inequalities and the GCD test [Ban88; Wol82]. To illustrate the algorithm, consider the example in Figure 2.6. Intuitively, the write in line 4 to `A[i+2*N]` depends on the value of `A[i+N]` read in line 2 in the iteration `i`. Given the loop bounds, we easily see that this index is never written in the loop. More general, for this type of index expressions of the form $a*i+c_1, a*i'+c_2$ the dependence testing algorithm calculates the *dependence distance* as $d = i - i' = \frac{c_1 - c_2}{a}$. The dependence distance is the number of iterations that lie between the two accesses. For the accesses in lines 2 and 4 of Figure 2.6 to access the same bytes of memory, there need to be N iterations between first the write in iteration i and later the read in iteration $i + N$. Substituting the values from the example in the dependence distance formula, we indeed get $d = N$. Since $d > N - 1$, the number of loop iterations, we can disprove dependence for these two accesses. On the other hand, considering the accesses in lines 3 and 4 instead, we get $d = 2 * N - 2 * N + 1 = 1 \leq N - 1$, which means there exists a flow dependence here.

Given both the control and data dependences, the compiler can now search for parallelizable tasks. An established data structure facilitating this job is the *program dependence graph* (PDG) [FOW87]. The PDG is a graph representing a function whose nodes are instructions and edges are control or data dependences.

In this representation, detecting both task and data parallelism is straightforward. Any two distinct subgraphs where neither one is reachable from the other describe tasks that can be executed concurrently. Similarly, any task that is not part of a cycle is a candidate for data parallelism. An unfortunate limitation of the PDG is that it only analyzes a single function. This means that tasks containing function calls cannot be considered for parallelization. Interprocedural analysis, i.e., looking at multiple functions at the same time, offers a way around this limitation. Unfortunately, precise interprocedural analysis is hard [Rep96]. Function summaries as described by Sharir and Pnueli [SP78] however present a technique to trade precision for speed and represent function calls within a PDG. The general idea is to produce a summary for functions recursively, starting at the leaves of the call graph, i.e., those functions which contain no calls. The summary describes accesses to arrays not defined within the function. Callers of the function build their own summary by virtually inlining the summaries at call sites. This of course only works if a function's accesses are computable, and the function is not part of a cycle in the call graph, which are present in recursive programs.

2.2.3 The Polyhedral Model

The polyhedral model [FL11] offers a formal approach to reason about programs, by representing program parts in a compact mathematical abstraction. The program parts representable in this model are called *static control parts* (SCoP) which are connected single-entry-single-exit subgraphs of the control flow graph. Static control requires that the only control constructs in the subgraphs are loops with affine loop bounds, and all memory accesses are either to scalar variables or to arrays with affine subscripts in the loop counters. Figure 2.7 shows a simple 2D matrix multiplication code based on the example by Verdoolaege et al. [Ver+13]. This loop nest exhibits static control: all array access index expressions in the statements `S1` and `S2` are affine in the loop bounds, and the loop bounds are integer *parametric constants*, and thus also affine. Parametric constants are loop invariant program variables. Although the loop upper bound N in the example is a program variable, it is considered constant with respect to the program part because it is invariant within it.

```
  // Compute N x N matrix product A = B * C
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
S1:   A[i][j] = 0
      for (int k = 0; k < N; k++)
S2:     A[i][j] += B[i][k] *  C[k][j]
    }
```

Figure 2.7: Matrix multiplication: A three-dimensional loop nest with static control (based on the example by Verdoolaege et al. [Ver+13]). The memory accesses in the statements `S1` and `S2` are all affine.

2.2.3.1 Modeling Affine Loop Nests

The core of the mathematical representation of affine loop nests are *Presburger sets* and *relations* [PW94; Ver16]. A Presburger set $\left\{N(i) \mid cons(i,p), i \in \mathbb{Z}^n, p \in \mathbb{Z}^k\right\}$ is a vector space over $\mathbb{Z}$. Here, $p \in \mathbb{Z}^k$ is a vector of parametric constants. N is an optional name, included to aid readability. The constraints *cons* are Presburger formulas, which are first-order formulas over integer inequalities of quasi-affine expressions. Quasi-affine expressions are sums and differences between variables, parametric constants, integer constants, and quasi-affine products, divisions, or modulo operations. Products are quasi-affine if one operand is constant, divisions and modulo operations are quasi-affine if the divisor is constant. A *Presburger relation* $\left\{N(i) \to M(j) \mid cons(i,j,p), i \in \mathbb{Z}^n, j \in \mathbb{Z}^m, p \in \mathbb{Z}^k\right\}$ is defined similarly.

Using these definitions, we can now model the SCoP of the example in Figure 2.7 (cf. [Ver+13]). The *iteration space* or *iteration domain* is defined by

$$D = \{\texttt{S1}(i,j) \mid 0 \leq i,j < N, \texttt{S2}(i,j,k) \mid 0 \leq i,j,k < N\}\,.$$

The *read access relation* is

$$R = \{\texttt{S2}(i,j,k) \to \texttt{A}(i,j), \texttt{S2}(i,j,k) \to \texttt{B}(i,k), \texttt{S2}(i,j,k) \to \texttt{C}(k,j)\}\,.$$

and the *write access relation* is

$$W = \{\texttt{S1}(i,j) \to \texttt{A}(i,j), \texttt{S2}(i,j,k) \to \texttt{A}(i,j)\}\,.$$

Lastly, a *schedule* defines an execution order of statements preserving the data dependences. Schedules are a relation, mapping statement instances to virtual timestamps, which are executed in lexicographic order. The schedule for the example is

$$S = \{\texttt{S1}(i,j) \to (i,j,0,0), \texttt{S2}(i,j,k) \to (i,j,1,k)\}\,.$$

An analysis that the model facilitates is dependence analysis. Using the techniques by Feautrier [Fea91], we obtain

$$\Delta = \{\texttt{S1}(i,j) \to \texttt{S2}(i,j,0), \texttt{S2}(i,j,k) \to \texttt{S2}(i,j,k+1)\}\,.$$

This dependence relation expresses that in every iteration (i,j) of the outer two loops, the statement `S2` in iteration $k = 0$ of the inner loop depends on the statement `S1`, and in iteration $k+1$ on result of itself from the iteration k.

Another interesting analysis the polyhedral model offers is counting. Because the model is a bounded integer polyhedron, we can use Barvinok's algorithm [Bar08], which counts the number of points in integer polyhedra in polynomial time. The number of points, which in general is parametric in the parametric constants, corresponds to the exact number of instructions executed by the SCoP. From this, we can make accurate predictions of the dynamic runtime of the SCoP, given that accurate estimates for the runtime of the individual instructions are known. Restricting the model to specific instructions further allows additional analyses, such as computing the exact number of bytes read or written in the SCoP.

While the polyhedral model offers a compact and highly versatile abstraction it is also subject to severe limitations. Modeling a loop nest requires the loop to be (part of) a SCoP. If the task is only parallelization, for example, this restriction is stronger than necessary: For instance, an inner loop with non-affine or even unknown bounds does not prevent outer loop parallelization if the inner loop does not access memory, or only memory that is provably not accessed by other outer loop iterations. Further, to date the polyhedral model is applicable only intraprocedurally. Interestingly however, there are techniques to create optimistic models such as those developed by Doerfert et al. [DGH17], which build the model and collect assumptions along the way. An example assumption is the non-aliasing of arrays. The assumptions are then simplified and compiled into runtime checks which only execute transformed code if the assumptions hold.

2.2.3.2 Polyhedral Scheduling

The power of the polyhedral model is that it allows for applying sequences of loop transformations purely on the abstraction, and subsequently generating new code from the transformed representation. Transforming the model means finding alternative schedules that improve performance while preserving data dependencies. This process is called *scheduling*. Scheduling can increase locality, or expose parallelism. For example, minimizing dependence distance improves locality: The more recent an address was accessed, the more likely it is still in the cache. A dependence distance of 0 even means that accesses happen within the same iteration. Such accesses do not cause loop-carried dependencies, and thus do not inhibit parallelization.

For the matrix multiplication in Figure 2.7, the dependence distances are the differences of the timestamps of statement instances involved in the dependence relations Δ. For the schedule $\mathcal{S}$, the dependence distances are $\mathcal{S}_{\mathtt{S2}}(i,j,0)-\mathcal{S}_{\mathtt{S1}}(i,j) = (0,0,1,0)$ and $\mathcal{S}_{S2}(i,j,k+1) - \mathcal{S}_{S2}(i,j,k) = (0,0,0,1)$. In both cases, the entries for the i and j dimensions are zero, indicating that the i and j loops are parallel. Furthermore, because all entries are non-negative, this means that the loops in the example are freely *permutable* without affecting the program semantics [Aho+07, pp. 864]. Permuting loops (sometimes also called loop interchange) exchanges inner with outer loops. More importantly, though, permutability guarantees that loop tiling is legal. Loop tiling, also called blocking in the literature, is a transformation that decomposes a single loop into two: The inner loop, called the point loop, iterates over the elements of a tile (or block) of elements of the original loop. The outer loop iterates over the equally sized tiles of original elements. The tiling operation is able to increase locality and expose parallelism. We refer to Aho et al. [Aho+07] for more in depth introduction.

One widely used algorithm for tiling loops and exposing parallelism is the Pluto algorithm [Bon+08]. Using integer linear programming, it produces a tiling schedule which minimizes the dependence distance. The integer linear program (ILP) is formulated from scheduling functions constructed successively to produce non-negative dependence distances. Loop nests can of course not generally be scheduled to be fully permutable, which Pluto solves by creating a chain of permutable loops. Each set of permutable loops within this chain is then called a *permutable band* or *tilable band*. The linear program then optimizes the schedule function coefficients to minimize dependence distance.

Another frequently cited scheduling approach is PPCG [Ver+13], which extends the Pluto algorithm to apply it to *GPU mapping*. Mapping is the process of assigning loops to GPU thread grid and block dimensions. Instead of a chain of bands, PPCG builds on top of `isl` [Ver10], which computes a tree of bands. The PPCG mapping algorithm then works as follows. First, it finds the outermost bands (i.e., highest with respect to the tree) containing parallel loops, for each of which it will create a separate GPU kernel. Every such outermost band is then tiled. Two parallel outermost tile loops are mapped to the blocks of the grid, three parallel outermost point loops are mapped to the threads of a thread block. To maximize locality, the thread mapping occurs inner-dimension first. That means, the outermost thread block dimension is assigned the iterations of the innermost loop.

The concept of a tree of bands has since been generalized to the notion of *schedule trees* [GVC15]. Grosser et al. implemented this scheduling method in `isl`. The trees are also used in recent versions of the PPCG mapper. Schedule trees provide a way to represent piecewise schedules and modulo arithmetic, which greatly simplifies polyhedral code generation and circumvents limitations of previous code generators. The tree is composed of eleven types of nodes. The most important ones will be introduced here:

Domain The domain node is the root of the tree and introduces the scheduled statement instances.

Context A context node introduces parametric constants not part of the domain.

Filter Filter nodes select subsets of statement instances that have been introduced and not filtered out by previous Filter nodes. Filters are the only legal children of set and sequence nodes.

Sequence A sequence node schedules its filter children in sequence.

Set A set node schedules its filter children in arbitrary order.

Band The band node represents a partial schedule.

Figure 2.8 shows the schedule tree for the matrix multiplication example. Deriving the schedule $S = \{\texttt{S1}(i,j) \to (i,j,0,0), \texttt{S2}(i,j,k) \to (i,j,1,k)\}$ from this is straightforward: The outermost `band` maps the i and j dimensions of both statements to i and j. The `sequence` node schedules the statements selected by its `filters` in sequence. The simplest such schedule thus maps `S1` to a constant 0, `S2` to a constant 1. The innermost `band` lastly maps the k dimension of `S2` to k.

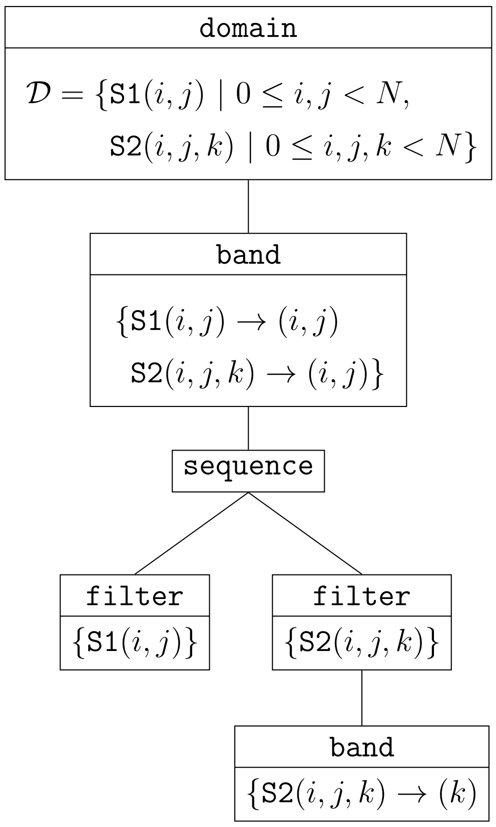

domain
$\mathcal{D} = \{\texttt{S1}(i,j) \mid 0 \leq i,j < N,$
$\texttt{S2}(i,j,k) \mid 0 \leq i,j,k < N\}$
band
$\{\texttt{S1}(i,j) \to (i,j)$
$\texttt{S2}(i,j,k) \to (i,j)\}$
sequence
filter
$\{\texttt{S1}(i,j)\}$
filter
$\{\texttt{S2}(i,j,k)\}$
band
$\{\texttt{S2}(i,j,k) \to (k)$

Figure 2.8: Original schedule tree of the matrix multiplication example in Figure 2.7.

Chapter 3

An Overview of the APHES Framework

The APHES framework is the contribution of this thesis to satisfy the requirements carved out in Section 1.1. To fulfil the thesis goal it offers a compiler and runtime system to parallelize sequential applications for heterogeneous platforms and to autotune the execution. Autotuning operates by using either empirical search to explore new configurations or prediction to provide configurations for known and unknown context states. The decision between either configuring method is made upon every change in the dynamic context.

In this chapter, we provide a high-level overview of the framework and its components, and outline how it achieves the thesis goal. The two main components are the libtuning autotuner and the aphes compiler. In the following, we first present the autotuning library and show how application developers may use it for online optimization of their programs. Second, we present the compiler and outline how it integrates the tuner with the application as part of its runtime system.

3.1 The libtuning Autotuner

The main part of the framework is the libtuning autotuner. We use it within the APHES framework to optimize the cooperative execution, but it is in fact a standalone tool. Application developers can use it as a C++ library to automate the optimization of tunable parameters in an application independent of the APHES framework. The library implements a versatile, generic, and easily extensible online autotuner.

```

tuning::Tuner OMPTuner;
int NumThreads = 8; // Initialize to some default.

vector<int> add(vector<int> A, vector<int> B) {
  vector<int> Result(A.size());

  // Update NumThreads with the next configuration.
  OMPTuner.start();

#pragma omp parallel for num_threads(NumThreads)
  for (size_t I = 0; I < Result.size(); ++I)
    Result[I] = A[I] + B[I]

  // Report feedback. The API measures time automatically.
  OMPTuner.stop();

  return Result;
}

int main() {
  OMPTuner.addParameter(&NumThreads, /*min=*/1, /*max=*/16);

  // Call add(...) repeatedly
}
```

Figure 3.1: Example: Tuning OpenMP vector addition with `libtuning`.

An application interacts with autotuner iteratively: For a previously registered set of application parameters, the application requests a new configuration for these parameters, and reports a performance metric for this configuration back to the tuner.

Integrating an application with `libtuning` thus requires two steps: Exposing and registering tunable parameters, and introducing update and feedback calls. A tunable application parameter can be of any type and have arbitrary semantics, as long as it is freely configurable by the autotuner. Updating the application parameters with the next configuration and reporting performance feedback about the configuration back to the tuner encompasses the tuning kernel of the application. The update-sample-feedback sequence thus is the body of the tuning loop. Most of the time, the feedback is a runtime measure, although this is not required. The autotuner attempts to find the parameter configuration that minimizes the metric, independent of the metric's meaning.

Figure 3.1 depicts an OpenMP example using `libtuning` to optimize the number of threads. The `NumThreads` parameter is registered by reference with the tuner, using a range of valid values between 1 and 16. When calling `OMPTuner.start()` in line nine in every call to `add(...)`, this automatically updates `NumThreads` with a new sample value from this range. Reading from the variable thus uses the new configuration. Subsequently calling `OMPTuner.stop()` then takes care of reporting runtime feedback to the tuner, using the wall clock time passed since the last call to `start()`. The `start()`/`stop()` API is provided for convenience. There are alternative APIs for using arbitrary feedback values. Note that this example is not simplified and shows the actual sequence of calls required to set up and perform autotuning, although excluding the tuning loop.

The `libtuning` autotuner is both generic and extensible. Its architecture is shown in Figure 3.2. The behavior of the tuning process is entirely customizable: The search algorithms and online learning mechanisms can be altered or replaced by application-supplied plugins. To be fully generic, the tuner imposes no restrictions on the application parameters' types, ranges, or semantics. At its core, `libtuning` provides an abstraction between the application and the search engine. The search engine is a driver for arbitrary search algorithms such as the Nelder-Mead algorithm, ϵ-Greedy search, or genetic algorithms. Application developers pick an algorithm from a small selection of built-in searches or implement their own. Searches are isolated from the application through an adaptation layer, which

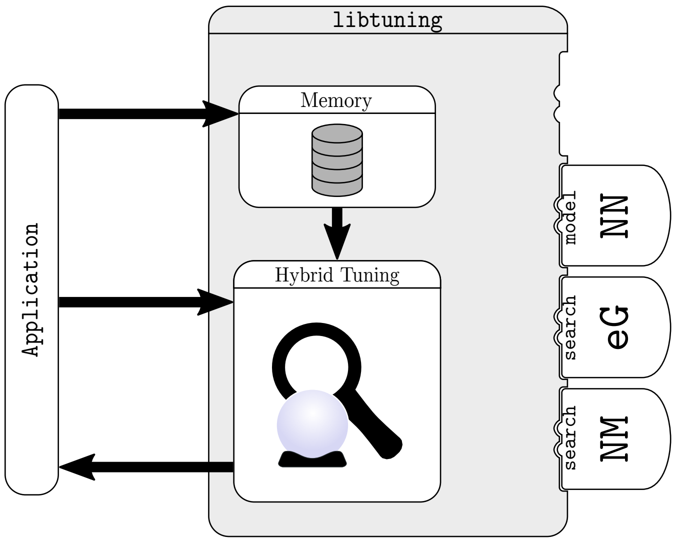


Figure 3.2: The architecture of the `libtuning` autotuner: Application feedback is the input to both a tabular memory keeping tracking best known configurations and to the hybrid tuner. The hybrid tuner combines model-based prediction and empirical search, both of which are implemented to extensible plugins. Included by default are Nelder-Mead (NN) and ϵ-Greedy(eG) search, as well as nearest-neighbor (NN) prediction.

maps between unconstrained real-valued search spaces and the often constrained and arbitrary-valued application parameter spaces. The library provides sensible defaults for this mapping for the most common parameter types: Bounded or unbounded integer or floating point values, as well as non-numeric values describing a set of choices. The mapping can be modified by the application developer.

The features outlined so far make `libtuning` applicable to most tuning scenarios. Besides its usability, `libtuning`'s key features are mechanisms to improve amortization. The autotuner reduces both the total number of configurations tried and the number of extraordinarily bad configurations tried. It achieves this through three methods:

Hybrid tuning: The performance of the tuning kernel generally not only depends on the tuning parameter configuration, but also on the system and application state. To exploit that fact, `libtuning` implements a hybrid autotuning method, combining search, memorization, and model-based prediction. An application may register arbitrary "indicators" with the tuner which characterize the dynamic state of the application and the system. An example is an abstract "input size". Using online learning, `libtuning` automatically detects significant changes in the dynamic state and reacts to the change: Based on a predictor built through online learning, hybrid tuning can immediately switch to a learned configuration. If the exact same dynamic state was observed in the past, the memorized search result for that state is used. Otherwise, a trained model is queried for a predictably good configuration. The application is not required to provide training samples. The predictor is automatically trained for all observed indicator states from training samples produced during search.

Parameter constraints: Special application parameters often induce constraints on large portions of the parameter space. One example is a parameter controlling the choice of an algorithm. Any tuning parameters contained within one of these algorithms should only be updated if this specific algorithm has been selected. The algorithmic choice parameter thus controls the *relevance* of the algorithm's parameters. On the other hand, the parameters controlled by the choice parameter are not independent from those that are not, in general. They cannot be tuned separately. If such relevance constraints are marked by the application developer, `libtuning` constructs hierarchical

```

tuning::Tuner OMPTuner;
int NumThreads = 8;
// Control parallel execution:
bool Parallel = true;

vector<int> add(vector<int> A, vector<int> B) {
  vector<int> Result(A.size());

  OMPTuner.start();

  // Only run in parallel if tuning parameter is set:
#pragma omp parallel for num_threads(NumThreads) \
                         if (Parallel)
  for (size_t I = 0; I < Result.size(); ++I)
    Result[I] = A[I] + B[I]

  OMPTuner.stop();

  return Result;
}

int main() {
  auto N = OMPTuner.addParameter(&NumThreads, 1, 16);
  // Parallel is a nominal parameter:
  auto P = OMPTuner.addParameter(&Parallel, {false, true});
  // Only tune NumThreads if Parallel is true:
  OMPTuner.recordConstraint(N, P, true);

  // Call add(...) repeatedly
}
```

Figure 3.3: Example: Record dependences between parameters.

search spaces to decrease dimensionality as much as possible while preserving dependencies. This additionally allows using different search algorithms for different parts of the search space. Figure 3.3 shows an example of this feature. Introducing a `Parallel` parameter allows to also execute the addition sequentially, in which case the `NumThreads` parameter becomes irrelevant. Registering this dependency with the tuner allows it to exploit that fact.

Analytical feedback: In many practical applications, the application developer has domain knowledge of parameter configurations and can predict whether or not a configuration is reasonable. In the example of the `aphes` compiler, several such predictions are possible. For instance, assigning two iterations of a parallel loop to a GPU is essentially guaranteed to result in non-optimal performance. To account for this, applications may reject configurations without measuring them. This way, `libtuning` can decrease the number of sampled bad configurations to improve amortization time.

To motivate hybrid tuning, reconsider the OpenMP vector addition example in Figure 3.1. The time measurements taken for the execution time of the `for`-loop tuning kernel depend not only on the `NumThreads` parameter, but also on the vector size. As the example stands, the `NumThreads` variable is the only thing the tuner can observe and manipulate. To also make the tuner consider the loop's iteration count, we extend the example as shown in Figure 3.4. Now explicitly using the hybrid autotuner, we add an indicator in line 25 to capture the input size, which governs the loop's iteration count. After registering the indicator with the tuner, the tuner automatically discriminates different input sizes. Configurations are searched per input size, and once converged will be used when a known input size is passed. Although in the example we differentiate between inputs that differ only in a single element, `libtuning` allows for arbitrary resolutions.

The simplest method to map indicators to configurations is a table. In fact, that is what the tuner in the example is using. It is obvious though that a tabular approach is limited: There can be an arbitrary number of different inputs, so the table would reach an impractical size. If inputs are similar, it might not be necessary to distinguish between them. For example, a configuration of the thread count for 2,000 elements will likely be good for 2,001 elements. Additionally, indicators might reflect continuous quantities, which need to be discretized. An application developer can solve both cases by choosing the appropriate resolution,

```

tuning::HybridTuner OMPTuner;
tuning::Indicator<size_t> InputSize;
int NumThreads = 8;

vector<int> add(vector<int> A, vector<int> B) {
  vector<int> Result(A.size());
  InputSize = Result.size(); // Update the indicator state

  // Update NumThreads with the next configuration.
  OMPTuner.start();

#pragma omp parallel for num_threads(NumThreads)
  for (size_t I = 0; I < Result.size(); ++I)
    Result[I] = A[I] + B[I]

  // Report feedback. This API measures time automatically.
  OMPTuner.stop();

  return Result;
}

int main() {
  OMPTuner.addParameter(NumThreads, /*min=*/1, /*max=*/16);
  OMPTuner.addIndicator(InputSize); // Size is set in add()

  // Call add(...) repeatedly
}
```

Figure 3.4: Hybrid, input-sensitive tuning of OpenMP vector addition with `libtuning`.

but what is a good resolution? To relieve the developer of that task, we borrow from the field of machine learning, using online learning to construct a model of the indicator-configuration mapping on the fly. However, we will defer an in-depth discussion of this approach and other `libtuning` concepts, features, and design decisions until Chapter 5.

3.2 Autotuning with `libtuning`

For application developers, the main entry point into the library is the `Tuner`, which functions as a controller for both the search algorithms and the search space. It provides the API which drives the tuning process. Essentially, this API in its base form consists of two methods, `next()` and `feedback(double)`. The former configures all the parameters with the configuration the search algorithms have requested as the next sample. The `feedback` method passes an application-specific feedback measurement to the search algorithms. The search algorithms immediately return the configuration to be sampled next, but the tuner only applies it at the next call to `next()`. By decoupling the parameter update from the search's request, applications are able to modify the parameters without confusing the search: If the parameters were updated upon a call to `feedback`, any modification made by the application after the update would carry over into the next call to `next` and taint the measurement passed to the tuner.

On top of this basic API, `libtuning` provides a set of *decorators* that extend its core functionality. Application developers may thus compose decorators to create an autotuner that best fits their needs. In the following, we provide an overview of the three most important decorators. They are by default enabled in `libtuning`'s simplest tuner type, which was shown in the example in Figure 3.1.

Stopwatch Most of the time, applications intend to tune the runtime of some hot part of the code. The Stopwatch decorator implements the time measurement functionality, and exposes it through an alternative `start()`/`stop()` API, which is implemented in terms of `next()` and `feedback()`. The API measures the time span between calls to the two functions and passes that to the `feedback()` function.

NestedTuners In general, applications may wish to tune the implementation at different granularities. For example, an implementation of a specific matrix multiplication algorithm can integrate an autotuner optimizing the parameters of this algorithm. An application using this algorithm can itself contain another tuner optimizing all the algorithms' parameters plus, for instance, the algorithmic choice. To avoid running these tuners concurrently, the NestedTuners decorator allows developers to aggregate tuners, so that at any point in time at most one of them may run. This is particularly important to the `aphes` compiler, which might parallelize both a function and its (indirect) caller. To prevent confusing the effects of parameter configurations, tuning both functions must be mutually exclusive.

Stabilization The measurements observed during tuning might be noisy. This is especially the case when optimizing runtime. To account for noisy measurements, the Stabilizing decorator reduces the effect of noise by repeating configurations and aggregating measurements. Application developers may request a relative error margin for a fixed 95% confidence interval. The decorator then repeats configurations until that error margin (or a maximum sample count) is reached.

3.3 Autotuning and Automatic Parallelization with APHES

Within the APHES framework, whose overall architecture is illustrated in Figure 3.5, we use `libtuning` as part of the runtime system, which is one of the two integral components of the framework. The other component is the `aphes` compiler, which parallelizes sequential applications and integrates them with the runtime system. Using APHES for an application requires no interaction from the application developer. A successfully parallelized application will automatically execute on the targeted platforms. The runtime system, which has been integrated into the application by the compiler, uses the autotuner to configure the work distribution across platforms and platform-specific parameters.

The compile-time component, the `aphes` compiler, analyzes and transforms sequential source programs. It performs two primary tasks. First, it searches for parallelizable loops, which we call the *source regions* of the program. The compiler then transforms the source regions into per-platform *(parallel) target regions.*

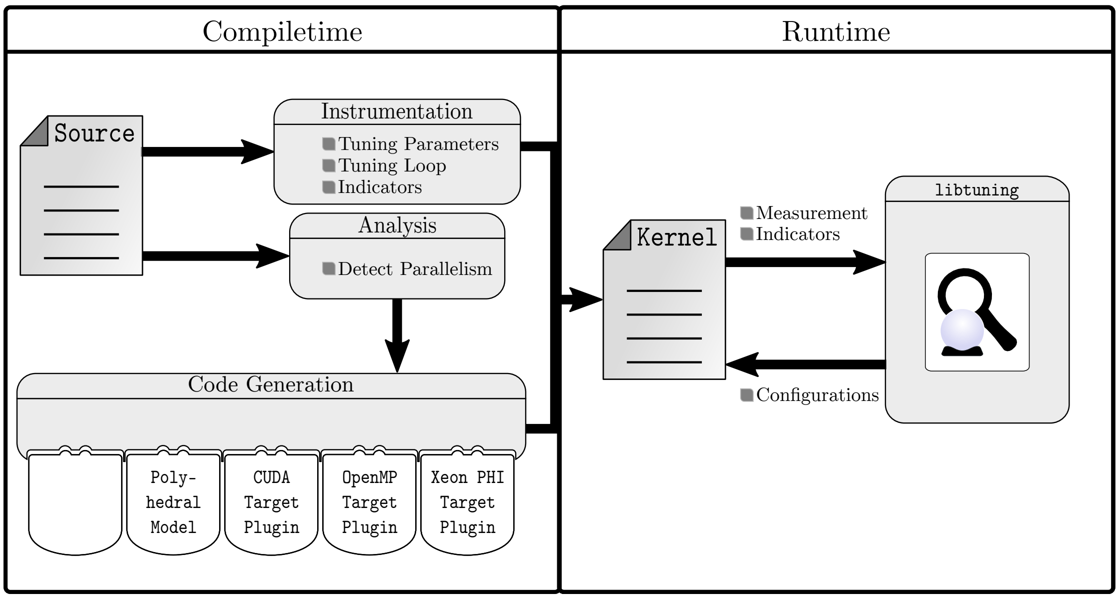


Figure 3.5: The APHES Framework Architecture: The framework is composed of a compiler and a runtime component. The compiler analyzes the program, transforms it for work sharing among multiple platforms, and instruments it to interact with the runtime system. The platform-specific transformations are implemented by an extensible collection of plugins. The runtime system provides platform management and interfaces with the libtuning autotuner.

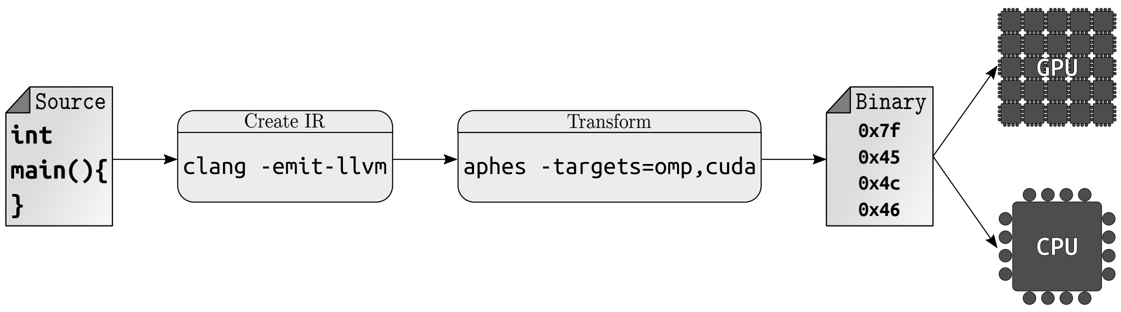


Figure 3.6: Program parallelization with the `aphes` compiler.

To identify the parallel source regions, `aphes` uses both the polyhedral model and classical dependence testing. Target regions are produced either based on the polyhedral model, if applicable, or using ad hoc transformations. The second task of the `aphes` compiler is to instrument the transformed program to interoperate with the runtime component. Instrumentation includes introducing tuning parameters to control the work distribution between platforms as well as platform specific parameters, such as thread counts.

The runtime component can be further subdivided into two main parts. The first part is a small runtime library tightly coupled with the `aphes` compiler. Its main purpose is bookkeeping and execution management, handling device memory allocations, data transfers between host and target platforms, and most importantly the interaction with the autotuner. The `libtuning` autotuner is the second part of the runtime component.

The `aphes` compiler itself is implemented as a plugin system. Compiler developers can add new target platforms as additional plugins. Code generation for new platforms then interoperates with existing platform code generators automatically, and work sharing will happen without additional effort by the developer.

When parallelizing a program, the compiler first runs program analyses to determine parallelizable loops. These loops are then passed to the target plugins for them to generate platform-specific code. The compiler supplies the plugins with all necessary dependence information and takes care of the work distribution. A plugin developer can thus focus on implementing the platform specific parts. Additionally, the compiler maintains a registry of tuner instances, one instance per parallelizable loop. Plugins can register arbitrary tuning parameters with the tuner responsible for the loop currently being transformed. Parameter dependency constraints are automatically generated.

All of the inner workings of the parallelization and optimization are entirely transparent to the application developer, to whom the end-to-end process of parallelization looks like shown in Figure 3.6. Running for example the `clang` compiler on an input program in a supported programming language emits the LLVM intermediate representation of that program. Then, running the `aphes` compiler on the generated IR produces a complete binary, automatically linked with all required libraries. That includes the `APHES` runtime library and `libtuning`, but also any target specific libraries, such as the OpenMP and CUDA runtimes. The compiler currently does not take care of translating the high-level frontend language into the LLVM representation. The application developer thus has to use a frontend compiler for the high-level language for this task.

We discuss further details of the compiler, its analyses and existing transformation plugins in Chapter 6.

3.4 Summary

In this chapter we presented an overview of the `APHES` framework and its components. We introduced the `libtuning` autotuner that is used within `APHES` to optimize platform specific tunable parameters the allocation of work to the platforms. Using a hybrid tuning approach combining empirical search, memorization and online learning, it offers an efficient and effective way to optimize program configurations while being sensitive to the varying tuning contexts at the same time.

Second, we introduced the `aphes` compiler, which searches for parallelism in input programs and parallelizes them for multiple platforms. Without any application developer involvement, the compiler transforms the program to enable automatic work sharing between heterogeneous platforms and to expose tunable parameters. To optimize the program at runtime, it is instrumented to interoperate with a runtime library which in turn interfaces with the `libtuning` autotuner for optimization.

Chapter 4

Related Work

This thesis presents a compiler that automatically parallelizes sequential applications. Transformed applications are instrumented to adapt to the actual system and inputs on which they are executed and react to changes therein. This dynamism is achieved through empirical online autotuning and machine learning methods.

To put these contributions into context with prior art, we look at them from two angles, which are parallelizing compilers for heterogeneous systems and autotuning. In the field of autotuning, we must further differentiate between general purpose autotuning systems, machine learning and model-based techniques, and finally the intersection of compilers and tuning in the form of tunable parallel programming languages and parallelizing compilers.

Since each of the fields on which this thesis touches has a rich history dating back at least two decades, the overview we give in this chapter cannot be complete. We thus focus on the most important or most closely related publications and cover the broad spectrum of the techniques. We will first look at autotuning in Section 4.1. In particular we discuss general purpose autotuners, both based on classical empirical search, and on machine learning and explicit performance models. In Section 4.2 we examine recent advances in automatic parallelization for heterogeneous systems. Lastly, in Section 4.3 we introduce approaches that combine parallelization and tuning through compilers and languages, either by tuning the compilation process itself or by adjoining the compiled program with a tuning system.

We close each section of this chapter by summarizing the individual features of the presented articles and comparing them to the features of `APHES` and `libtuning`. We focus on features that contribute to the goal of tuning automatic cooperative multiplatform execution, which we briefly introduce in the following. The first set of features refers to the automatic parallelization of programs. It describes whether the compiler targets *data* or *task parallelism* and analyzes the program *statically* or *dynamically*, whether the parallelized program executes *heterogeneously* targeting a single platform other than the CPU, targeting *multiple platforms* at the same time, or *cooperatively* on multiple platforms. To be considered cooperative, the heterogeneous multi-platform execution must use all platforms simultaneously, each working on a part of the problem, and then merge the partial results. If a parallelized program executes on multiple platforms but not cooperatively, this means that only a single platform executes the parallelized region while other platforms are waiting. Such a program is not using the full capacity of the system. Secondly, approaches that somehow optimize the generated program (or the hand-written application in the context of the general purpose autotuners in Section 4.1) can do so either *offline* or *online*. Offline optimization refers to the compiler or autotuner optimizing the program given sample inputs before any production runs. Online optimization in turn happens during a production run on inputs observed in production, and can also occur within runtime systems or JIT compilation scenarios. We additionally emphasize whether the tuning is *search-based*, *model-based*, or using simple heuristics. If it is model-based, we distinguish between a *learned model* or one that has been hand-implemented by either the application developer or the compiler developer. Moreover, the model inputs can either be *automatically extracted program features* or can be supplied by an application developer. Lastly, we consider whether an approach is *input sensitive*, i.e., whether it is able to react appropriately to varying inputs. Input sensitive approaches are required to react to new inputs at runtime (i.e., online), but the decision making might be trained offline on a-priori sample data.

The feature summaries following each section are in tabular form. To ease readability the tables always include all features listed above. Because each of the following sections focuses on works dedicated to particular aspects of autotuning or automatic parallelization, not all features apply in general. In that case, the irrelevant features are marked in gray.

4.1 Autotuning

In this section, we discuss prior art related to the goals of this thesis in the field of autotuning. This will cover two major areas:

1. General purpose autotuning approaches and their applications (Section 4.1.1).
2. Machine learning and performance models to adapt an application to new hardware or new inputs (Section 4.1.2).

4.1.1 Tuning Algorithms and Autotuners

At the turn of the century, the high-performance computing community began to realize that hand-optimizing the same algorithms over and over for every new upcoming system and accelerator hardware as well as different types of inputs was not sustainable. Consequently, the interest in automating this tedious task was spawned.

In 1995, Brewer was the first to address selection of algorithms and data layouts at runtime [Bre95]. His system composes a library of algorithm implementations with an autotuner, to automatically select an algorithm and optimize algorithm parameters for given inputs. The autotuner generates generalized least square models to predict execution time from a set of input features, such as problem size. This system achieves portability across platforms through profiling calibration inputs automatically generated from the algorithms and parameter ranges. Hence, developer effort is limited to the burden of implementing the different variants of the algorithms. Brewer reports a success rate of beyond 99% for this prediction on four different systems. Although his work is not the first to explore performance prediction, it can be considered the first applicable to online tuning. Considering the domain of offline tuning, however, it is predated by Alan Sussman's papers and doctoral thesis, whose parallelizing compiler will be discussed in Section 4.3.1.

The ATLAS system by Whaley and Dongarra [WD98] is a well-known approach combining a BLAS library with autotuning capabilities. During installation, the library automatically self-optimizes loop structures, tiling, cache reuse, and parallelism for the system to which it is being deployed. The tuning is offline, during installation, and samples a number of benchmark programs shipped with the library. The optimization itself is based on a hand-crafted model, whose parameters are configured offline during optimization, and which can then pick the fastest linear algebra algorithm implementation for a given input.

In his doctoral thesis and subsequent research, Richard Vuduc made major contributions to the field of autotuning as a whole and to tuning BLAS algorithms in particular [VD00; VDB01; Vud03]. In 2001, he was the first to advocate empirical search in the space of possible implementations [VDB01]. Using the PHiPAC system (which we discuss here in Section 4.3.1), he demonstrates that approximative global optimization can produce configurations that are within 5% of the optimal performance. Approximation is implemented by stopping an exhaustive search early based on a statistical model. To account for the fact that different configurations are optimal given different inputs, the system constructs run-time selection rules using Brewer's linear regression approach as well as Support Vector Machines.

The year 2002 marks the publication of ActiveHarmony by Ţăpuş et al. [ŢCH02], which is to date one of the most prominent general purpose autotuners. It provides a sophisticated distributed system that integrates with arbitrary client libraries or programs. The performance monitor that observes execution time in the client application relays information to a central server, which responds with tuning decisions. In the central Harmony server, the system employs the Nelder-Mead downhill simplex algorithm.

In addition to ActiveHarmony, the Nelder-Mead algorithm is also applied to ATLAS. You et al. [YSD05] make several modifications to the original algorithm. These are necessary because autotuning differs in several details from the numerical optimization problems it was designed for. In practice, tuning parameters are often integer valued and bounded. Besides the bounds, more complex restrictions may arise from parameter semantics or dependencies between parameters, leading to some possible configurations becoming invalid. The modified algorithm accounts for these through stationary penalty values (i.e., setting the function value for invalid configurations to infinity). The implementation of the algorithm in this thesis is closely based upon You et al., albeit choosing a different strategy to deal with boundaries and invalid points.

With AtuneIL, Schaefer et al. [SPT09] do not develop an autotuner or tuning algorithm, but an annotation language to integrate an autotuner with a C#, C++, or Java program. By inserting pragma statements into an existing program, developers define parameters, parameter types, and domains, as well as time measuring points for the tuner to sample. Additionally, the annotations may contain constraints encoding dependencies between parameters. Using constraints, AtuneIL can drastically reduce the search space that needs to be explored. In a case study on two applications, Schaefer uses AtuneIL as part of the Atune framework [Sch09] and evaluates search-based autotuning using hill climbing and random sampling. Speedups of three and seven are reported, while the search time for hill climbing was reduced by over 50% due to AtuneIL's handling of constraints.

With Perpetuum, Karcher and Pankratius [KP11] introduced an always-on online tuner that is integrated into the Linux kernel. Through system calls, client applications interface with the tuner, registering application level parameters which Perpetuum manipulates directly. Like ActiveHarmony, Perpetuum uses the Nelder-Mead search algorithm.

In 2012, Bergstra et al. [BPC12] develop an approach that combines empirical search with model-based tuning. They identify issues that are closely related to those explored in this thesis. Model-based tuning on the one hand allows for fast exploration of the search space, but suffers from inaccuracies caused by incomplete knowledge about runtime-characteristics. Examples are varying inputs and hardware. Empirical tuning on the other hand can evaluate the program with real inputs and real hardware, but the size and dimensionality of the search space render this a time-intensive process. This realization concurs with the motivation for hybrid tuning. As a compromise, Bergstra et al. propose combining a statistically-derived model with empirical search, albeit in a way that is orthogonal to hybrid tuning. The key difference between the two techniques is that hybrid tuning uses empirical search to explore the real search space, whereas Bergstra et al. apply it *on* their model. Evaluating their approach on the GPU implementation of a filterbank correlation kernel demonstrates it to be competitive compared to pure empirical tuning, but at a fraction of the cost.

Another autotuning technique that pursues very similar goals as this thesis is SiblingRivalry [Ans+12]. Targeted towards always-on online autotuning, Ansel et al. devise a scheme to prevent bad configurations to poison the amortization time. By splitting compute resources into two equal partitions, they are able to evaluate two configurations at the same time. In one partition, they always execute the currently best known variant. In the other one, a genetic algorithm samples the search space. Whichever variant completes first wins the race, the other one is immediately terminated. Through this strategy, the effect of selecting bad configurations is never visible at runtime. However, this property comes at the cost of half the execution resources, which ultimately disqualifies this technique from being applicable to the problems investigated in this thesis: we are investigating optimal resource utilization, so consequently diverting parts of the available resources to different tasks contradicts the goal. Moreover, some platforms such as GPUs, are always allocated to exactly one task. On such platforms SiblingRivalry is thus not applicable.

AtuneRT [Til+14] developed by Tillmann et al. is an online autotuner similar to ActiveHarmony and the successor to the tuning framework Atune. Like ActiveHarmony, AtuneRT uses Nelder-Mead search. In the article, the autotuner is applied to optimizing platform-specific parameters for CUDA applications, that is to say, thread block sizes. For one benchmark, also an application-specific parameter is tuned. The authors report speedups of 4-28%.

Currently, the tuner that is most widely used in various applications across the fields is OpenTuner, also developed by Ansel et al. [Ans+14] in 2014. A simple interface and a Python implementation make it easy to integrate with most applications. Its biggest selling point however is that it alleviates the application developer of the need to select a tuning mechanism. In OpenTuner, this hyperparameter is treated as a first class tuning parameter and is controlled by a meta-tuner. Application developers select a set of search- or model-based techniques from a builtin library, or implement their own. The meta-tuner then orchestrates the searches. The AUC Bandit strategy [Pac+12] selects a search technique in every iteration, advancing a global knowledge database of the search space. Because of this meta-technique, OpenTuner does not work well as an online tuner, since it violates one of the requirements we defined in Section 1.1: Sampling continues indefinitely and there is no notion of convergence.

The most recent general offline tuning framework is ATF [RHG17] published in 2017. Like AtuneIL, interaction with the tuner is implemented with the help of a pragma DSL for C and C++ programs. A key feature of ATF is the ability to specify arbitrary constraints on parameters. To enforce these constraints, ATF exhaustively enumerates all points in the search space. While this is an elegant way to avoid holes in the space that conflict with preconditions of many numerical optimization algorithms, it is not applicable to floating point parameters, which cannot be realistically enumerated. The framework is further generic in the sense that it allows for extension with arbitrary user-provided search algorithms. In

Table 4.1: Comparison of the tuning algorithms and autotuners

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
Brewer								✓			✓	✓		
ATLAS									✓		✓	✓		
Vuduc									✓	✓		✓		
ActiveHarmony								✓		✓				
You et al.									✓	✓				
Atune(IL)									✓	✓				
Perpetuum								✓		✓				
Bergstra et al.									✓	✓	✓	✓		
SiblingRivalry								✓		✓				
AtuneRT								✓		✓				
OpenTuner									✓	✓	✓			
ATF									✓	✓				
`libtuning`								✓		✓	✓	✓	✓	

their paper, Rasch et al. provide a simulated annealing implementation as well as an interface with OpenTuner, giving access to its extensive algorithm library and meta-tuner technique.

In Table 4.1 we summarize the central aspects of the approaches presented in this section and compare them to our `libtuning`: Ours is the only technique combining search-based and model-based online tuning, learning the model on-the-fly. Of the other online-autotuners only Brewer combines search and modeling, which allows him to be sensitive to inputs. Unlike our approach, however, Brewer's models is linear and constructed offline, which means it cannot capture non-linear effects and adapt to changing inputs or hardware. The same argument applies for the input sensitive and model-based offline tuners. Although SiblingRivalry is not input sensitive, it pursues a goal similar to ours, since its main focus is to reduce amortization time. The reduction comes, however, at the price of half the compute resources, which requires the compute resources to be partitionable. Both are undesired restrictions. The remaining two online tuners we presented, ActiveHarmony and Perpetuum, both make no attempt to be input sensitive or to optimize amortization time. Note that AtuneIL is missing from this list, because it is not itself an autotuner, but a language to interface between a program and a tuner. Because we are comparing between different autotuners here, we grayed out the properties that are specific to parallelization and compilation here. Automatic parallelization will be covered in Section 4.2.

4.1.2 Machine and Performance Models

Although general autotuners and tuning algorithms are powerful and available, applications often go a different direction to optimize their parameters. Due to the generality, autotuners need to solve a black box optimization problem. Because a white box approach promises more precise tuning, with shorter search times and possibly better results, many applications employ machine or performance models to optimize themselves. The advantage of white box tuning comes at the cost of having to implement the model. The quality of the model further strongly affects the resulting performance. In this section, we examine related work applying white-box tuning in various forms.

One of the first approaches applying performance models to online tuning was MATE [Mor+03; Mor+04]. It provides a dynamic distributed tuning environment for PVM applications. MATE integrates the optimization of messaging parameters (e.g., aggregation and TCP settings), memory allocation, PVM communication options, as well as application parallelism settings such as the number of worker threads. For all of these components, MATE supplies measuring instrumentation (to detect performance problems), performance models (to detect the problem), and tuning options (to correct the problem). Based on these means MATE then adapts the distributed application dynamically.

With StarPU, Augonnet et al. offer a runtime system and API to unify heterogeneous tasking [Aug+09]. For a task implementation (e.g., matrix multiplication), developers explicitly express the task's data dependencies and access behavior. The accessed slice of memory is labeled as read, write, or read-write. The runtime system then takes care of task scheduling onto available platforms according to data and explicit task dependencies and attempts to find an optimal schedule.

It can thus be considered both an online and offline tuning system. Scheduling is guided using predefined policies based on task priorities and per task performance models. Greedy policies implement a first-come-first-serve behavior, where a pending task is assigned to the first ready platform. Similarly, a weighted random policy chooses a platform at random with user-supplied per-platform weights. Lastly, performance model policies predict the runtime of a task per platform using models provided by the developer or automatically generated by StarPU [ATN09].

Another runtime system managing tasks across heterogeneous systems was developed by Kicherer et al. [Kic+12]. On top of having platform specific implementations per task, they also support having *multiple* implementations per platform that differ in features such as floating point precision or vector instruction set. Implementations can also require or favor specific input-set characteristics. For an application, users provide a library of implementations for the different platforms and attributes. At runtime, the management system will then pick the best implementation whose attribute requirements are satisfied. To find the best implementation, Kicherer et al. employ an online learning scheme [KBK11]. Based on user-defined features (e.g., “matrix size”), the classification scheme samples inputs for several feature values. Missing performance values for features and implementations are obtained through linear interpolation, thus assuming a linear relationship between the features and the observed runtime. Finally, the classifier picks the best implementation for observed feature values and determines value ranges for each implementation.

In 2013, Balaprakash et al. [BGW13] proposed surrogate modeling using active learning of dynamic trees to automatically derive performance models from observations. The surrogate model predicts the program runtime for a given configuration. With active learning, their approach is able to refine the performance model iteratively, determining the unevaluated parameter configuration to sample next. Balaprakash et al. chose dynamic trees as their surrogate model, which combine classical regression trees with Bayesian inference. They apply this technique to loop optimizations of numerical kernels, modeling their energy, and optimizing an MPI topology. With only a few hundred samples, they show that their models make the correct prediction over 90% of the time.

Much like the work by Kicherer et al., Nitro [Mur+14] is an autotuning system for picking optimal code variants depending on inputs. Developers define multiple variants of an algorithm and expose real-valued features that classify the input

Table 4.2: Comparison of the model-based tuning approaches

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
MATE									✓		✓	✓		✓
StarPU								✓	✓		✓	✓	✓	
Kicherer et al.								✓			✓	✓	✓	
Balaprakash et al.									✓		✓		✓	
Nitro									✓		✓	✓	✓	
Bao et al.									✓		✓	✓	✓	✓
`libtuning`								✓		✓	✓	✓	✓	

data. For example, for a sparse matrix vector product, such a feature is the average number of non-zero elements per row. Given training inputs provided by the application developer, Nitro builds an SVM model for the inputs' feature vectors and observed runtimes during an offline training phase. At runtime, it will then classify a new input and thus predict the optimal implementation for this particular input.

Bao et al. [Bao+16] take a different view on autotuning. Instead of optimizing performance, they optimize energy efficiency. For a given benchmark, various CPUs achieve maximal efficiency at different frequencies. Using the polyhedral model, the approach derives various static properties from the representation, such as the number of (parametric) loop iterations, the data access behavior, as well as cache utilization and miss rate estimations. For a given target CPU, they compute a profile for benchmarks selected to stress individual features. From this profile, they then construct a decision tree to determine optimal frequency settings at runtime.

In Table 4.2 we summarize the central aspects of the works presented in this section, and compare them to our autotuner: `libtuning` is the only approach combining online search and online learning. Balaprakash et al.'s surrogate models stand out in the comparison because they are the only generic approach. All others are tied to a specific application and thus solve only white-box tuning problems. Note further that while we labeled some approaches as "Offline", the actual adaptation of these applications happens at runtime, as it must because of the input sensitivity. The model, however, is learned a-priori on sample benchmarks. StarPU and Kicherer et al.'s approach are the only two here that are capable of online learning. Both construct a linear regression model at runtime from observations, which is a sensible choice in their respective applications. In `libtuning`, we support more general models and combine them with search to cope with the problem of exploring in large search spaces. We marked the properties that are irrelevant in this comparison because they are specific to parallelization and compilation.

4.2 Automatic Parallelization for Accelerators

With the advent of vector machines and array computers in the 70s, automatic transformation of sequential codes into parallel programs for high performance machines became a goal both in research and practice. In more recent years a new class of parallel machines, namely massively parallel accelerators such as GPUs or FPGAs, has reignited the hunt for automatic tools to exploit these platforms optimally. The increased complexity of these platforms, exacerbated by the fact that today's systems have access to multiple accelerators *at the same time*, has spawned a large new body of research of parallelization tools that target individual accelerators and, most importantly, simultaneous execution on multiple accelerators.

The critical component of the parallelization process is identifying parallelizable program parts. Currently, approaches can be characterized into one of four classes:

1. Automatically based on dependences: Computing data and control dependences between memory accesses, either conservatively or optimistically, yields independent program paths. These paths may be executed in parallel.

2. Using the polyhedral model: Although parallelization using this model is also based on computing data dependencies, it is unique in the way it models control. As a comprehensive algebraic representation of the code it offers a unified framework for dependence analysis and code transformations.

3. With programmer help: Conservative dependence analysis is often too imprecise to produce meaningful parallelizable regions, whereas an optimistic analysis comes with the hazard of introducing bugs. Instead, parallelization hints by the programmer in the form of explicitly parallel DSLs lift this burden from the compiler.

4. Detection of parallel patterns: Where dependence-based parallelization is too inaccurate and parallel DSLs are too costly in terms of programmer time, pattern-based parallelization can provide a middle ground. Acknowledging the fact that programmers tend to build programs from specific building blocks, *parallelizable building blocks* can be detected and transformed into well-known parallel design patterns, e.g., pipeline or master-worker constructs.

In the following, we will examine works from either of these categories with goals related to this thesis. We limit the scope in this section to approaches that manage load balancing between platforms and platform-specific configurations either statically or through simple heuristics. Dynamically adaptive approaches will be discussed in Section 4.3.

4.2.1 Dependence-based Parallelization

Dependence testing is the fundamental operation in parallelizing loops. To decide parallelizability, memory accesses in a loop are checked for loop-carried data dependences. That is, a memory location read or written in one loop iteration must not also be read or written in another. The vast majority of approaches for automatic parallelization rely on this test. In the following, we give an overview over the parallelizers from that class published within the last 25 years. We focus on both heterogeneous as well as classical CPU-only targets and limit the scope to tools making tuning decisions statically based on simple heuristics.

In 1991, Irigoin et al. [IJT91] published the PIPS framework for parallelization using interprocedural dependence analysis. The framework targets data parallelism using static analysis. Although a handful compilers predate it, the main goal of those was vectorization and fine-grained parallelism. PIPS addresses coarse-grained parallelism by transforming Fortran77 loops into vectorized or `DOALL` versions. Based on the findings in 1992 [BE92] on the lacking effectiveness of prevailing auto-vectorizers targeting SMPs, Blume et al. [Blu+96] designed Polaris for Fortran programs. Polaris, similarly to PIPS, targets data parallelism using static analysis. Its key features were privatization of temporary variables and appropriate handling of induction and reduction variables.

The SUIF compiler [Hal+96] was a long-running effort beginning in the mid-90s at Stanford University to provide a compiler intermediate representation and framework. Its authors Hall et al. are likely the first to recognize that with symmetric multiprocessors, cache-oblivious parallelization can actually harm overall performance. SUIF thus explicitly optimizes data layout and accesses to bridge this gap. To make parallelization applicable to programs beyond simple numerical kernels, SUIF performs flow- and context-sensitive interprocedural dependence analysis statically [Hal+05].

Wang et al. [Wan+08] present a fundamentally different approach to dependence-based parallelization. Instead of finding independent iterations of a loop, the authors detect independent sets of instructions *within* loop iterations. To be precise, they compute backwards slices for instructions within time-intensive regions of the program. The backwards slice for an instruction is the subset of instructions from that region that have an effect on this instruction. Individual slices are executed in parallel. Moreover, slices are built speculatively, meaning that data dependences are computed dynamically from a profile, which might miss existing dependences that occur only rarely.

Another approach based on dynamic dependences is by Hammacher et al. [Ham+09], who determine parallelizability optimistically from Java programs traces. Optimistic parallelization bears the risk of producing unsound transformations, leading to bugs. As a consequence, the authors do not automatically transform the program, but instead present parallelization suggestions to the developer, relying on their expertise.

Implemented on top of PIPS, Guelton et al. [GAC12] use convex array regions to compute the extent of the memory region accessed by a parallel or parallelizable loop. This type of information is required to determine data transfers on parallel NUMA machines, most importantly for targeting GPUs for offloading computations. With Par4All [Ami+12], PIPS and convex array region analysis are combined for code generation and offloading to CUDA or OpenCL GPU targets.

DiscoPoP [LJW13] is a tool which goes beyond parallelizing loops. It combines dynamic dependence profiling with an alternative view on data flow. Decomposing the program into read-modify-write sequences of memory locations, the tool obtains a dependence graph of small computational units. If independent, such units can be executed in parallel. Mapping units onto threads provides control over the granularity of the parallelization, and allows parallelizing loops, (sub-) tasks, or entire functions in a uniform fashion. This method can be applied both to data and task parallelism.

In 2014, Sura et al. [SOB14] presented a hardware-software co-design to exploit a form of instruction-level parallelism, automatically mapping sequential innermost loops to multiple cores. They find sequences of instructions independent from each other with respect to memory and control dependences. These are statically dispatched to instruction queues attached to the parallel cores. Because that requires hardware support, performance is evaluated in a simulator.

Streit et al. [Str+12; Str+13] developed a tool called Sambamba, treating task parallelization as an optimization problem. The dependence graph of the source program is mapped onto tasks by formulating the mapping as an integer linear program, a technique the authors named *generalized task parallelism* [Str+15]. A landmark feature of Sambamba is the ability to dispatch tasks adaptively based on heuristics. The runtime system uses a small set of (static) heuristics to decide whether a new task should be spawned, or whether to select the sequential version. The heuristics limit the number and nesting depth of tasks based on the number of cores and the current system load. The sequential version is selected if system load exceeds 90%, if the number of tasks exceeds twice the number of cores, or if the parallel nesting level exceeds the logarithm of the number of cores.

Table 4.3 summarizes the properties of the presented dependence-based parallelization techniques, and compares them to the properties of the `APHES` framework: Except Par4All, all compilers create parallel programs for the CPU platform. While Par4All does target the GPU, it only targets a single platform, and thus does not address cooperative multiplatform execution. Another distinction between the `aphes` compiler and those listed here is that we rely only on static analysis and data parallelism detection. While dynamic program analysis produces more precise results, it is valid only for the analyzed inputs. For us this is insufficient because being able to optimize performance online and for varying inputs is of little relevance when the transformed program is only valid for a known set of inputs. We further do not support task parallelism because GPUs generally offer very little acceleration in that domain, especially since the number of different tasks is constant and often small. In the context of our classification criteria,

Table 4.3: Comparison of the dependence-based parallelization approaches

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
PIPS	✓		✓											
Polaris	✓		✓											
SUIF	✓		✓											
Wang et al.		✓		✓										
Hammacher et al.	✓	✓		✓										
Par4All	✓		✓		✓									
DiscoPoP	✓	✓		✓										
Sura et al.	✓	✓	✓											
Sambamba		✓	✓											
`APHES`	✓		✓		✓	✓	✓	✓		✓	✓	✓	✓	✓

instruction-level parallelism as it is exploited by Sura et al. can be understood as either data- or task-parallel and thus ticks both columns in the table. We discussed parallelization approaches in this section that do not use autotuning to make optimization decisions and have grayed out the properties in which a comparison against APHES is thus impossible. The autotuning features of APHES are shown in the table for completeness.

4.2.2 Parallelization of (Mostly) Affine Programs with the Polyhedral Model

The polyhedral model, also referred to as the polytope model, is a framework for loop representation and optimization that goes back as far as the mid-sixties. The foundations were laid by Karp, Miller, and Winograd [KMW67], who investigated scheduling of computations based on recurrence equations. Lamport [Lam74] was the first to apply it to automatic parallelization in the seventies. In the decades since then, dozens of systems have been developed which perform automatic parallelization with the help of the polyhedral model in some form. Because of this, we will limit ourselves to the most recent advances, and especially those that target heterogeneous systems. The polyhedral model is a static analysis technique and helps to maximize data parallelism. Consequently, all approaches we discuss satisfy the according classification criteria. Probably the first to apply polyhedral optimization to automatic offloading to GPUs are Baskaran et al. [BRS10]. Because their technique relies on a form of autotuning however, their work will be discussed in Section 4.3.2.

In her doctoral thesis, Jimborean [Jim12] developed a framework for speculative parallelization with the polyhedral model. The VMAD ("Virtual Machine for Advanced Dynamic analysis and transformation") system generates multiple OpenMP-parallelized versions of loops at compile time, and patches the running application dynamically if an applicable parallel version is available. Parallel versions are selected with the help of instrumented versions that profile the code.

PPCG by Verdoolaege et al. [Ver+13] is a polyhedral source-to-source compiler which transforms C to CUDA code. It generates GPU code by applying multilevel tiling to account for the levels of parallelism and memory hierarchy of CUDA GPUs. To also parallelize iterative computational kernels that exhibit an outer "time" loop, Grosser et al. extend PPCG with a split tiling mechanism

[Gro+13] and a hexagonal tiling scheme [Gro+14]. PPCG uses variants of the Pluto [Bon+08] algorithm and the CLooG [Bas04] code generator for scheduling and code generation.

Whereas in PPCG the GPU mapping occurs fully automatically, the Loo.py [Klö14] programming system gives the developer control over the polyhedral transformations applied to loops. Embedded in Python, it allows the developer to write computational kernels in an array language, and to apply transformations and inspect the results.

The KernelGen tool [Mik+14] is similar in spirit to PPCG. The key difference between the two is that KernelGen is based on LLVM, thus supporting a more general range of input languages, and that KernelGen attempts to minimize data transfers between host and device. Instead of eagerly transferring all data at every GPU kernel launch, the runtime system provides a segmentation fault handler which performs the transfers just-in-time.

Another work that emphasizes memory transfers and uses the polyhedral model to optimize them is Molly [Kru14a]. Molly is an LLVM-based tool implemented on top of Polly [GGL12], LLVM's polyhedral loop optimizer. The unique goal of this tool is to parallelize lattice QCD simulations on a distributed machine [Kru14b], which requires generating efficient code for the explicit communication between nodes.

Closely related to Molly is also the work by Damschen et al. [Dam+15]. Using a just-in-time compiler that is executed as a server on an Intel Xeon Phi accelerator, a client running an application can dispatch parallelizable loops to this server, which then compiles and runs the code. The parallelization itself is implemented again with the help of Polly, which is used to generate OpenMP code.

In 2016, Polly itself gained the ability to also generate code for CUDA GPUs [GH16]. Under the name Polly-ACC it uses PPCG as a library to perform polyhedral GPU mapping, and then generates CUDA code as well as the necessary data transfers. Tile sizes, GPU parameters, and memory mappings are decided using PPCG's built-in heuristics.

In Table 4.4 we summarize the properties of the polyhedral compilers we presented, and compare them to the `APHES` framework. Although many approaches exist that offload computations to accelerators such as GPUs or Xeon PHIs, none of them target multiple platforms, neither as alternatives nor for cooperative exe-

Table 4.4: Comparison of the polyhedral parallelization approaches

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
Lamport	✓		✓											
Jimborean	✓		✓											
PPCG	✓		✓		✓									
Loo.py	✓		✓		✓									
KernelGen	✓		✓		✓									
Molly	✓		✓											
BAAR	✓		✓		✓									
Polly-ACC	✓		✓		✓									
`APHES`	✓		✓		✓	✓	✓	✓		✓	✓	✓	✓	✓

cution. Again, we mark those properties which do not compare in this section and show the autotuning features of `APHES` for completeness.

4.2.3 Explicitly Parallel DSLs

In this section we discuss prior art in the field of explicitly parallel programming languages. Because this thesis deals with *multiplatform* parallelization, this section is scoped to languages that target multiple platforms simultaneously. Many languages exist that target multiple platforms, but do so explicitly: The developer has to manually choose a target device and program accordingly. Popular languages such as OpenMP and OpenCL belong into this category, where device types and data movements need to be written out. Instead, we focus here on approaches that make data movement and platform selection implicit.

The Merge framework [Lin+08] is a library approach to programming a heterogeneous system based on the `mapreduce` programming pattern. A `mapreduce` program is built from a sequence of map and reduce operations, which are trivially parallelizable. The framework supports execution of the work on a GPU and a CPU simultaneously, heuristically splitting the workload: The application developer implements function variants for multiple platforms and annotate them accordingly. The framework then distributes tasks (called "work units") from a work queue among available platforms for which an implementation for the task is available. Whenever two platforms are available and viable, Merge chooses based on preferences hand-encoded by the application developer.

Although not specifically a DSL itself, Ocelot [Dia+10] transforms an existing DSL, namely CUDA, for execution on both GPUs and CPUs. Unlike in this thesis, however, an Ocelot-transformed program executes only on one platform at a time, and does not distribute the work. Ocelot is a binary translator, compiling PTX code, an assembly format for CUDA programs, back to LLVM for offloading to various platforms. Compilation happens just-in-time, which in principle enables dynamically adapting to inputs and switching between platforms, although the authors do not investigate this in depth.

Ocelot bears similarities to another system, named TwinPeaks [Gum+10]. Instead of targeting PTX, TwinPeaks is based on OpenCL, and compiles OpenCL code targeted to GPUs or CPUs. Although OpenCL is a language built to be target agnostic, optimized programs are often not. This tool hence takes great

Table 4.5: Comparison of the languages and parallelizing compilers

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
Merge					✓	✓	✓							
Ocelot					✓	✓								
TwinPeaks					✓	✓								
OMPSs					✓	✓								
`APHES`	✓		✓		✓	✓	✓	✓		✓	✓	✓	✓	✓

care of optimizing the GPU-targeted program for CPU memory and cache layouts as well as vector units.

OpenMP is a well-known domain specific language that greatly simplifies writing parallel codes by extending sequential code with simple and compact annotations. In 2010, Ferrer et al. proposed the OMPSs programming model [Fer+10], that extends OpenMP with the ability to explicitly offload OpenMP-parallelized regions to accelerators. Originally, OMPSs targeted OpenCL, but has since been incorporated in the OpenMP 4.5 standard and now supports NVidia GPUs as well as accelerators [SKK17].

We summarize the explicitly parallel languages discussed here in Table 4.5. While all presented languages and compilers generate code for multiple platforms, only the Merge framework supports cooperative execution. The execution is, however, limited to task parallelism, which is orthogonal to what we investigate with `APHES`. Merge furthermore determines the distribution of work heuristically based on decisions made by the application developer, whereas we achieve this automatically. Marked in gray are the autotuning properties which are not considered in this section.

4.2.4 Pattern-based Detection of Parallelism

To avoid the inaccuracy of dependence-based parallelism detection and the programmer effort mandated by parallel DSLs, some approaches focus on detecting recurring parallel patterns in sequential programs instead. These approaches are discussed in this section. Although the Patty tool by Molitorisz et al. [MMT15] uses a pattern-based parallelism detector, its discussion is postponed until Section 4.3.2 because of its autotuning facilities.

Rul et al. [RVD07] claim the first attempt to identify certain patterns in sequential programs at a function scope. While existing parallelizers aim to transform loops in mostly numerical programs, Rul et al. operate at the function level. With the help of data dependence profiling, they construct an interprocedural data flow graph, in which they detect pipeline patterns for parallelization.

Another work that extracts pipeline parallelism was presented in 2010 by Tournavitis and Franke [TF10]. Starting from a sequential C program, they iteratively decompose the data dependence graph top-down into pipeline stages. Further, they also identify pipeline stages that allow replication. In principle, the decomposition and replication enables optimizing pipeline throughput automatically, however the authors do not discuss this possibility.

Bones [NC14] by Nugteren et al. is a static pattern-based tool for automatic parallelization for CUDA GPUs. It classifies algorithms in C code into “algorithmic species” based on the memory access pattern, computed from either a polyhedral representation or array reference characterizations. The detected algorithmic species are mapped onto a predefined catalogue of parallel skeletons.

With few exceptions, the applicability of automatic parallelization is limited to (multidimensional) array computations, because only in this case it is possible to derive accurate dependence information. In 2015, Aguston et al. [AAH15] presented a technique to tackle parallelization for linked data structures. Finding specific patterns in the source code that correspond to loops traversing linked data structures, they embed this linked list into arrays and convert the loop to index-based iteration, a process they call “skeletonization”. The result of this is presented to the user as a recommendation, providing parallelization hints.

The Distributed Multiloop Language by Brown et al. [Bro+16] is in the narrowest sense of the term not a tool for automatic parallelization. Instead, it poses as an intermediate language for implicitly or explicitly parallel programming languages. The intermediate language provides representations for data as well as

Table 4.6: Comparison of the pattern-based parallelization approaches

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
Rul et al.		✓		✓										
Tournavitis and Franke		✓	✓											
Bones		✓	✓		✓									
Aguston et al.	✓	✓	✓											
DML	✓		✓		✓	✓								
`APHES`	✓		✓		✓	✓	✓	✓		✓	✓	✓	✓	✓

nested parallelism, and supports multicore processors, GPUs, and even distributed, heterogeneous systems with non-uniform memory accelerators.

We summarize this section in Table 4.6, and compare the presented approaches to the `APHES` framework. Most of the pattern-based parallelizers target task parallelism, likely because tasks are easily represented as patterns. Of the discussed works, only Bones and DML target heterogeneous platforms. DML also supports multiple different platforms, albeit not cooperatively. Cooperative execution is thus the biggest distinction between the `APHES` framework and the works we presented in this section.

4.3 Autotuning Languages and Compilers

In this section we review recent advances in combining the fields of autotuning, compilers, and automatic parallelization.

In the domain of automatically producing portable application performance, two flavors have emerged. On the one side, programming languages or DSLs were developed in which the performance-relevant degrees of freedom are explicit. The language's compiler or execution environment configures these parameters, either at compile or execution time.

On the other side, the compiler community has invested time and effort in automating the tuning of program transformations for general purpose languages. That includes traditional compiler optimizations, automatic parallelization, or switching between versions or implementations. Traditionally, compilers make decisions based on heuristics, or cost or performance models. Models and heuristics must make assumptions about inputs and hardware characteristics. For general applications, the input domains are unknown, making accurate estimates of their characteristics hard. Hardware secondly evolves over time, which requires constant updates to the models and heuristics. Empirical search investigated in addition to heuristics and models to relieve compiler designers of those two difficulties.

In the following, we first examine related work in the domain of compilers that automatically parallelize or optimize programs either with the help of performance models and machine learning, or using empirical search. Subsequently, we explore the domain of explicitly tunable programming languages and DSLs.

4.3.1 Modeling and Machine Learning for Autotuning Compilers

Applying execution models to mapping an implicitly data parallel language onto a parallel machine was pioneered by Alan Sussman in 1992 [Sus92]. He builds a compiler for the functional Sisal language. Based on predictions of the parallel runtime obtained from an analytical model, his compiler maps Sisal programs onto systolic array processors. The model is implemented by the compiler and thus does not require the application developer to define the relevant features. Because we focus on compilers in this section, the same holds for all approaches we discuss here. Similarly, we classify all approaches in this section as offline tuning techniques, because they build the model offline.

A related approach is taken by Cavazos et al. in 2006 [Cav+06]. Instead of using a hand-crafted analytical model to predict execution time for sequences of optimizations in the SUIF compiler, they train a feed-forward neural network on a set of sample transformations. These *probe transformations* are applied to the input program. Observing the resulting speedup provides a training sample for the model. They show that only a small number of samples (64) is required to achieve an error rate better than 10%. In 2007 [Cav+07], they complement this approach by learning a logistic regression model instead. In this case, they build a model not on speedups but directly on dynamic application characteristics. The features that define these characteristics are hardware performance counters. The compiler observes the counters for an input program, and with the help of the model predicts the optimal sequence of optimization transformations.

Wang et al. provide an approach that can be seen as an intersection between the work of Sussman and Cavazos et al. [WO09]. They build a compiler for OpenMP programs which optimizes the parallel mapping. In a (simple) OpenMP program, there are two degrees of freedom: The number of threads active in a parallel section and the policy of scheduling chunks of data onto threads. On a set of program features, they train a neural network to predict the optimal number of threads, and a support vector machine to determine the scheduling policy. Relevant program features are static properties, such as instruction or load/store counts, and dynamic features, such as loop iteration counts and cache hit rates. Subsequently, Tournavitis et al. [Tou+09] extend this technique for automatic parallelization. They detect parallelism dynamically, and map it onto OpenMP. SVM training and prediction are applied to optimize the mapping.

The Milepost GCC [Fur+11] compiler complements the Open Source compiler `gcc` with machine learning capabilities. From a large set of static program features, such as the number of basic blocks, the CFG structure, or the number of calls, they build a model on training programs and use it to select a set of optimization options for the `gcc` compiler for an unknown program.

An interesting goal is pursued by Fried et al. [Fri+13]. Instead of hunting for optimal compiler optimizations or parallel mappings, they use machine learning to predict which parts of a sequential program should be parallelized. With DiscoPoP they analyze the target program for dependences and extract features, for instance dependence graph properties such as dependence counts between instructions. The paper explores the machine learning algorithms SVM, decision trees, and AdaBoost for prediction and reports a prediction accuracy of up to 92%.

Building upon the work by Cavazos et al. [Cav+06; Cav+07], Park et al. construct an optimizer for polyhedral compilation [Par+13]. The polyhedral model provides a unified framework for applying loop transformations. Which transformations to apply and how to parameterize them is considered a tuning problem by Park et al. They consider the transformations loop fusion and distribution,

Table 4.7: Comparison of the model-based autotuning compilers

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
Sussman	✓		✓						✓		✓			✓
Cavazos et al.	✓		✓						✓		✓		✓	✓
Wang & Tournativits et al.	✓			✓					✓		✓		✓	✓
Milepost GCC									✓		✓		✓	✓
Fried et al.				✓					✓		✓		✓	✓
Park et al.	✓								✓		✓		✓	✓
`APHES`	✓		✓		✓	✓	✓	✓		✓	✓	✓	✓	✓

loop and register tiling, wavefronting, thread-level and SIMD-level parallelization, and pre-vectorization. They implement the speedup prediction model by Cavazos et al. with the help of six machine learning algorithms, among them linear regression, neural networks, and SVM. The selected transformations achieve substantial speedups on a variety of benchmarks.

Table 4.7 recapitulates the core aspects of the approaches in this section, comparing them to the `APHES` framework. Neither of the presented compilers targets heterogeneous platform. Although all compilers except Sussman's learn their model either offline or online based on automatically extracted features, none of these approaches consider program inputs at runtime. Both of these missing aspects set the `APHES` framework apart from the compilers in this section.

4.3.2 Search-based Tuning Compilers

Using machine models and machine learning methods to determine optimization sequences or parallelization strategies has a long history. The greatest downside of this approach is that the quality of its prediction is tightly coupled to the quality of the model. A model that does not accurately reflect the hardware or application characteristics may not provide the compiler with the optimal decisions. Moreover, the model is inherently tied to applications and hardware properties it was *designed* for. These properties are, however, by no means static. Especially hardware features change at an astonishing pace, and models must adapt at the same rate. Consequently, there is an interest in the research community to explore decision making processes based on empirical search as an alternative.

The first compilation framework that explored this path was PHiPAC [Bil+97] by Bilmes et al. Initially designed as a catalogue of coding guidelines for BLAS C programs, the authors provide a tunable code generator for GEMM kernels based on these guidelines. The generator encodes preferences such as prefer local variables over array reads, or increase locality. To configure tuning parameters such as tile sizes, they use a simple random search, recompiling and benchmarking the kernel for every configuration.

Almagor et al. [Alm+04] pursue a similar goal as the Milepost framework. For their own adaptive compiler they attempt to find alternative optimizer sequences instead of the predefined `O2` or `O3` levels provided by most established compilers. Using a full exploration of the space of compilation sequences they show that most local minima an empirical search might produce are within an acceptable range from the global optimum. To find a local minimum, they employ genetic algorithms, hill climbing, or greedy search.

In 2008, Chen et al. [CCH08] developed the CHiLL framework. CHiLL is a polyhedral loop transformation framework whose key idea is to control transformation sequences via high-level transformation recipes. Sequences of operations such as permutation, tiling, unrolling, or splitting are defined by a developer. Free parameters in these rules such as unroll factors and tile sizes are determined through a systematic full exploration. In 2009, Tiwari et al. [Tiw+09] extend this to use a more sophisticated search, Parallel Rank Ordering, which bears similarities to the Nelder-Mead simplex algorithm and which can be executed in parallel. Tiwari and Hollingsworth later develop this further into an online tuning scenario [TH11]. Exploiting ActiveHarmony's online tuning capabilities, they produce an adaptive compiler, which generates new code on the fly. By extending CHiLL with GPU-specific loop transformations, Rudy et al. develop CUDA-CHiLL [Rud+10] in 2010. Using CHiLL's polyhedral loop transformation engine and transformations such as tiling, permutation, and data transfers, CUDA-CHiLL generates CUDA code from sequential inputs. While the transformation sequences are still handwritten, Rudy et al. use autotuning in their evaluation to optimize tile sizes.

The ROSE compilation system by Liao et al. [Lia+09] can be seen as a frontend compiler for CHiLL and ActiveHarmony, although it supports alternative optimizers and search engines as well. The primary goal of ROSE is to simplify whole-program tuning. To reduce the complexity of analysing a program entirely, the compiler first extracts candidate kernels into standalone functions. Candidate kernels need to be identified by an application developer.

The Cetus automatic parallelizer [Joh+04] is a source-to-source compiler which translates parallelizable loops in C programs into parallel OpenMP loops. In 2010, Dave and Eigenmann [DE10] combine it with an offline tuner. Using Combined Elimination by Pan and Eigenmann [PE06], the tuner provides optimization directives to Cetus. In each iteration, Cetus generates a program version, which is again evaluated by the tuner.

Baskaran et al. use the polyhedral model to automatically translate C programs to CUDA [BRS10]. On top of the polyhedral parallelization and optimization algorithm Pluto, their compiler uses autotuning to pick optimal transformations. To select tile sizes that optimize the data movements required for GPU offloading, they formalize the cost of data movement as a constraint non-linear optimization problem and solve it with sequential quadratic programming [Bas+08b]. For tuning the remaining parallelization and transformation process, they employ empirical search. To reduce the complexity of the search space, they drastically prune it using analytical models, representing among others memory loads and stores, data transfer costs, and shared memory accesses [Bas+08a].

INSIEME, an autotuning compiler for parallel languages, represents an hybrid between offline and online tuning [Jor+12]. The compiler component of this approach lowers parallel C or C++ programs into INSPIRE, an intermediate representation able to model the parallel behavior of OpenMP, MPI, or OpenCL. With help from the autotuner, it optimizes the program according to *multiple objectives* concurrently, such as runtime and energy efficiency. In an offline tuning phase, the tuner determines Pareto optimal configurations using differential evolution. The resulting different versions of the code are compiled into a single binary program. At execution time, the runtime component then dynamically dispatches between versions according to a policy defined by the application developer.

Table 4.8: Comparison of the search-based autotuning compilers

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
PHiPAC			✓						✓	✓				
Almagor et al.			✓						✓	✓				
CHiLL/ROSE	✓		✓		✓			✓	✓	✓				
Cetus	✓		✓						✓	✓				
Baskaran et al.	✓		✓		✓				✓	✓	✓			
INSIEME								✓	✓	✓				
Patty	✓	✓		✓				✓	✓					
Banerjee & Ranka			✓						✓	✓				
`APHES`	✓		✓		✓	✓	✓	✓		✓	✓	✓	✓	✓

The Patty tool by Molitorisz et al. [MMT15] can be seen as an interactive automatic parallelization tool. Integrated into a C# editor, it detects parallelizable segments dynamically. Detected segments that match data-parallel loops, master/-worker patterns, or pipeline patterns are presented to the application developer, and, if confirmed, automatically parallelized. Patty does not tune the transformation directly, but generates parallel code annotated with the TADL language, which is an annotation DSL to define tunable parallel architecture. We describe TADL further in Section 4.3.3. The TADL program can be optimized by an offline or online tuner.

Banerjee and Ranka present a specialized tuner generating matrix multiplications for a spectral element solver [BR15]. Unlike most compilers discussed in this section, their transformations of the code are based on domain knowledge. Using a genetic algorithm, they optimize the loop structure with the help of CHiLL.

We compare the `APHES` framework to the parallelizing compilers using search-based tuning presented in this section in Table 4.8. There is no support for multiplatform or cooperative execution in any approach except ours, and only CHiLL and Baskaran et al.'s polyhedral compiler generate GPU code. Unlike `APHES`, none of the listed compilers produces input-sensitive optimizations.

4.3.3 Languages

While automatic parallelization is a key area in compiler design and research, the result is in practice often inadequate. The primary cause is that compilers need to reason about code statically and thus often must make worst-case assumptions. Transforming code optimistically on the other hand can introduce errors and is thus mostly avoided. This bar can be lowered with the help of specialized general purpose or domain specific languages. Making a developer explicitly express the assumptions a compiler has to make enables automatic transformation with little developer overhead. Prominent examples of this approach are OpenMP or CUDA. In this thesis, however, we are not primarily concerned with automatic parallelization, but with combining automatic transformation with runtime autotuning. In the following, we thus examine related works from the domain of tunable languages.

Voss and Eigenmann's ADAPT system [VE00; VE01] is one of the first approaches to apply this technique. It can be seen as a meta compiler. Developers describe loop optimizations in the ADAPT language. The ADAPT system applies transformations dynamically according to heuristics defined by the developer using dynamic compilation. It explores the optimization space by exhaustively sampling the runtime of the variants that are applicable according to the developer defined heuristics.

The features of the Qilin programming system by Luk et al. [LHK09] are closely related to the goals of this thesis. Using a C++ API language, developers implement explicitly parallel kernels, also including descriptions of the data layout. This language is JIT-compiled into machine code for CPUs and GPUs. As in the present thesis, the kernel workload is distributed between CPU and GPU dynamically, however using a fundamentally different tuning technique. To estimate the optimal distribution, Qilin samples several subdivisions of the work. First, the input is split into two components, one for each platform. Each partition is further subdivided, and then sampled on its respective platform. Qilin assumes that the runtime on each platform is linear in the partition size. It fits linear models to the observed samples, which model the runtime as a function of portion of the input assigned to a platform. Qilin can then estimate the optimal distribution by minimizing the sum of the models.

Otto et al. [OPT09] presented XJava, an explicitly parallel programming language embedded into Java. Using new language constructs, developers express parallel patterns such as pipelines or master-worker in a concise and compact way. In 2010, Otto et al. extended their compiler and runtime system to automatically tune parameters they infer from the XJava program [Ott+10]. Parameters are for instance the thread count, thread load balancing, or pipeline stage replication factors. The runtime system applies white box tuning to these parameters. Otto et al. provide heuristics for optimal choices of configurations, for example minimizing the number of idle threads by greedily replicating stateless pipeline stages or by creating nested parallel tasks until worker threads are saturated. This tuning approach is similar to the one adopted in Sambamba.

As part of his doctoral thesis [Ans14], Ansel et al., who published OpenTuner in 2014, proposed PetaBricks in 2009 [Ans+09]. The PetaBricks language is implicitly parallel and features algorithmic choice. With this feature, the authors tackle the problem of choosing the right implementation of an algorithm for varying inputs. For example, the optimal choice of a sorting algorithm depends strongly on the size of the input. To select an algorithm, PetaBricks learns thresholds that define which algorithm to choose. For this, the PetaBricks autotuner originally used an evolutionary algorithm [Ans+11], which was later supplemented by OpenTuner. As an additional tuning mechanism, Ding et al. [Din+15] train PetaBricks to adapt to inputs with a two-level learning approach. First, inputs are clustered into a fixed number of input classes, for which it then determines the optimal algorithmic choice using autotuning. The second level refines the clustering by training a set of classifiers on the clusters of the first stage.

Concurrently to XJava, Schaefer et al. [SPT10] developed the Tunable Architecture Description Language in 2010. Like XJava, TADL is a language mechanism to develop architectures from parallel patterns such as pipelines or producer-consumer. An architecture description embeds the API of a C# program into a high-level parallel pattern. For example, the description defines the order of stages in a pipeline, each of which is implemented by a C# method. Algorithmic choice is also supported, modeled as a nominal n-ary decision between alternatives. The description compiler deduces tuning parameters from the architectural composition. As in XJava, example parameters are the number of threads, producer-consumer buffer sizes, or pipeline stage replication behaviour. TADL shares two co-authors with XJava.

The PATUS code generation and auto-tuning framework by Christen et al. [CSB11] targets stencil computations on CPU and GPU platforms. It offers a language to define iterative stencil codes in an array notation. On top of that, the language can be used to define strategies for parallelization or access optimizations, such as blocking. In a strategy, parameters may be marked as *auto* for autotuning. This specification is then mapped onto parallel hardware using OpenMP and CUDA. The autotuner iterates the mapping process, using Powell search, Nelder-Mead, or evolutionary algorithms to search for parameter configurations.

Inspired by Ocelot the work by Lee et al. [LRG12] compiles PTX for CPU and GPU and distributes the workload. The workload is executed cooperatively, which is why they name their tool *Cooperative Heterogeneous Computing* (CHC). Most relevant to this thesis, the authors devise a scheme to determine the workload distribution automatically. The distribution scheme is largely similar to Qilin, although CHC does not sample several partitions of the input for every platform. Instead, it measures the extreme points, i.e., the smallest and largest partition. Like Qilin, it then estimates a linear interpolation between the extremes. The work is partitioned according to the intersection of the linear predictions for both platforms.

The Halide language by Ragan-Kelley et al. [Rag+13] pursues a similar goal as PATUS, and is another stencil DSL. Unlike PATUS, however, it is primarily targeted at image processing. Using a functional language, developers can compose large and complex image processing pipelines. In a second step, developers then build schedules from building blocks such as tiling, fusion, or sliding window. This choice is either manual or with the help of an autotuner. Scheduling additionally includes parallelization or vectorization, and also generates GPU kernels. Using genetic algorithms, the autotuner attempts to optimize the schedule, iterating compilation of the stencil pipeline.

With libHawaii [RDP14], Ranft et al. present a library-based approach to cooperative heterogeneous execution. Like Qilin, libHawaii offers building blocks to implement parallel programs. However, it is targeted at stream processing applications and enables developers to implement nested processing pipelines out of task-parallel and data-parallel building blocks. Both task-parallel and data-parallel implementations can execute on any platform in the heterogeneous system. For data-parallel execution, the work can be partitioned across platforms and be executed cooperatively. The libraries runtime system optimizes the execution of the pipeline for both energy and runtime efficiency. For that, it exploits the streaming nature: For the data-parallel execution, the relationship between work partition and runtime is assumed linear as do Qilin and CHC. To optimize runtime, the partitioning ratio is updated using a Kalman filter [WB95] after every processed work item. The key difference to Qilin and CHC is that the optimal ratio is refined progressively without sampling a fixed set of partitionings first. Because the ratio is perpetually adapted, libHawaii is also able to adjust to changing inputs automatically by effectively re-tuning its parameters.

PolyMage [MVB15] is another DSL for image processing. Like in Halide, image operators are defined in a functional language. Using a polyhedral compiler, PolyMage optimizes especially the tiling of the stencil loops. Being model-driven, its scheduling algorithms have few parameters, which the compiler explores exhaustively. Using this technique Mullapudi et al. report significant speedups over hand-tuned and autotuned Halide schedules.

Although it is not itself a DSL, the LIFT project [SRD17; Hag+18] considers itself an intermediate representation for DSL compilers. LIFT is itself a functional data-parallel language for which its compiler generates accelerator code with the aid of ATF and OpenTuner. Primitives of the language are operations such as `map`, `reduce`, `join`, or `split`. Optimizing a LIFT program means applying rewrite-rules to the functional representation. Low-level operations in LIFT that relate to OpenCL features lend themselves to tuning by exposing parameters such as device thread count, tile sizes, or work per thread. These parameters are optimized using ATF and its OpenTuner search.

The most recent contribution to tunable DSLs was made by Facebook AI Research in the form of Tensor Comprehensions [Vas+18]. The TC language is designed for expressing the complex computations and computational graphs involved in building deep neural networks in a compact and simple way inspired by the Einstein notation. In this sense, it is comparable to Halide and PATUS. The key difference is that TC aims to generate GPU code, which in the former two examples is only possible with complicated, hand-built schedules. Using the polyhedral model, the compiler optimizes loop fusion, multi-level parallelism, and the use of GPU memory hierarchies. A genetic algorithm-based autotuner drives the

Table 4.9: Comparison of the autotuning languages

	Data Parallelization	Task Parallelization	Static Analysis	Dynamic Analysis	Heterogeneous	Multiplatform	Cooperative	Online Tuning	Offline Tuning	Search-Based	Model-Based	Input Sensitive	Learned Model	Automatic Features
ADAPT								✓		✓				
Qilin					✓	✓	✓	✓			✓	✓		
XJava								✓			✓			✓
PetaBricks									✓	✓	✓	✓	✓	
TADL									✓	✓				
PATUS					✓				✓	✓				
CHC					✓	✓	✓	✓			✓	✓		
Halide					✓	✓			✓	✓				
libHawaii					✓	✓	✓	✓			✓	✓		
PolyMage									✓		✓			
Lift					✓	✓			✓	✓				
TC					✓				✓	✓				
`APHES`	✓		✓		✓	✓	✓	✓		✓	✓	✓	✓	✓

JIT compiler to optimize the degrees of freedom in e.g., the parallelism mapping and memory promotion.

Table 4.9 summarizes the languages and compilers presented in this section and compares them to `APHES`. Of all publications we reviewed in this chapter, Qilin, CHC, and libHawaii offer functionality and features most closely related to our own work. However, because they assume a linear relationship between platform distribution and performance, they are unable to optimize any additional parameters besides the distribution.

4.4 Summary

In this chapter we revisited prior art in the fields of autotuning and automatic parallelization, and investigated approaches combining the two techniques. The works we discussed all fulfil only a subset of the requirements of cooperative heterogeneous parallelization. With few exceptions, existing parallelizing compilers do not approach cooperative execution, because they are lacking the means to optimize the work distribution and parallelization simultaneously. Both the standalone autotuners and those integrated with the compilers are unfit to solve that particular optimization problem, because they either cannot effectively explore the search space or fail to correlate findings with current inputs. Effectively exploring the search space requires empirical search. To relate configurations and inputs, the empirical search must happen online, and the results of the search must additionally feed a model or lookup table. Although some of the tuning techniques we saw tick both boxes, they altogether are white box techniques, which ultimately disqualifies for an application in a multiplatform parallelizing compiler.

The works most closely related to this thesis are Qilin, CHC and libHawaii, which execute parallel kernels on CPU and GPU cooperatively, splitting the workload dynamically. Their approaches, however, are not automatic: Developers are required to program in a DSL. Moreover, they assume a linear relationship between workload size and performance and use this assumption to derive the optimal partitioning analytically. While this assumption is sensible in their use case, we cannot assume linearity when optimizing additional tunable parameters.

Chapter 5

Hybrid Online Autotuning

The `libtuning` library implements the hybrid online autotuning approach as a black-box tuner. Hybrid tuning combines online search and model-based prediction to provide the following three main features:

Context sensitivity: The runtime of the tuning kernel not only depends on configurations of tuning parameters, but also on the dynamic tuning context. In `libtuning`, a tabular memory records configurations and tuning context states. When a previously seen state recurs, the previous configuration for that state is used. In addition to the tabular memory, a prediction model can be constructed to predict good configurations for unknown context states.

Effective exploration: To explore the search space and bootstrap the table and prediction model, the tuner uses online search. The set of implemented default search algorithms can be extended by application developers.

Efficient exploration: While state of the art online search finds a (locally) optimal configuration relatively fast, there is room for improvement. The `libtuning` search algorithm called *hierarchical search* accelerates the search time substantially by automatically exploiting structure in the search space and by avoiding known bad configurations.

In this chapter, we present the hybrid tuning approach and `libtuning` in detail. We first described the hybrid tuning approach in our publication [Her+19], where it was applied to tuning parallel ray tracing[1]. The hybrid tuning approach presented in this chapter extends the approach described in the publication by introducing

[1]Hybrid tuning was contributed to the publication by the author of this thesis.

the underlying formalism, and integrating it with hierarchical search and into our `libtuning` autotuner. We investigated early designs of the hybrid approach in two Master's theses [Wen16; Kop18], which demonstrated the general feasibility of the approach. Hierarchical search was presented in our publication [PGT19], in which we apply it to polyhedral cooperative parallelization[2]. The introduction to hierarchical search we give here is based on that article and extends it by describing the full formalism of the approach.

In the following, we first introduce hybrid tuning in Section 5.1. Subsequently, Section 5.2 describes the `libtuning` architecture and its main building blocks. In Section 5.3 we present hierarchical search. Section 5.4 discusses the prediction models used in hybrid tuning. We finally summarize the details presented in this chapter in Section 5.5.

5.1 Hybrid Tuning: Combining Search and Prediction

Two directions exist in current state of the art autotuning designs: Optimization is generally either machine-learning or model-based, or search-based. The two alternatives offer different benefits and drawbacks. When parameter configurations are produced by a machine-learning or model-based predictor, an immediate reaction to a change in the dynamic context is possible: The predictor is queried for the updated context state and returns a single parameter configuration. With search-based tuning on the other hand, every change in the context entails overhead: To find a new configuration, the search is restarted, sampling new configurations until a new (local) optimum is found. The drawback of using prediction, however, is that the quality of the configuration depends on the accuracy of the predictor. If the predictor was hand-crafted, its accuracy is determined by how precisely its designer modeled the application and the dynamic context. If the predictor is learned, then its accuracy depends on how well the learned model generalizes from training data and how well the data reflected reality. While machine-learning eliminates the need for a meticulous human model designer, it generally requires large amounts of training samples to be accurate. With offline training, the qual-

[2]Both hierarchical search and polyhedral cooperative parallelization were contributions of the author of this thesis to the publication.

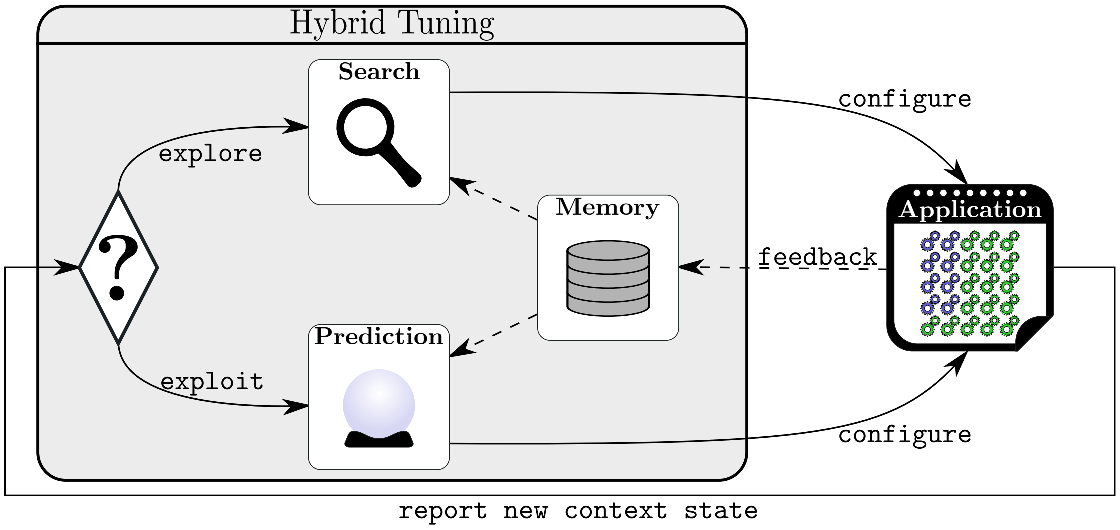


Figure 5.1: The hybrid online tuning workflow (figure based on [Her+19]). Feedback from the application is recorded in the tuning memory and is used to update the prediction and search states. When the application reports a change in the dynamic context, hybrid tuning selects between search and prediction to provide the next configuration.

ity of the learned predictor is governed by how well the provided training data represents the data that will be observed in production. Online training on the other hand can learn from data actually occurring in production, but there is a bootstrapping problem: If the initial model is imprecise, initial predictions are sub-optimal and performance is decreased. With search-based tuning, there is no bootstrapping problem and the quality of the found configuration is independent from any a-priori sample data.

With hybrid tuning, we present a compromise between the two directions. Hybrid tuning combines prediction and search, using online search to explore the search space and to continuously provide training data to update the prediction model. Because training happens online, prediction is insensitive to any a-priori sample bias. The hybrid tuning workflow is shown in Figure 5.1: Whenever a change in the dynamic context is detected, hybrid tuning can choose between exploration of the search space using search or exploitation of the trained prediction model. Any performance feedback provided by the application in either mode is used to refine the prediction model. A configuration table additionally stores the best known configuration for observed context states.

To implement hybrid tuning, we model the changing context and the reaction to changes as a Markov Decision Process (MDP). The model represents context changes as process state transitions and uses MDP decision making to choose between exploration and exploitation. In the following section, we briefly introduce our process model. Subsequently we discuss how we implement detection of context changes and our approach to balancing exploration and exploitation.

5.1.1 Context Sensitivity in Online Autotuning

To represent the context sensitivity problem in `libtuning` we construct an infinite-horizon MDP. As introduced in Chapter 2, an MDP is the quadruple (Σ, A, Φ, ρ).

In this model Σ is the *state* space, and A is the space of possible *actions*. The state space represents the dynamic context in which the tuner operates. In MDP literature the tuner would be called the "agent". A change in the context thus constitutes a change in the state space, causing the agent to pick an action. We define the action space $A = T$ as the set of possible configurations.

The agent picks a configuration according to a policy, usually called $\pi(s), s \in \Sigma$, which determines actions for every state. The *action model* $\Phi : \Sigma \times A \times \Sigma \to [0, 1]$ describes the probability $\Phi(s, a, s') = p(s'|s, a)$ when in state s to transition to the state s' when performing action a. We cannot define a "on-size-fits-all" action model, since it is unknown whether a chosen action has any actual influence on state transitions. Consider again the library parallelization example discussed previously: Actions affect the runtime of the called library functions, but we cannot reasonably assume that they have any effect on the context states reported to the library by its client. The action model thus has to be learned by observing state transitions.

Finally, the *reward function* $\rho : \Sigma \times A \times \Sigma \to [0, 1]$ determines the reward $r = \rho(s, a, s')$ for transitioning from state s to s' under action a. We set $r = -m$, i.e., the immediate reward achieved after every action selection is just the performance measurement. The negative sign serves to turn the minimization goal of autotuning into a maximization objective for the agent.

5.1.2 Approximating and Observing the Context

Before the autotuner can adapt dynamically to a context change, it first has to detect it. To achieve this, we first have to model and *quantify* the context. Thus, we need to determine the momentary state of both the application and the system. For this, we introduce *context indicators* $I_c(t) \in \mathbb{R}$ as "probes" serving to monitor various static and dynamic properties of the context at every time point t. Together, the set of the *application context indicators* $I^A_{c_A}(t)$ and the set of the *system context indicators* $I^S_{c_S}(t)$ produce a model $\tilde{K} = (I^A_0, \ldots, I^A_a, I^S_0, \ldots, I^S_s)$ of the real dynamic context K at a given point in time. In the remainder of this section we will only ever refer to the approximated context, and will thus use the terms interchangeably.

Indicators in hybrid tuning may represent both discrete and continuous quantities. While this is necessary to support general applications, it strongly affects the MDP model. Since the context forms the state space of the process model, incorporating continuous indicators turns it into a continuous Markov decision process. Such processes have significantly different solutions and convergence guarantees.

The discrete versus continuous indicator distinction also affects change detection. In a discrete context, detecting change just means watching the indicators for value changes. The same is possible for continuous indicators of course, but we define two additional schemes. Continuous quantities often exhibit high frequencies. Consider a system load indicator for instance once more: Measuring it essentially produces a different value every time, since the system the application is running on is never really at rest. Here, smoothing the response function with some low-pass filter can help, e.g., by averaging multiple indicator samples over time. While the changes of the indicator values may be high-frequency, they are often low-amplitude at the same time: When the system is busy, for example, the system load usually stays in the 100% range. Additionally it is easy to assume that small changes in the indicator values only effect negligible changes in performance, and thus in the optimality of the current configuration. For example, if the system load changes from 98% to 99%, the effect is likely not measurable. Both smoothing and down-sampling can help with this effect. Alternatively, it is possible to detect state changes purely by observing the effect of the indicators: Monitoring the measuring function, we can detect state changes when there are drastic changes in the function's value.

5.1.3 Adapting to Context Changes

Having a device to quantify the context and detect changes, the next step is to react to this change. Within the framework of our model reacting to changes is implemented by the policy π. We borrow from the field of reinforcement learning to develop our policy, which is loosely based on ϵ-Greedy, a frequently used policy in the field: With a probability of ϵ, we *explore* the search space using hierarchical search when encountering a state change. With a probability of $1 - \epsilon$, we *exploit* a predictor which is trained during exploration from the sampled configurations. Because the application-tuner interaction is iterative, both options are not "atomic" in the sense that they are a singular reaction to a state change. Instead, the hybrid tuner can be either in exploration mode or exploitation mode. While in the former, it will produce configurations as directed by hierarchical search. Hierarchical search is introduced in Section 5.3. While in the latter, it uses the predictor. We discuss prediction in more detail in Section 5.4. In particular, being in exploitation mode implies that there is only a single configuration update, namely when entering that mode. In exploration mode on the other hand, multiple configurations are sampled. Consequently, another state change might occur while the search is still ongoing. In that case, the current search state is saved and the search will be continued when the previous state recurs.

5.2 The `libtuning` Architecture

In this section we discuss the main components of the `libtuning` autotuner. Figure 5.2 shows its most important logical units and their interaction, both with each other and with the application. The primary application-facing component is the `HybridTuner` which encapsulates the other building blocks and manages their interactions. In terms of the sequence of operations, the first inner units the application will interact with are the `ParameterSpace` (labeled `PS` in the figure) and the `IndicatorSpace` (labeled `IS`). The first manages the application's tunable parameters, the second the application and system indicators. The `Hybrid Tuning` unit implements the hybrid tuning approach and interoperates with the application only through the `ParameterSpace` and `IndicatorSpace` layers. This component can be extended through application developer-provided `Search` and `Model` implementations. The former supplement hierarchical search with fundamental search algorithms it can use. The latter provide prediction models to hybrid tuning's prediction component.

In the following sections we introduce the components in more detail. In Section 5.2.1 and Section 5.2.2 we discuss the `ParameterSpace` and `IndicatorSpace`

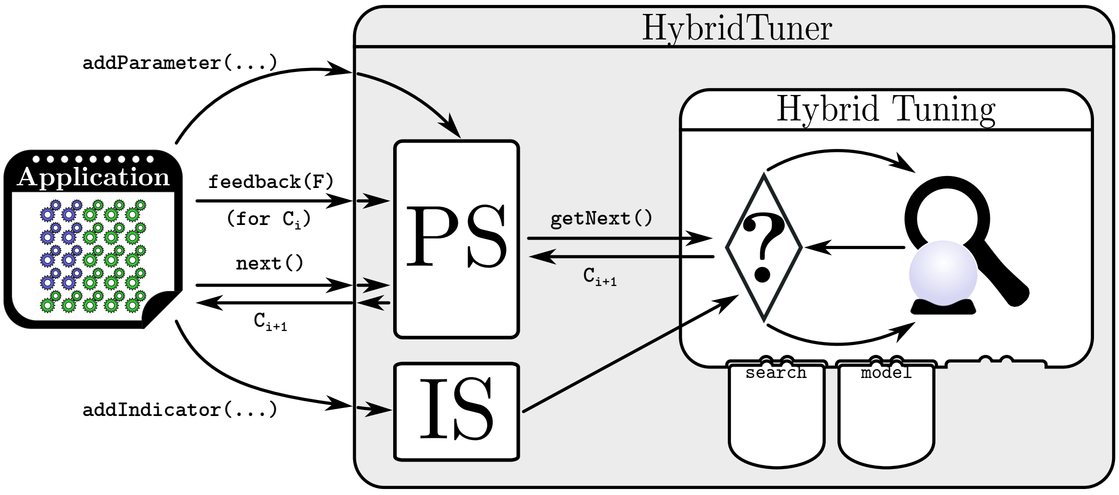


Figure 5.2: The `ParameterSpace` (PS) serves as an adapter between the application and the search algorithm. Feedback from the application is passed through the adapter to hybrid tuning, which computes the next configuration for the `ParameterSpace` to cache. Cached configurations are passed to the application upon the next call to `next()`. The `IndicatorSpace` (IS) is a registry for application-defined indicators.

components respectively. Subsequently we describe the `Search` implementations already included in `libtuning`and used by default by hierarchical search. Hierarchical search itself will be introduced in detail in Section 5.3. We defer the discussion of the builtin `Model` implementations until Section 5.4, where we introduce the model-based prediction component of hybrid tuning.

5.2.1 Tuning Parameters

Application developers employ autotuning to optimize parameters according to some criterion. The *nature* of these parameters is entirely specific to the particular application. Black-box autotuners are oblivious to the parameters' semantics. More importantly, the parameters have different *domains* in practice. For example, some parameters may be integer, other parameters may be floating point or might not even be numbers at all. Some parameter might assume legal values from an interval of $[0, 10]$, others from an interval of $[0.0, 1.0]$. Having to handle different domains is particularly cumbersome for `Search` and `Model` implementations. Hence, `libtuning` isolates the implementations from these application-specific parame-

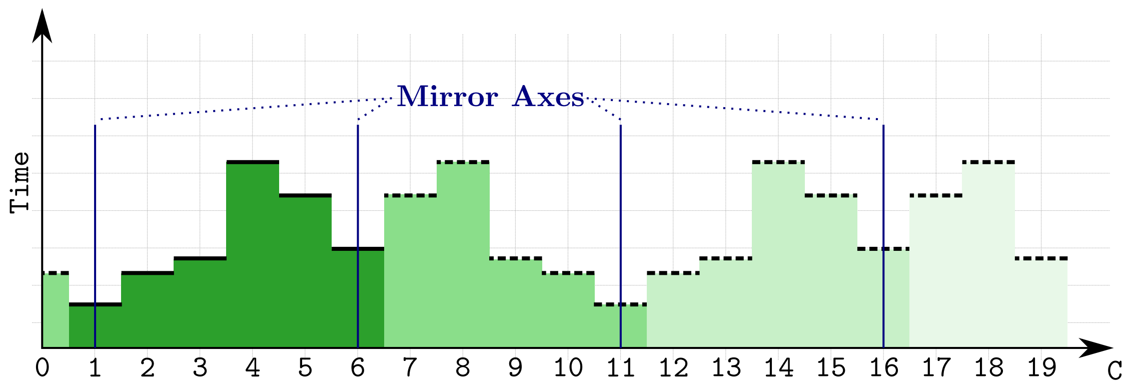


Figure 5.3: Mirroring Example: For an integer parameter from the range of $[1, 6]$, the measurement function as observed by search algorithms becomes a periodic step function.

ter properties. The isolation is provided by the `ParameterSpace` translation layer, which is an adapter between the application and its parameters on the one side and the search and prediction algorithms on the other. The translation allows search and prediction algorithms to operate on an n-dimensional real-valued unbounded space called the *search domain*, where n is the number of parameters. When the application reports feedback for a parameter configuration to the adaptation layer, hybrid tuning's state is updated and it is queried for the next configuration sample. This sample is a real-valued vector in the search domain. The configuration is cached within the translation layer. Once the application requests the next parameter update from the `HybridTuner` component, the cached configuration sample is translated into the application domain and then passed to the application. This mechanism has two positive effects. Firstly, it allows the development of search algorithms independent of the application which uses them. Secondly, it prevents several numerical problems in the search algorithms. For example, we observed that search algorithms get stuck in non-optimal configurations when trying to optimize in integer spaces. The searches never converge but flip-flop between a small set of configurations due to rounding. Through the `ParameterSpace` component, we move the responsibility for, e.g., rounding and boundary treatment away from the search implementation and into the application domain.

When registering parameters with `libtuning`, applications may define the individual domain and boundary mapping behavior as required. Additionally, `libtuning` provides a number of sensible default adapters for the parameter types used most often. These are integer and float parameters, either unconstrained or from an interval, and bag-of-labels nominal parameters. The latter are parameters which may take any value from a set of unordered labels. Through the default adapter, these are embedded as interval-constrained integer parameters, effectively numbering the unordered labels. Integer parameters are by default converted from the real-valued search domain by rounding. To restrict legal configurations to intervals, the adapter virtually mirrors the measurement function at interval boundaries. Configuration values returned by the search are then mapped into the interval with respect to the mirroring. Figure 5.3 shows an example of the procedure for a parameter that accepts integer values from the range $[1, 6]$. The figure displays the search space for the single parameter as it appears to the search algorithms. Even though the parameter can only accept values between one and six, the search is free to sample any value. For instance, sampling a configuration value of $C = 1.3$ would through rounding set the application parameter to a value of one. Sampling a value of $C = 10.7$ would first apply mirroring and then rounding, setting the application parameter to a value of three. Negative values are treated accordingly, but are for simplicity not shown in the example.

This mirroring strategy has a number of interesting properties. Most importantly, it retains all local and global optima: No newly introduced point can exceed the global optimum and all previously local optima remain local. Any local extreme on the *boundary* of the interval is still local, but its neighborhood is now symmetric. We cannot expect the mirroring strategy to be valid for every possible search algorithm. For instance, it is possible to misinform searches that make assumption based on *distance* of points in the search space. To the search, points may appear to be far apart, even though their measurements were obtained from points mapped to close proximity in the application domain. In this case we expect the application developers to supply a suitable adaptation mechanism with their parameter definition.

5.2.2 Indicators

Compared to tuning parameters, managing application and system indicators is much less complex. The `IndicatorSpace` component is little more than a registry containing the set of indicators. Besides registration, the component also implements change detection. What constitutes a change can be overridden by an application defining its own indicators. By default, a change happens when-

```
struct Search {
  // (Re)start the search and get the first configuration
  virtual Configuration &start() = 0;

  // Update the measurent and get the next configuration
  virtual Configuration &getNext(double Measure) = 0;

  // Reject the current configuration. Returns a new
  // configuration or null if no better one was found.
  virtual Configuration* reject();
};
```

Figure 5.4: The `Search` implementation interface.

ever the numerical value of an indicator changes. Because applications are forced to express their indicators as numerical values, that is sufficient to realize hybrid tuning. No domain translations as in the `ParameterSpace` are necessary.

In `libtuning`, application and system indicators need to be provided by the application developer. When used within the `APHES` framework, the frameworks compiler assumes the role of the application developer and defines the indicators automatically. Developer-provided indicators may be as simple as arbitrary application-side variables. Using program or function inputs as indicators is thus trivial, but arbitrarily complex indicators are possible. For example, an indicator monitoring system load could be defined, which would require querying operating system statistics every time its indicator state is queried. Additionally, depending on the nature of an indicator, observing its values may require down-sampling. For this purpose, the application developer can add a target resolution. This is likely useful for instance for an indicator monitoring system load. Wether the load is 99% or 100% likely has no discernibly different effect on tuning kernel performance, but the difference between 10% and 100% is probably noticeable. It is up to the application developer to define a meaningful sampling resolution, since `libtuning` cannot assume any semantics of a particular indicator.

5.2.3 Fundamental Search Algorithms

Because of the isolation and abstraction provided by the `ParameterSpace` adapter, implementations of search algorithms are required to supply only a minimal interface. Figure 5.4 shows the base class. In particular, the search algorithm is not concerned with actually sampling. Instead, it returns to the calling `ParameterSpace` layer a sequence of configurations. It is the callers responsibility to obtain the measurements and pass them back to the search. In the following, we use the phrase “sampling” to mean a single round-trip of returning a configuration to the caller and receiving a measurement in the next iteration. The mandatory operations a search algorithm implementation must provide are `start` and `getNext`, which (re-)start the search and obtain the next configuration, respectively. When obtaining the next configuration, callers also provide the measurement for the last configuration as an argument. The search implementation is expected to update its internal state upon a call to `getNext` and to return the next configuration it wants to sample. Additionally, an implementation may support rejecting individual configurations without measuring them. Callers use this mechanism to filter out illegal configurations or to ignore configurations they do not wish to sample. That can be the case if they use performance models and predict that a configuration will be sub-optimal.

When tuning an application, developers can provide their own implementation of a search algorithm or pick a default from `libtuning`'s small catalog of implementations. The library distinguishes between algorithms for nominal and non-nominal parameters. Unless the application developer requests otherwise, the novel *hierarchical search* algorithm for search in mixed spaces is used, In this section, we describe the two default fundamental search techniques used by hierarchical search. The hierarchical search technique itself will be discussed in Section 5.3. The tuning library includes as fundamental search algorithms an implementation of the Nelder-Mead downhill simplex algorithm, an ϵ-Greedy algorithm, and random and full exploration. While the latter two are self-explanatory, the variants of Nelder-Mead and ϵ-Greedy will be discussed in the following.

5.2.3.1 The Nelder-Mead Algorithm

In `libtuning`, the fundamental search strategy for non-nominal parameters is an implementation of the Nelder-Mead Algorithm [NM65] a heuristical derivative-free

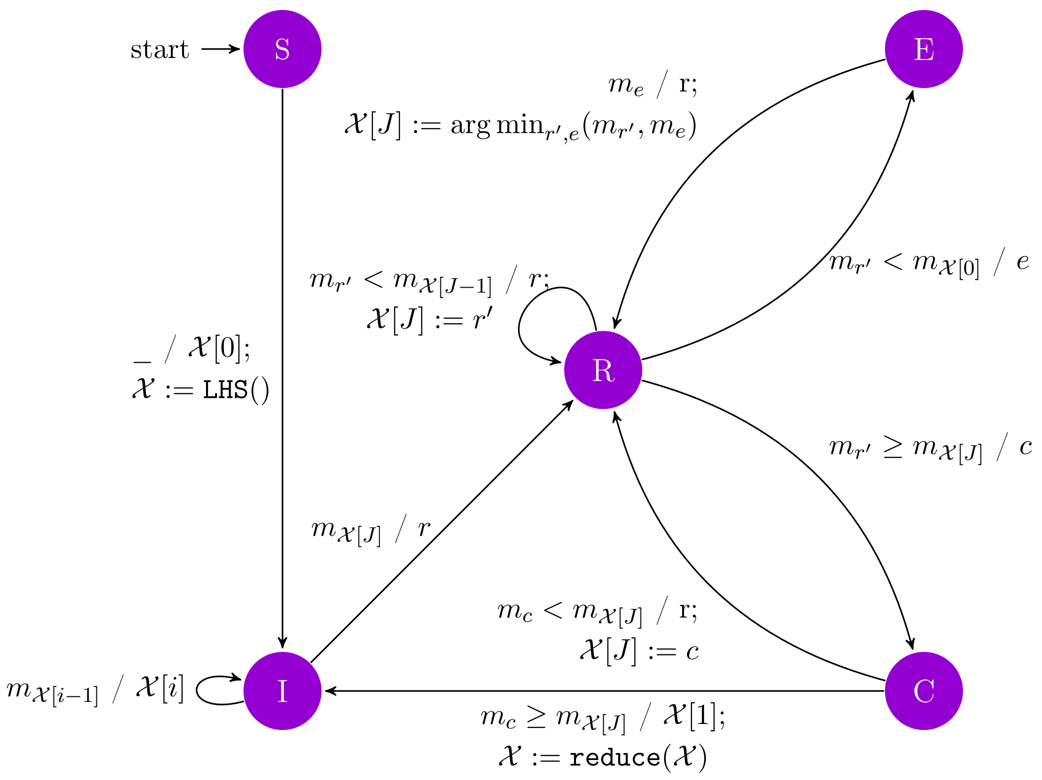

start
S
E
R
I
C
m_e / r;
$\mathcal{X}[J] := \arg\min_{r',e}(m_{r'}, m_e)$
$m_{r'} < m_{\mathcal{X}[J-1]}$ / r;
$\mathcal{X}[J] := r'$
$m_{r'} < m_{\mathcal{X}[0]}$ / e
_ / $\mathcal{X}[0]$;
$\mathcal{X}$:= LHS()
$m_{r'} \geq m_{\mathcal{X}[J]}$ / c
$m_{\mathcal{X}[J]}$ / r
$m_c < m_{\mathcal{X}[J]}$ / r;
$\mathcal{X}[J] := c$
$m_{\mathcal{X}[i-1]}$ / $\mathcal{X}[i]$
$m_c \geq m_{\mathcal{X}[J]}$ / $\mathcal{X}[1]$;
$\mathcal{X}$:= reduce($\mathcal{X}$)

Figure 5.5: The Mealy automaton for the Nelder-Mead algorithm: State transitions are triples $M/C;U$, where C is the next configuration and M is the optionally constrained measurement feedback $m_{C'}$ for the previous configuration C', and U is an optional update to the Nelder-Mead simplex $\mathcal{X}$.

numerical optimization technique. We introduced the algorithm in Section 2.1.2 and present our implementation here. We implement the algorithm as a small Mealy automaton, a finite-state machine shown in Figure 5.5. The implementation maintains a simplex as an array of search space points $\mathcal{X}$ along with their measurement values $m_x, x \in \mathcal{X}$. The points in $\mathcal{X}$ are sorted in ascending order with respect to m_x. The state machine changes its state in every tuning iteration. A state transition is defined as a triple $M/C;U$, where C is the next configuration and M is the measurement feedback $m_{C'}$ for the previous configuration C', optionally including a constraint on the measurement value. Lastly, U defines an optional update to the simplex $\mathcal{X}$, which is applied before the next configuration is returned. The state machine is defined by five states with the starting state S. The transitions implement the Nelder-Mead update rules.

From the starting state, the automaton transitions into the initialization state I by initializing the $J+1$-dimensional simplex from a Latin Hypercube sample. The first simplex point is returned as next configuration to the application. Latin Hypercubes [Par94; MBC79] are a technique to obtain space-spanning configurations and are often applied to seed search algorithms [Cha12]. To obtain a simplex based on this technique, every dimension of a bounded J-dimensional space is split into $J+1$ equally sized intervals. Then, we draw samples randomly such that no two samples fall into the same interval in any given dimension. Once the application returns the measurement feedback for the last simplex point, the automaton performs the `reflect` operation by transitioning into the reflection state R and returning the reflected point r. The transition into the reflection state's successor depends on the measurement feedback m'_r for the last reflected point r'. If m'_r is less than $m_{\mathcal{X}[0]}$, the measurement value of the best simplex point, the automaton performs an `expand` operation, returning the expanded point e and transitioning into the expansion state E. If m'_r lies between the best and worst simplex point, substitute the worst simplex point $\mathcal{X}[J]$ for the previous reflected point r' and perform another reflection, producing the next reflected point r. Otherwise, if m'_r s greater than or equal to the measurement value of the worst point, perform a `contract` operation and return the contracted point c. The expansion state E transitions back into the reflection state R after receiving the feedback m_e for the expanded point e. The transition substitutes the worst simplex point for the better one among e and r' and executes another `reflect` operation producing the next r. Similarly, the automaton performs a `reflect` operation in the contraction state C if the feedback m_c for the contracted point c is better than the value of $\mathcal{X}[J]$. In that case, $\mathcal{X}[J]$ is substituted for c. Otherwise, if m_c is greater than or equal to $m_{\mathcal{X}[J]}$, a `reduce` operation is performed. This operation forms a new simplex by moving every point towards $\mathcal{X}[0]$. Subsequently, every simplex point except $\mathcal{X}[0]$ needs to be sampled, so starting with $\mathcal{X}[1]$ the automaton enters the initialization state I. The reflected, contracted, and expanded points r, c, and e are computed as described in Section 2.1.2 using default parameter values suggested in the original Nelder-Mead article: $\alpha = 1, \beta = 0.5, \gamma = 2$. Our implementation of the Nelder-Mead algorithm does not support rejecting individual configurations.

Table 5.1: Tuning parameters for a simplified heterogeneous mapping example

Parameter	Range	Semantics
P	$\{CPU, GPU\}$	Binary platform choice
T^{CPU}	$[1, 48]$	The number of CPU threads
T^{GPU}	$\{32t, t \in [1, 32]\}$	The number of GPU threads
SM	$\{\texttt{true}, \texttt{false}\}$	The flag to enable shared memory
U^{SM}	$\{\texttt{true}, \texttt{false}\}$	The flag to unroll shared memory copy loops

5.2.3.2 The ϵ-Greedy Algorithm

For nominal parameters, `libtuning` implements an ϵ-Greedy algorithm, which achieved the best performance in an exploratory case study [Pfa+17]. The algorithm keeps track of the best known configuration. At every step, it either samples a random configuration with probability ϵ or uses the best known configuration with probability $1 - \epsilon$. Compared to the 2017 article, our implementation adds another hyper-parameter, controlling the decay of ϵ over time. We call this parameter $\gamma_\epsilon \in (0, 1)$. The probability to pick a random configuration at step i then becomes $\epsilon_i = \epsilon \cdot (1 - \gamma_\epsilon)^i$. Thus, γ_ϵ offers direct control over the convergence rate. Setting this hyper-parameter to zero returns to the original ϵ-Greedy behavior.

When the application rejects a configuration, the algorithm returns a new random sample.

5.3 Hierarchical Search

In real-world programs, parameters are not only frequently integer, but also often nominal. Applying a search algorithm to optimize a mixture of the two classes is highly problematic, because it limits the choices for apt search algorithms. Although `libtuning` does not proactively prevent application developers from picking an unsuitable search algorithm for the parameter mix, it is strongly discouraged. Search algorithms drawing conclusions based on, e.g., neighborhood of configurations will derive information that is purely coincidental. Consider as an illustration tuning the algorithmic choice of a matrix multiplication algorithm. When the nominal tuning parameter $A = \{ijk, ikj, strassen\}$ is embedded into integer space by simply enumerating the algorithms, the hill climbing algorithm

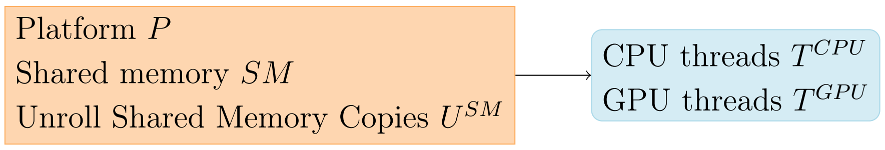


Figure 5.6: Decompose the global search space of the simplified heterogeneous mapping according to the parameter classes.

can be applied. If the hill climber now observes $m(ijk) = 2$ and $m(ikj) = 1$ it will conclude that space looks promising in the direction if ikj. Independent of whether $strassen$ is actually better than ikj this conclusion is already a coincidence. The order of the algorithms is irrelevant. As we found in our exploratory case study [Pfa+17], the vast majority of the empirical search algorithms used today are faced with this problem: They operate on a notion of direction or distance. A prominent exception are genetic algorithms, which, when using appropriate crossover and mutation operators, works well on combinatorial optimization problems.

In `libtuning`, we approach search in such mixture spaces in a structured way that is automatic and transparent to the application developer. As a motivating example, consider a simplified version of the heterogeneous parallelization approach: We generate parametric parallel code for the CPU and the GPU and add a tuning parameter to select the platform. In this example, we tune five application parameters in total: P, the binary platform choice, T^{CPU}, the number of CPU threads, T^{GPU} the number of GPU threads, SM, the flag to enable shared memory, and U^{SM} the flag to unroll the memory copy loops moving data into shared memory. Table 5.1 summarizes these parameters together with their ranges. Traditional, state-of-the-art tuning solutions embed these parameters into a single five-dimensional space containing $2 \cdot 48 \cdot 32 \cdot 2 \cdot 2 = 12,228$ valid points. The first improvement we make is to separate this single space into two smaller spaces by distinguishing between nominal and non-nominal parameters. Figure 5.6 shows the effect graphically: The space on the left now contains only nominal parameters, the one on the right only non-nominal parameters. Note that this separation does not actually reduce dimensionality. There are $2 \cdot 2 \cdot 2 = 8$ points in the nominal space and $32 \cdot 48 = 1,536$ points in the non-nominal space. Finding configurations in the individual spaces *independently* is however not legal, since the parameters are not independent: Assume for example that on the GPU the kernel runtime is defined by $m_{GPU} = 0.5 \cdot T^{GPU}$ if shared memory is disabled ($C_{SM} ==$ `false`), and

$m_{GPU} = \frac{1}{T^{GPU}}$ otherwise. Which value of T^{GPU} is optimal obviously depends on the selected value of SM. Ignoring the selected value of the nominal parameter SM when selecting a value of the non-nominal parameter T^{GPU} is hence not a viable path. Instead, configurations must be sampled from the cross product of both spaces, which means that the cardinality of the separated space remains unchanged. Nevertheless, with this first decomposition, we are now able to explore the spaces with *different* search algorithms that are appropriate for the respective parameter class, e.g., the ϵ-Greedy algorithm on the one hand and the Nelder-Mead algorithm on the other. To retain the dependence between parameters we configure parameters *hierarchically*: We first determine a configuration for the nominal parameters using the ϵ-Greedy algorithm and then a configuration for the non-nominal parameters using the Nelder-Mead algorithm. Because we cannot select configurations independently, we maintain one instance of the Nelder-Mead search per nominal configuration. In this example, the memory consumption of our Nelder-Mead implementation is thus increased by a factor of eight. A Nelder-Mead instance consumes $O(n^2)$ space where n is the dimension of the search space. The actual increase in memory footprint is however only about 30%, or by a factor of $\frac{8 \cdot 2^2}{5^2}$ to be precise: While the space consumption of Nelder-Mead is increased eight-fold, the number of parameters goes down from five to two. Although a 30% space increase appears to be small, the penalty can actually have a drastic impact, since it is exponential in the dimensions of the nominal search space. On the other hand, having paid this price we have achieved much simpler search spaces for the individual search algorithms and have enabled targeting the individual parameter classes with appropriate algorithms. Additionally, the instantiation of search instances occurs lazily. We thus incur the worst case memory consumption overhead only if we sample every nominal configuration, which in practice happens only for small nominal spaces.

Decomposing the search space into nominal and non-nominal subspaces opens the door for another optimization. In practice, nominal parameters often refer to algorithmic choices and as a consequence lead to different control flow paths in the application. In the example, the parameter P controls whether the CPU or the GPU implementation of the parallel code is to be executed. So transitively, this parameter also controls the *relevance* of the parameters T^{CPU} and T^{GPU}: When running the CPU version of the code, e.g., configuring the GPU thread count has no effect. Using the application developer's context knowledge of the application parameters' semantics, we can leverage such constraints. In the example above we can identify two more constraints: The number of GPU threads is tied to the selection of the GPU and unrolling the loop moving data into shared memory is only relevant when shared memory is enabled. Honoring these constraints produces the search space decomposition pictured in Figure 5.7. Unlike the first-level decomposition, this second step does reduce the cardinality of the space. We see that for this example we have achieved the highest degree of dimensionality reduction possible, having only one-dimensional search spaces. That also reduces the size of the sample space, because the constraints exclude irrelevant configurations. The total number of sampleable points is now $48 + 32 + 2 \cdot 32 = 148$: There are 48 configurations where $C_P == CPU$, 32 configurations where $C_P == GPU, C_{SM} ==$ `false`, and $2 \cdot 32$ configurations where $C_P == GPU, C_{SM} ==$ `true`. Although this example demonstrates that the constraints reduce the number of sampleable points significantly, we in general also incur a loss of information. When separating nominal and non-nominal parameter spaces, we were able to retain hidden dependences between nominal and non-nominal configurations by replicating the non-nominal search state. Since nominal configurations are selected first and non-nominal configurations second, replication is practical. For the second-level decomposition, this approach is infeasible because when two nominal spaces are interdependent, there is no meaningful way to order them after our decomposition. Early experimentation showed however that the constraint-based decomposition is not noticeably sensitive to the impact of missed hidden dependences, unlike the class-based decomposition.

In summary, even for this small example we managed to reduce the size of the search space by two orders of magnitude, while at the same time allowing the tuner to explore parts of the search space using different, appropriate search mechanisms. The size reduction is due to a decomposition of the search space by characterizing parameters into their respective classes and based on relevance constraints. In the following, we formalize the algorithm that implements the decomposition and search in the decomposed space. The general idea is to compute a decomposition into the minimal number of subspaces such that the relevance and class constraints are preserved. The first level of the decomposition is straight forward: We partition the search space $T = T^N \cup T^R$ into the naturally disjoint subspaces T^N of the nominal parameters and T^R of the non-nominal parameters.

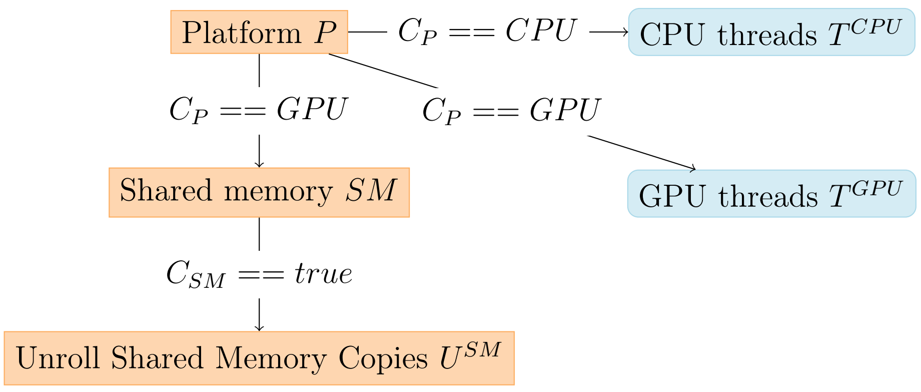


Figure 5.7: Decompose the global search space of the simplified heterogeneous mapping according to the parameter classes and according to application developer supplied constraints (based on Pfaffe et al.[PGT19]).

Relevance Constraints. The second level of the decomposition is computed based on the *relevance constraints* defined by the application developer. We write $\tau_i \xrightarrow{v} \tau_j$ where $\tau_i \in T^N$,$\tau_j \in T$ and $v \in \tau_i$ to say that the relevance of the tuning parameter τ_j depends on the configuration value $C_{\tau_i} = v$ of the nominal tuning parameter τ_i. When a parameter is relevant, the search must provide a configuration value for it. Otherwise the parameter should not be configured. Note that only nominal parameters may appear on the left hand side of the constraint. Naturally, neither circular nor contradicting constraints are allowed.

Local Search Spaces. We call subspaces of both T^N and T^R *local search spaces* when all elements of a subspace are subject to the same relevance constraints. To be precise, a subspace $\mathcal{L} \subseteq T$ is a local search space if and only if either $\mathcal{L} \subseteq T^N$ or $\mathcal{L} \subseteq T^R$ and $\forall(\tau_i \xrightarrow{v} \tau_j), (\tau_i \xrightarrow{v'} \tau_j'), \tau_i \in T^N, \tau_j, \tau_j' \in \mathcal{L} : v = v'$.

Search Space Graph. Together, the local search spaces and the relevance constraints form a directed acyclic graph, which we call the *search space graph* $\mathcal{S} = (\mathcal{L}_\mathcal{S}, E_\mathcal{S})$. The nodes $\mathcal{L}_\mathcal{S}$ of the graph are the local search spaces induced by the relevance constraints, which form the edges $E_\mathcal{S}$ of the graph:

$$E_\mathcal{S} = \{(\mathcal{L}_i, \mathcal{L}_j) \mid \exists \tau_i \xrightarrow{v} \tau_j, \tau_i \in \mathcal{L}_i, \tau_j \in \mathcal{L}_j, v \in \tau_i\}.$$

The roots of the graph are formed by precisely the local search spaces containing all of the unconstrained parameters.

Before we proceed with the introduction of the hierarchical search procedure, let us first discuss another more complete example of the search space graph decomposition. Consider the parameter set $T = \{A, B, C, D, X, Y, Z, W\}$, for which A

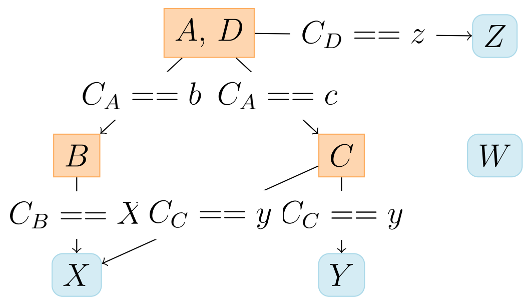


Figure 5.8: Example: Search space graph for the parameter set $T = \{A, B, C, D, X, Y, Z, W\}$, where $\{A, B, C, D\}$ are nominal and $\{W, X, Y, Z\}$ are non-nominal. The parameters are subject to the relevance constraints $A \xrightarrow{b} B$, $A \xrightarrow{c} C$, $B \xrightarrow{x} X$, $C \xrightarrow{y} Y$, $C \xrightarrow{y} X$, and $D \xrightarrow{z} Z$.

through D are nominal, X through Z are non-nominal. The first-level decomposition is then $T^N = \{A, B, C, D\}$, $T^R = \{W, X, Y, Z\}$. Further, let the dependence constraints be $A \xrightarrow{b} B$, $A \xrightarrow{c} C$, $B \xrightarrow{x} X$, $C \xrightarrow{y} Y$, $C \xrightarrow{y} X$, and $D \xrightarrow{z} Z$. A valid search space graph is shown in Figure 5.8. There are two subspaces without incoming constraint edges: The one containing the nominal parameters A and D and the one containing the non-nominal parameter W. The two spaces are disjoint because they contain parameters of different classes. Although X and Y both depend on the same configuration value for C, they are in distinct local spaces because X has another dependency, on B, which Y does not share. To find a configuration in the search space graph decomposition, hierarchical search traverses the graph along edges whose constraints are fulfilled. The process is described in the following.

Local, partial, and maximal partial configurations. Configuring the tuning parameters on the basis of the graph means determining *local configurations* for all nodes whose constraints are fulfilled. A local configuration for a local search space l is a configuration C^l for the parameters in that local search space. We call a set of local configurations for distinct local spaces a *partial configuration* $C^{\mathcal{S}}$ for the search space graph $\mathcal{S}$ when it contains no contradicting local configurations. Local configurations $C^l, C^{l'}$ contradict if there are two paths including l and to l' containing contradicting constraints. A partial configuration is called a *maximal partial configuration* if there is no possible local configuration that can be added to the partial configuration without contradicting a local configuration in the set.

As an illustration of contradicting configurations consider the local spaces containing B and C in the example above. The two spaces cannot be part of a partial

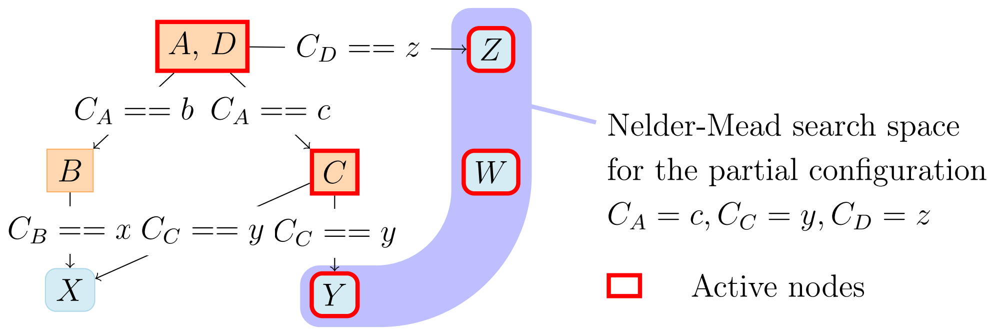

A, D
$C_D == z$
Z
$C_A == b$
$C_A == c$
B
C
W
$C_B == x$
$C_C == y$
$C_C == y$
X
Y
Nelder-Mead search space for the partial configuration $C_A = c, C_C = y, C_D = z$
Active nodes

Figure 5.9: Example: Hierarchical search in the search space graph for the parameters $\{A, B, C, D, X, Y, Z, W\}$. The active nodes are highlighted in red. The blue-shaded area marks the non-nominal spaces which are searched together by a single instance of Nelder-Mead for the partial configuration $C_A = c, C_C = y, C_D = z$.

configuration, because that would required A to be configured with the values `b` and `c` simultaneously.

Active nodes. Given a partial configuration $C^{\mathcal{S}}$, we say a node $l \in \mathcal{L}^{\mathcal{S}}$ is *active* for $C^{\mathcal{S}}$, if for all edges $\tau_i \xrightarrow{v} \tau_j$ on all paths p from l^R to l, $C_{\tau_i} = v$. In particular, any nodes without incoming edges are always active. There can be two such nodes of the graph, one containing all unconstrained nominal parameters and one containing all unconstrained non-nominal parameters. Producing a maximal partial configuration means producing local configurations for all active nodes recursively.

Hierarchical search produces maximal partial configurations by successively producing local configurations in active nodes until no additional nodes become active. After obtaining a local configuration for the current node, the search recurses into active neighbors of the current node in a deterministic order. Accumulating the local results into the partial configuration until no active neighbors remain produces a maximal partial configuration. For nominal spaces, hierarchical search exploits that finding nominal parameter configurations is a combinatorial optimization problem. We exploit that fact to enable using an individual search instance on a single active nominal node in isolation. Because parameters are interdependent, the search instance must consider the current partial configuration. For that reason, hierarchical search maintains concurrent search state for all partial configurations it produces. For non-nominal spaces on the other hand, this kind of isolation of the active spaces is not practical. Instead, all active non-nominal spaces are joined into a single space which is than explored using a Nelder-Mead instance. To account for the dependencies between nominal and non-nominal configurations, that instance is again maintained per partial nominal configuration. Because the non-nominal spaces are the leaves of the search space graph, they are activated last and as a single unit.

Consider Figure 5.8 for a step-by-step example of the activation process. Initially, all search state is empty. For the $\{A, D\}$ node, let the ϵ-Greedy search select the configuration $\{b, z\}$, which according to the constraints activates the nodes $\{B\}$ and $\{Z\}$. Nominal nodes are processed first, hence let the ϵ-Greedy instance for the current partial configuration $\{b, z\}$ pick the configuration $\{x'\}$ for the node $\{B\}$. That yields the partial configuration $\{b, z, x'\}$, which we use to finally activate the Nelder-Mead instance for the node $\{Z\}$ to produce a maximal partial configuration. Let the application return a measurement value of 0.7 for this configuration, then the search state after this first tuning iteration is:

Space	Instance	Local Config.	m_C
$\{A, D\}$	$\emptyset$	$\{b, z\}$	0.7
$\{B\}$	$\{b, z\}$	$\{x'\}$	0.7
$\{Z\}$	$\{b, z, x'\}$	$\{\ldots\}$	0.7

In the next tuning iteration, let the search instance for the $\{A, D\}$ node return the configuration $\{c, z'\}$. This partial configuration activates only the node $\{C\}$. Assume this node's search instance for this configuration returns $\{y'\}$ and the application measures a value of 0.9 for the resulting maximal partial configuration. Highlighting the updated values, the search state after the second tuning iteration becomes:

Space	Instance	Local Config.	m_C
$\{A, D\}$	$\emptyset$	$\{b, z\}$	0.7
		$\{c, z'\}$	0.9
$\{B\}$	$\{b, z\}$	$\{x'\}$	0.7
$\{Z\}$	$\{b, z, x'\}$	$\{\ldots\}$	0.7
$\{C\}$	$\{c, z'\}$	$\{y'\}$	0.9

For the third tuning iteration, assume the $\{A, D\}$ instance returns $\{c, z\}$, activating $\{C\}$. Let the ϵ-Greedy instance for $\{c, z\}$ return $\{y\}$. Then, Figure 5.9 shows the current activation state for this partial configuration. By applying two steps of the ϵ-Greedy algorithm, hierarchical search has produced the partial configuration $C_A = c, C_C = y, C_D = z$ and has thus activated the local spaces $\{A, D\}$, $\{C\}$, $\{W\}$, $\{Y\}$, and $\{Z\}$, all highlighted in the figure. Although $\{W\}$ was active concurrently to $\{A, D\}$ because it has no constraints, it is a non-nominal subspace and is thus only activated once the partial configuration is maximal with respect to the nominal parameters. For the given partial configuration the non-nominal spaces $\{W\}$, $\{Y\}$, and $\{Z\}$, highlighted in blue, are activated together for a single Nelder-Mead instance. Assuming the application reports a measurement of 0.6 for the resulting maximal partial configuration, the search state after the third iteration becomes:

Space	Instance	Local Config.	m_C
$\{A, D\}$	$\emptyset$	$\{b, z\}$	0.7
		$\{c, z'\}$	0.9
		$\{c, z\}$	0.6
$\{B\}$	$\{b, z\}$	$\{x'\}$	0.7
$\{Z\}$	$\{b, z, x'\}$	$\{\ldots\}$	0.7
$\{C\}$	$\{c, z'\}$	$\{y'\}$	0.9
	$\{c, z\}$	$\{y\}$	0.6
$\{Y, Z, W\}$	$\{c, z, y\}$	$\{\ldots\}$	0.6

Note that at this point, there are two concurrent search instances for the $\{C\}$.

Decomposing the search space in this manner does not only help to reduce the size of the space effectively, it also allows rejecting configurations more efficiently. Applications reject configurations when they determine heuristically that a configuration would produce poor performance. Often, however, a combination of only a small number of parameters contributes to this fact. Consider the heterogeneous offloading example (Figure 5.7, page 98): If the parallelized code is trivial or executes only for a small number of iterations, the application may rightfully judge that offloading this computation to the GPU is highly likely to actually decrease performance because of the overhead. The only parameter relevant to this is P, the one controlling the platform selection. Neither of the remaining ones contributes to the offloading decision. With the search space graph, we allow applications to also reject *partial* configurations. For example, the application is able to reject as soon as a partial configuration setting $C_P = GPU$ is constructed. As a consequence, we enable making rejection decisions early in the configuration process.

Rejecting configurations early further reduces the number of nodes that are ever activated.

5.4 Model-based Prediction

We explore two types of models for the prediction component of hybrid tuning in this thesis. The first one is a tabular nearest-neighbor approach. The second one is using generalized reinforcement learning, using function approximation to model the relationship between indicators, configurations, and runtime. The tabular predictor is simple: Every observed state is recorded in a table, alongside the best known configuration for that state. When the tuner enters a new indicator state, the table is queried for the geometrically closest known state. Although nearest neighbor searches can be implemented efficiently, there are two major drawbacks to this approach. The motivation for nearest-neighbor prediction is of course that the same configuration is likely good for similar states. When a new state is however a large distance from all states in the table, the query result can be arbitrarily bad. This problem can be mitigated by not using the query result if the distance is too large, but that requires defining a distance threshold. The second drawback is the table size: Storing information for every possible state is infeasible, especially when states are not discrete. Of course the table could only be populated through exploration or its size could be limited. But that would either disable the possibility to learn during exploitation or it would add another hyper-parameter to limit the size. In `libtuning`, we offer size limitation as well as control over the indicator resolution to the application developer.

A way around the drawbacks of tabular prediction is provided by generalized reinforcement learning. Whereas classical RL techniques are table-based themselves, generalized approaches approximate the tables through functions, as we have discussed in Section 2.1.3. By approximating the relationship between indicators, configurations, and runtime as a function, RL eliminates the need to either store every state or the need to choose which states to store. To predict a good configuration for a state, the function is used to find the configuration that maximizes an estimate of the reward for the state-configuration pair. Unfortunately, function maximization is expensive, which could bar hybrid tuning from being applicable to online tuning scenarios. Instead, we therefore keep a record of the best configurations found during exploration and consider only those as candidates for the prediction. Although this is still potentially large if hybrid tuning explores often enough, it is generally much smaller than a complete state table.

In `libtuning`, we implement a version of the Greedy-GQ variant of Q-Learning as our function approximation predictor. The Greedy-GQ algorithm was introduced in Section 2.1.3. We approximate the table as a function $\mathcal{Q}_\theta(s,a) = \theta \cdot \varphi(s,a)$, where the φ are features of the indicators defined by the application developer. As a default, `libtuning` uses normalized radial basis functions [KA97] as features: It defines $\varphi_c(x) = \frac{\exp(-\omega_c(x-\mu_c)^2)}{\sum_i \exp(-\omega_i(-\mu_i)^2)}$ where $x = (s,a)$ is the concatenation of the indicator state vector s and the configuration (or action) a. Compared to regular radial basis functions, which are just $RBF_c(x) = \exp(-\omega_c(x-\mu_c)^2)$, the normalized variant promises smoother space boundaries as well as smoothed out “gaps” between the basis functions [KA97]. The center μ_c and width ω_c are configurable parameters. The standard update equations of the Greedy-GQ algorithm are controlled by three hyper-parameters, α, β, and γ. Both α and β are learning rates. The third parameter, γ determines the influence of expected future rewards on the update. For our application, however, we assume that influence to always be zero: Accounting for future rewards allows propagating information about future states into the update. However, we assume that there is no correlation between the action we choose now and any subsequent state change. That assumption is of course only valid for applications in which the context state does not depend on tuning parameters. Setting γ to zero eliminates β, and simplifies the update equation to

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1} - \theta_t \cdot \varphi(s_t, a_t))\varphi(s_t, a_t).$$

5.5 Summary

In this chapter we introduced the design of the general purpose black-box online autotuner `libtuning`. At its core the tuner is a hybrid combining memorization, model-based prediction, and online search to quickly find configurations for known or unknown application and system states. Relevant aspects of the application and system state are, for example, the program or tuning kernel inputs and system load, respectively.

Using tabular memorization and model-based prediction, the autotuner remembers previously seen application and system states and the configuration it found optimal for those states. To provide a configuration for unknown states, the tuner uses online empirical search. The search algorithms in `libtuning` find (locally) optimal configurations, exploiting the structure of the search space and avoiding bad configurations to accelerate the search. By decomposing the global search space into a network of dependent subspaces based on parameter properties, our novel hierarchical search substantially reduces the size of the search space. Recommendations by the application rejecting configurations as “bad” further can reduce the number of sampled configurations which exhibit exceptionally bad performance.

Chapter 6

Automatic Heterogeneous Parallelization

In this chapter we describe the compiler component of the `APHES` framework in detail. The `aphes` compiler parallelizes loops in the input program for multiple platforms and instruments the result to interoperate with the `libtuning` autotuner through a runtime library. To analyze and transform the program we implement two techniques. First, the `aphes` compiler attempts to analyze the program using the polyhedral model. When this fails due to the constraints the polyhedral model imposes on representable programs, we fall back to classical dependence testing and more ad hoc transformations.

In the following, we first introduce the polyhedral model-based parallelization, and then the ad hoc method. The introduction is based on our publication [PGT19], that first presented our polyhedral parallelization approach[1]. Lastly, we discuss how the compiler and `libtuning` can interact to also incorporate analytical information that the compiler can derive into tuning decisions.

6.1 Polyhedral Parallelization and Partitioning

Generating code for cooperative heterogeneous execution from the polyhedral model is a three stage process. The process starts with computing a schedule for the SCoP that maximizes data parallelism. The next step is partitioning the outermost parallelizable loops for the platforms in question, which here are the CPU and a GPU, and then mapping the partitions onto the platform. Finally,

[1]The polyhedral parallelization approach was contributed to the publication by the author of this thesis.

the optimized and mapped schedule is translated into actual parallel code for the platforms. We describe our approach to partitioning and mapping as well as code generation, and finally the interoperation with the autotuner in the following sections.

6.1.1 Mapping and Partitioning the Schedule Tree

Our strategy for heterogeneous parallelization with the polyhedral model is based on Polly-ACC, and thus makes heavy use of the PPCG GPU mapper [Ver+13]. With its help, we first compute a schedule tree optimized for data parallelism from the SCoP. We describe the optimization algorithm in more detail in Section 2.2.3. Before mapping the optimized schedule to the GPU using the PPCG code generator we transform it in two steps, introducing tunable partitioning first and inserting tunable platform mapping second.

Fundamentally, partitioning a loop requires duplicating it and then modifying the boundaries of the copies: One loop copy handles the lower part of the iterations, the other one the higher part. To make that tunable, a tuning parameter determines the iteration that separates the lower and higher part. Mapping a partition to platforms is essentially the same operation: produce one instance of the partition for every platform and add a tuning parameter to select the instance dynamically. Note that these two steps only parallelize the outermost loop. Repeating the process recursively handles all data parallel loops. In the following, we describe in more detail how we realize this process within the polyhedral framework.

We transform the schedule tree top down recursively, searching for the outermost data parallel loops in the loop nest. To insert the partitioning, we begin by finding the outermost `band` which has leading parallel dimensions. The set of the leading parallel dimensions corresponds to the set of outermost data parallel loops. Every loop in this set is split into two parts: One for the high indices and one for the low indices. Although the number two appears to suggest itself because two platforms are targeted, the approach supports any number of partitions.

We split the outermost parallel dimension of the `band` by introducing a `set` node between the band and its parent with two filter children which implement the constraints for the partitions. A newly introduced parameter R_0 defines the split of the outermost parallel dimension of the band. R_0 is subject to the same constraints as the iterator of that dimension. In the partition filters, this parameter becomes

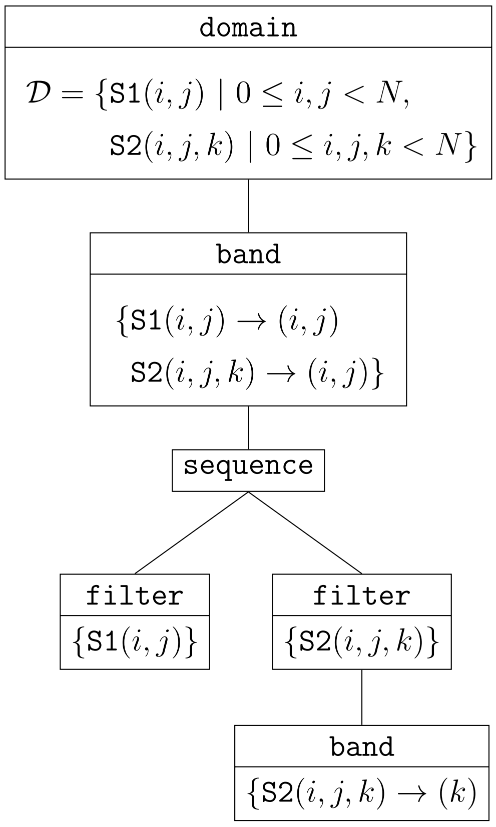
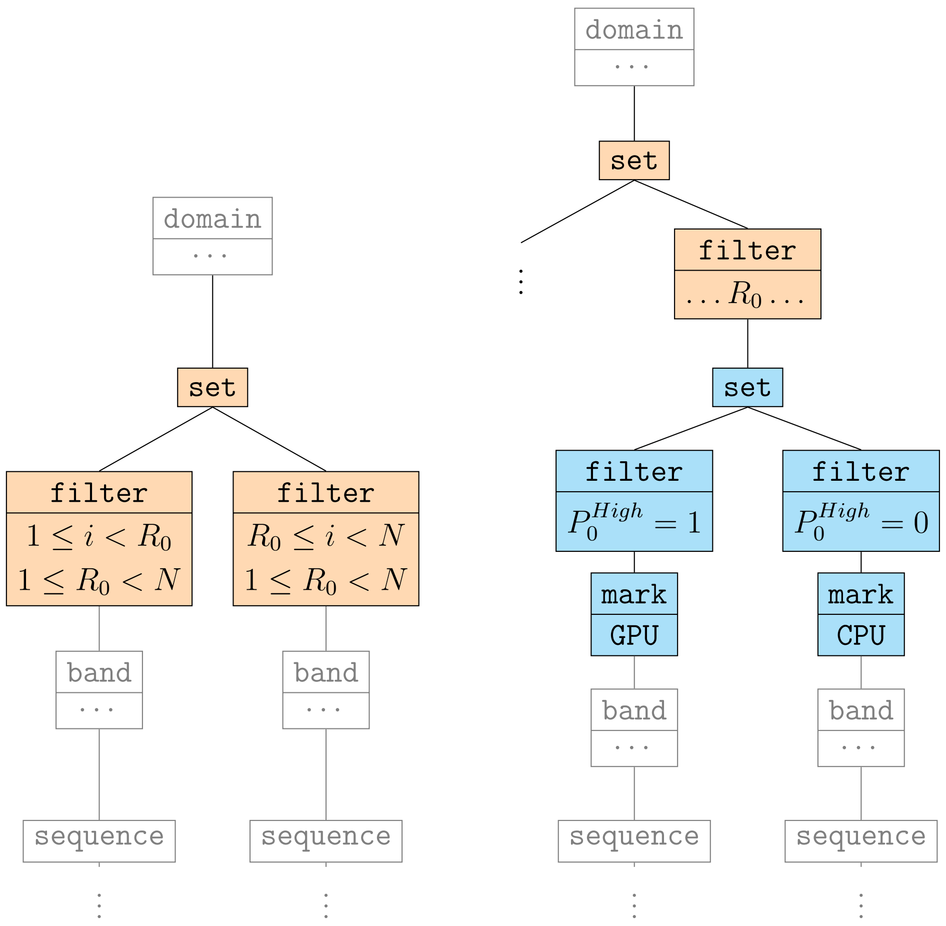

domain
$\mathcal{D} = \{\texttt{S1}(i,j) \mid 0 \le i,j < N,$
$\texttt{S2}(i,j,k) \mid 0 \le i,j,k < N\}$
band
$\{\texttt{S1}(i,j) \to (i,j)$
$\texttt{S2}(i,j,k) \to (i,j)\}$
sequence
filter
$\{\texttt{S1}(i,j)\}$
filter
$\{\texttt{S2}(i,j,k)\}$
band
$\{\texttt{S2}(i,j,k) \to (k)$

Figure 6.1: Original schedule tree of the matrix multiplication example in Figure 2.7. The figure is identical to Figure 2.8 and is repeated here for the reader's convenience.



(a) Partitioning the outermost `band` node: First, introduce a new tunable parameter R_0. Then insert a `set` node with two filters before the band that is to be partitioned. The new nodes are colored in orange. The filters select the indices i either less than, or greater or equal than this parameter, respectively. The original `band` node is duplicated.

(b) Selecting the platform for the high index partition: Between the `band` and the second partition's filter, introduce another `set` node layer with two filters, one for each platform. The new nodes are colored in blue. The filters select iterations based on the value of a new tunable parameter, P_0^{High}. The corresponding transformation introducing P_0^{Low} for the low index partition subtree is omitted here for clarity.

Figure 6.2: Example: Partitioning and platform mapping for the schedule tree in Figure 6.1. The nodes of the original tree are shown in lighter color.

the new upper and lower bound of that iterator to select high and low indices, respectively. The original `band` node is duplicated and attached to the filters. This process repeats recursively for the remaining leading parallel dimensions of the `band`. In summary, when partitioning d dimensions, we insert $2^d - 1$ `set` nodes in total, producing 2^d copies of the original band.

To implement platform selection, we follow exactly the same path. For every `band` node we created, we insert a new `sequence` node as its immediate parent, between the `band` node and the set filter that we created during partitioning. We introduce new parameters P_i^{Low} and P_i^{High} for the platform selection for the i-th loop, $P_i^r \in \{0,1\}$, for both the high and low index partitions. The two filters of the new `sequence` node implement the constraints $P_i^r = 0$ to select the CPU and $P_i^r = 1$ to select the GPU. Again duplicating the `band` node, we finally parallelize it for the GPU by applying the PPCG mapping algorithm to the band where $P_i^r = 1$. For its sibling, however, we insert a `mark` node, which we use during code generation as the starting point for generating OpenMP code for CPU parallelization.

Figure 6.2 illustrates this process for the schedule tree of the example in Figure 2.8, which we repeat here in Figure 6.1 for the reader's convenience. Partitioning the first dimension in the outermost `band` node of the tree produces the tree shown in Figure 6.2a. It shows the two newly inserted filters, using the partitioning parameter R_0 as a parametric lower and upper bound of the iterator respectively. Figure 6.2b then shows the platform mapping for one of the two partitions. Introducing a new parameter P_0, the inserted filters thus produce a GPU and a CPU path. The PPCG mapping algorithm is then invoked for the GPU path.

6.1.2 Code Generation

After all schedule transformations have been applied, we then convert the schedule tree back into LLVM IR. This procedure builds on top of the preexisting Polly-ACC [GH16] code generator. Most of the implementation of generating code for partitioning and mapping is thus already done.

We extend the Polly-ACC code generator in two major aspects, because it was designed to deal with GPU offloading only. To incorporate CPU parallelism, we integrated the code generator with an existing, experimental Polly OpenMP backend. This backend emits code for the `libgomp` OpenMP runtime library. The second extension is the instrumentation of the parallelized program to interoperate with the `libtuning` autotuner. During mapping and code generation we extract a number of tunable parameters from the parallel program. First and foremost those are the partition splits, $R_i \in [0, 1)$, as well as the platform mappings[2] $P_i^r \in \{CPU, GPU\}$ for $i \in [0, d)$ and $r \in \{High, Low\}$, where d is the number of outer loops being parallelized. For example, when parallelizing two outer loops of a parallel source region, this yields six tunable parameters: Two partition splits plus platform mapping deciders for the four partitions. Consequently, multiple partitions may be offloaded to the same platform at runtime. This is a potential source of extra overhead, but it is unfortunately unavoidable, because the non-affine constraints that would circumvent this are not expressible in the polyhedral framework.

Besides these work distribution parameters, we are able to elicit additional parameters from the mapping and code generation processes. Code generation for the CPU exposes the parameters $T_{i,r}^{CPU}$ controlling the number of threads, which are freely configurable at runtime. Tuning the GPU mapping on the other hand is unfortunately not as straightforward. Configuration options that affect mapping must be embedded within the polyhedral framework. An example is the number of threads per GPU block. Scheduling the threads is a tiling operation, i.e., the band is tiled into chunks of the number of threads. The point loop of the tiling then describes the kernel computations per thread, and the tile loop implements the execution of the distinct thread blocks. Making the number of threads parametric would turn this into a parametric tiling problem, which to date has no efficient general solution. Since we are not willing to forfeit tuning the GPU platform parameters, we are forced to follow an alternate route. To make the thread count tunable, we enumerate all configurations and generate the corresponding versions of the target region code statically. We express this directly in the schedule tree as a `set` node whose filter children represent the individual configurations. Thus, we obtain a single large schedule tree. Using the same approach, we also add parameters to determine the block height ($B_{i,r}$), to enable or disable promotion of local variables to shared memory ($SM_{i,r}$), and to unroll the loop moving data in and out of shared or local memory ($U_{i,r}^{SM}$, $U_{i,r}^{T}$). The block height parameters $B_{i,r}$ directly control the shape of a block of GPU threads. They thus affect the locality of accesses to global memory and the ability to coalesce accesses by the threads of

[2]In the schedule trees, parameters need to be integers. We use the symbols GPU and CPU here in place of the integer values to aid readability.

Table 6.1: Tunable parameters exposed by polyhedral parallelization

T	**Class**	**Range**	**Semantics**
Work distribution			
R_i	Ratio	$R_i \in [0, 1) \subset \mathbb{R}, i \in [1, d] \subset \mathbb{Z}$	Partitioning
P_i^r	Nominal	$P_i^r \in \{\texttt{CPU}, \texttt{GPU}\}, r \in \{High, Low\}$	Platform Mapping
GPU Platform Parameters			
$T_{i,r}^{GPU}$	Ratio	$T_{i,r}^{GPU} = 32k, k \in [1, 8] \subset \mathbb{Z}$	GPU Block Size
$B_{i,r}$	Ratio	$B = 2^k, k \in [0, 5] \subset \mathbb{Z}$	GPU Block Height
$SM_{i,r}$	Nominal	$SM_{i,r} \in \{\texttt{true}, \texttt{false}\}$	Shared Memory
$U_{i,r}^{SM}$	Nominal	$U_{i,r}^{SM} \in \{\texttt{true}, \texttt{false}\}$	Unroll Shared Accesses
$U_{i,r}^{T}$	Nominal	$U_{i,r}^{T} \in \{\texttt{true}, \texttt{false}\}$	Unroll Tile Accesses
CPU Platform Parameters			
$T_{i,r}^{CPU}$	Ratio	$T_{i,r}^{CPU} \in [1, \texttt{cores}] \subset \mathbb{Z}$	CPU Threads

a warp. The full list of parameters as well as their default ranges is summarized in Table 6.1. The value ranges of the block size and height parameters are not exhaustive. The chosen ranges are a trade-off between covering a sufficiently large area of the possible space and the number of versions we need to generate.

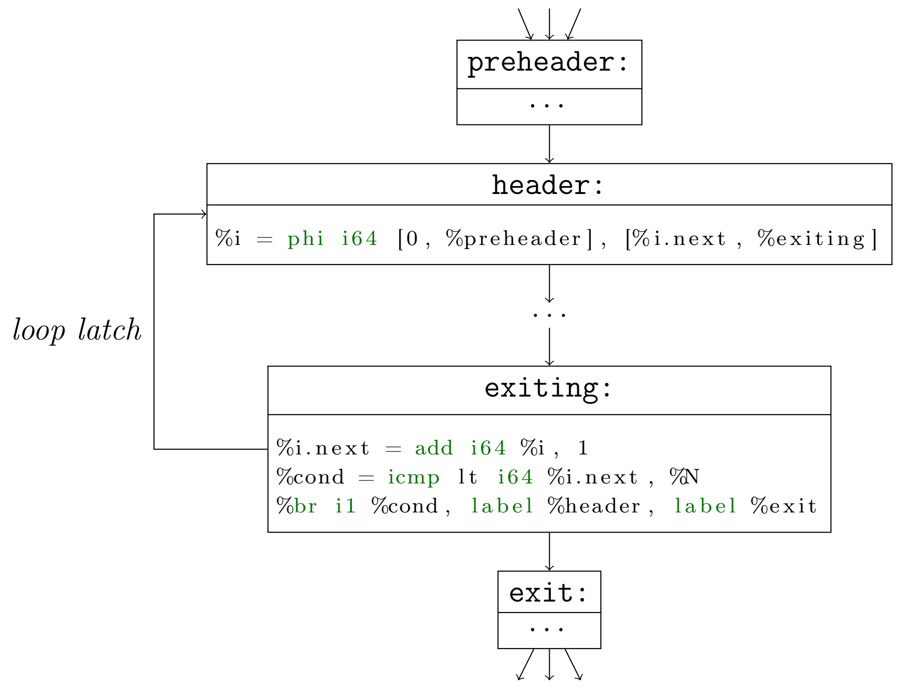
6.2 Dependence Testing for Parallelization and Partitioning

The polyhedral model fails to represent parallelizable loops in some cases. By relaxing its constraints, however, it is sometimes still possible to analyze the loops and detect data parallelism. Two examples for when that happens are function calls and inner loops without static control. Function calls are not modeled by the polyhedral model because the instructions within the function cannot be scheduled. Polyhedral compilers usually first inline functions and erase the function call because of this. For us that would however require speculatively inlining all function calls just to be able to detect parallelism, which would be incredibly expensive. Loops without static control on the other hand are not representable in a polyhedral framework at all. However, when only parallelization is the goal, such loops might be benign: If such loops contain no memory accesses or only accesses to memory that is provably local, parallelization remains unaffected. Therefore, when polyhedral modeling fails we fall back onto a second analysis and transformation pipeline, using classical dependence testing and ad hoc transformations. In an exploratory Bachelor's thesis [Bai16], sharing work between CPU and GPU was shown to be worthwhile. In this section we introduce the pipeline, the analyses performed by the `aphes` compiler to detect parallelism, and the platform specific program transformations.

6.2.1 Detecting Data Parallelism

The first step in parallelizing a given program is to detect the parts of the program that are parallelizable. In `aphes`, we focus on finding only data parallel loops during this phase. Böhm [Böh18] extended the scope to also support loops containing reductions, which can be seen as a special case of data parallelism with a relaxed constraint on data dependences. The detection of reduction parallelism is based on the work of Scheirle [Sch17]. On top of the restriction to data parallelism, we impose further constraints on the loops we analyse to simplify the implementation. Most importantly, we only consider top-level loops as candidates. Secondly, we assume loops have a particular structure in the control flow graph. The expected structure is illustrated in Figure 6.3: the loop must be defined by a single-entry-single-exit control flow subgraph, meaning there is a single basic block outside of the loop whose only successor is the loop header block and vice versa for the exit block. Further, there must be a unique exiting block, which branches to the exit and the header blocks. Consequently, the exiting block cannot be the loop header block. Note that this structural restriction is generally weak, since most loops can be normalized to take this form. A noteworthy counterexample however is a loop that has multiple exit blocks, which are basic blocks outside of the loop to which the loop body may jump. There is no universal way to merge these blocks. Lastly, we require that the outermost loop exhibits *static control*: The trip count, i.e., the number of iterations, must be statically computable. To be precise, this trip count may be either a constant value, or a provably loop invariant variable.

For all loops that pass these checks, `aphes` performs its *parallelism detection analysis*. This is a classical dependence analysis, attempting to disprove loop-carried data dependences. The LLVM framework contains various such analyses. The `aphes` compiler extends the existing facilities of the LLVM framework to support interprocedural analysis. It performs dependence analysis in loops which



```
%i = phi i64 [0, %preheader], [%i.next, %exiting]
```

```
%i.next = add i64 %i, 1
%cond = icmp lt i64 %i.next, %N
%br i1 %cond, label %header, label %exit
```

Figure 6.3: Expected control flow structure of a loop supported by `aphes`

contain function calls. To describe the effect of a call, we *summarize* all functions called within the candidate loop. Function summaries are effectively exhaustive lists of all memory accesses within the function that affect its pointer arguments or global variables. These accesses are precisely those that a caller is potentially able to observe. In general, summarizing requires over-approximating the accessed ranges since accurate alias analysis is in practice unavailable: When analyzing a memory access, it is often not exactly clear which argument or variable is read or written. In the worst case, summaries must assume that a function reads or writes every byte of memory in the program. That may for instance happen when accesses occur indirectly, that is, through pointers loaded from arguments or global variables. With the help of summaries, function calls now are analyzable by expanding the summary at the call site, thus virtually "inlining" the function. To disprove the existence of any loop-carried dependences, `aphes` extends LLVM's implementation of the dependence testing algorithm by Goff, Kennedy, and Tseng [GKT91] to support summaries.

To widen its applicability, `aphes` was extended to support for a parallelism scheme that in fact exhibits loop-carried dependences: parallel reductions [Böh18].

Parallel reductions differ from simple data parallelism in that they allow for the presence of specific data dependences, for which the values read and written to a conflicting memory location are dependent only via associative operations. A typical example of this is the summation of all elements of an array into an output variable. The output variable is read every iteration, incremented by an array element and written back into the same location. If the increment operator is associative, this operation may be executed in parallel. Because reductions can be seen as a slightly degenerated form of data parallelism, we support it as well.

6.2.2 Target Offloading

Having detected parallelizable loops, `aphes` then generates the parallel code. The code generation is implemented by an extensible collection of *target plugins*. A target plugin's purpose is to produce target platform specific code as well as the management code required to offload the computation. The `aphes` compiler then essentially becomes a driver for the transformations implemented by the plugins. Each plugin must thus emit three pieces of code: the *prologue*, the *target region*, and the *epilogue*. The interplay between these sections and the targets is shown in Figure 6.4, which expresses synchronous operations as solid arrows, and asynchronous operations as dashed arrows. In the prologue, targets are expected to set up target platform memory and asynchronously execute the target region. This potentially entails allocating space for the data accessed within the target region, including both arrays and scalar variables, and initiating data transfers. If host and target platform share the same physical memory, however, this step may be omitted. The `aphes` compiler processes the prologues of the individual targets in order, followed by all the epilogues. In the epilogue phase, targets await their target region's completion and post-process the results, if necessary. Post-processing can for example be necessary for parallelized reduction. They may do so asynchronously to overlap compute and all data transfers. When transforming loops that additionally contain reductions, the epilogues additionally aggregate the reduction results for the per-target slices of the work.

Currently, we have implemented two primary target plugins, namely for CUDA and OpenMP targets. To demonstrate that this approach also supports distributed systems, Lukas Böhm [Böh15] additionally provides a prototypical Xeon PHI target. In the following, we briefly discuss their design and implementation. Lastly,

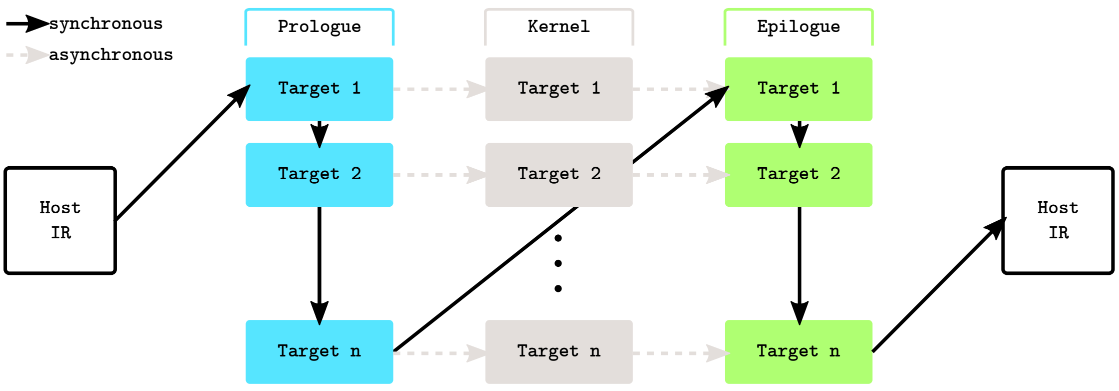


Figure 6.4: Code sections generated by target plugins. Solid arrows denote synchronous, dashed arrows asynchronous operations.

we describe the target specific tuning parameters the plugins expose, as well as how we model dynamic workload partitioning.

6.2.2.1 The OpenMP Target Plugin

Targeting OpenMP on the CPU is straightforward. Because of the shared memory, no complex management of allocations is required. To interoperate with the OpenMP library API, the plugin needs to outline the target region code into a separate function which will be called by the OpenMP runtime. Outlining mainly involves substituting the original boundary checks for checks against the boundaries for the chunk of iterations per OpenMP thread, and passing scalar variables and array pointers into the target region function. Hence, in the prologue, the target plugin copies scalars and pointers and invokes the OpenMP runtime. The epilogue merely requires waiting for the threads to finish.

6.2.2.2 The CUDA Target Plugin

In most systems, host and GPU do not share memory. Exceptions are mobile graphics units and system-on-a-chip platforms, which we at this point do not consider further in this thesis. Consequently, a major task of the CUDA target plugin is managing allocations and data transfers between the target platform and host. Unlike for the OpenMP target, we need to transfer both scalar variables as well as full arrays to the GPU. The number of scalar variables is statically known, which makes copying easy. On the other hand, both size and location of arrays are dynamic properties. Worse, both may change at runtime, and although the size of an array may be constant, the amount of data that needs to be transferred is not. On the one hand, the required data amount varies because of changing inputs to the program or parallelized region that affect the size of the array. On the other hand, changes are also caused by changes in the workload distribution due to tuning, which means that the changes are frequent. The CUDA target plugin thus manages memory dynamically through the runtime system further detailed in Section 6.2.3.

Having allocated and copied all data onto the GPU, the prologue then executes the target region, which is implemented as a CUDA kernel. Lastly, the prologue also initializes the transfer of the results back from the GPU device into the host memory. Besides allocations, all operations performed in the prologue are asynchronous with respect to the host, but strictly in order with respect to each other. Thus, CUDA API calls for copying to the device, launching the CUDA kernel, and copying from the device do not wait for the respective operation to complete. Waiting then happens in the epilogue, which synchronizes the host thread with the completion of the final copy.

6.2.2.3 The Xeon PHI Target Plugin

As a proof-of-concept, a target plugin for the Intel Xeon PHI parallel coprocessor was implemented [Böh15]. The Xeon PHI accelerator is an extension card mounted in a host system, running an independent instance of Linux. The host communicates with the Xeon PHI device system through a network stack, e.g., via low-level APIs or high-level MPI. For the `aphes` compiler, this has two important consequences:

1. Offloading computations to this platform requires a binary running on it natively.
2. Memory transfers and target region invocation require a custom communication protocol.

Generating the native binary is an involved process, because Xeon PHI support in open compilers and LLVM in particular is lacking. Böhm follows Damschen et al. [Dam+15] in their approach to generate C code from the LLVM IR and running the native Intel compiler. Fortunately, the Xeon PHI runtime framework supports

Table 6.2: Tunable parameters exposed by the dependence-based parallelization

T	**Class**	**Range**	**Semantics**
Work distribution			
P_i	Ratio	$P_i = [0, \hat{\mathcal{U}}]$	Partitioning and Platform Mapping
GPU Platform Parameters			
T^{GPU}	Ratio	$T^{GPU} \in [1, 32] \subset \mathbb{Z}$	Warps per GPU Block
CPU Platform Parameters			
T^{CPU}	Ratio	$T^{CPU} \in [1, \texttt{cores}] \subset \mathbb{Z}$	CPU-Threads

OpenMP, which is used for parallelizing the code. The target plugin reuses our existing OpenMP parallelizer to do so. Lastly, the parallelized code is linked with a runtime environment, which handles the communication, and target region and memory management. The communication protocol is further described in Section 6.2.3.

Despite these unique requirements, the behavior for emitting code deviates only slightly from the other two target plugins. Invoking the target region and transferring the data is a single operation that is performed during the prologue. The epilogue fetches the result data, which simultaneously synchronizes the execution. The target region is running asynchronously on the device and is executed immediately once the data arrives.

6.2.2.4 Target-Specific Tuning Parameters

For a given source region, `aphes` creates a single tuner instance that we refer to as the *tuning context* for the region. For this instance, it inserts a special prologue and epilogue, which are executed before all target plugin prologues and after all epilogues, respectively. In these sections, the runtime environment invokes the `libtuning` tuner. The prologue updates the configuration of all tuning parameters in this context and starts the time measurement, the epilogue stops the time measurement and updates the configuration.

The tuning parameters of a context are created either by the `aphes` driver or the individual target plugins, which can hook into the context and register arbitrary target-specific parameters. Examples are T^{CPU} for the number of OpenMP threads for the OpenMP and Xeon PHI target, or T^{GPU} the number of full warps per block for the CUDA target. For every target platform P_i, the `aphes` driver adds a single parameter $P_i = [0, \hat{\mathcal{U}}]$ for some constant upper bound $\hat{\mathcal{U}}$. The full set of extracted parameters is summarized in Table 6.2.

The parameters P_i simultaneously select the platforms to offload to as well as the amount of work assigned to each one. This is a trick we resort to to avoid having to introduce nominal parameters for mapping partitions onto platforms while at the same time ensuring that the mapping is one-to-one, i.e., a platform is only assigned at most one slice of the work. The objective of these parameters is, for each platform i, to assign it a partition of the program part's loop iteration range $\mathcal{P}_i \subseteq [\mathcal{L}, \mathcal{U}] = \mathcal{R}$, such that $\bigcup_j \mathcal{P}_j = \mathcal{R}$ and $\mathcal{P}_j \cap \mathcal{P}_k = \emptyset, \forall_{j,k} : j \neq k$. Interaction with the autotuner requires embedding parameters into a static numerical range, but $\mathcal{L}$ and $\mathcal{U}$ are dynamic in general. We solve this problem by defining $P_i = [0, \hat{\mathcal{U}}]$ for some arbitrary but fixed $\hat{\mathcal{U}}$. Then, every configuration $C_{P_i} \in P_i$ for the P_i produces a partitioning of the virtual iteration range $\hat{\mathcal{R}} = [0, \max_i C_{P_i}]$. For every $\hat{\mathcal{P}}_i \subseteq \hat{\mathcal{R}}$ we can thus obtain $\mathcal{P}$ by simply shifting and rescaling the interval. To compute $\hat{\mathcal{P}}_i = [l_i, u_i]$ we first sort the sequence of C_{P_i} in ascending order[3]. Let $\sigma(i)$ denote the index of C_{P_i} in the sorted sequence. Then we set

$$u_i = C_{P_i},$$

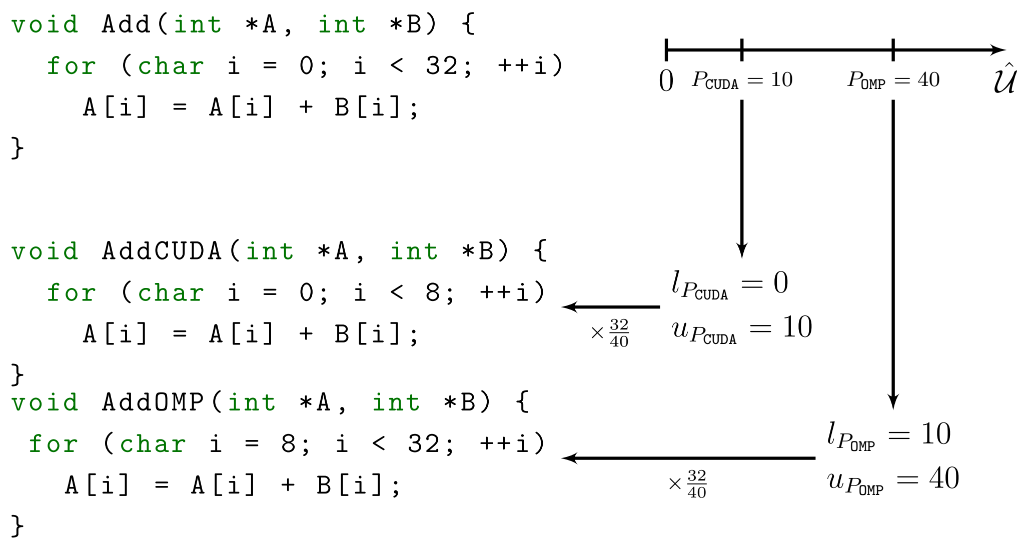
$$l_i = \begin{cases} 0 & \sigma(i) = 0 \\ C_{P_j} & \sigma(i) = \sigma(j) + 1. \end{cases}$$

Figure 6.5 shows an example of this process. $\hat{\mathcal{U}}$ is set to 256, and the original iteration range is $\mathcal{R} = [0, 32)$. The tuner has configured the tuning parameters to $(P_{\texttt{CUDA}}, P_{\texttt{OMP}}) = (10, 40)$. This produces the virtual iteration ranges $[l_{P_{\texttt{CUDA}}}, u_{P_{\texttt{CUDA}}}) = [0, 10)$, $[l_{P_{\texttt{OMP}}}, u_{P_{\texttt{OMP}}}) = [10, 40)$, and rescaling this by the factor $\frac{32}{40}$, we obtain the final iteration ranges $[0, 8)$ and $[8, 32)$ for both targets.

6.2.3 Runtime System

The `aphes` runtime system performs three primary tasks: Managing and asynchronously executing target regions, managing memory allocations and transfers, and interfacing between the parallelized application and the `libtuning` tuner. For the Xeon PHI target [Böh15], the first and second task involve driving the MPI communication with the device-side target binary. The runtime system is shipped

[3]If $C_{P_j} = C_{P_k}$, the elements' relative order is preserved to break ties deterministically



```
void Add(int *A, int *B) {
  for (char i = 0; i < 32; ++i)
    A[i] = A[i] + B[i];
}
```

```
void AddCUDA(int *A, int *B) {
  for (char i = 0; i < 8; ++i)
    A[i] = A[i] + B[i];
}
void AddOMP(int *A, int *B) {
 for (char i = 8; i < 32; ++i)
   A[i] = A[i] + B[i];
}
```

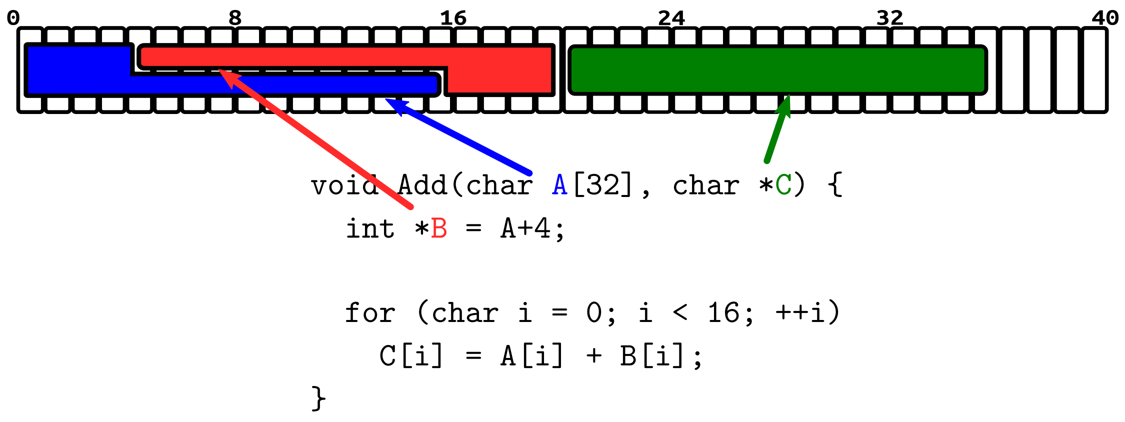
Figure 6.5: Example: Partitioning the work between an OpenMP and a CUDA target. The partitioned virtual index range is mapped onto the concrete range in the input program.

together with `aphes` as a library. The `aphes` compiler links it into the generated binaries automatically.

Target region management on the host side currently only has to deal with CUDA kernels. The `aphes` compiler translates the kernel into CUDA PTX, a CUDA-specific assembly code, and embeds it directly into the parallelized binary. To launch the kernel, the runtime system loads the assembly code and JIT compiles it for the CUDA driver. The result is cached for future executions. Once compiled, the kernel can be launched. For the Xeon PHI target [Böh15], launching the target region and transferring memory is done in a single, five-step process: The host first sends a single region identifier to the device-side binary[4]. Following the identifier, the host first sends all scalar variables, then all memory ranges read by the region. Once this transfer is complete, the device-side binary automatically launches the selected region. Upon completion of the region, the device-side binary returns all memory ranges written by the region to the host-side runtime.

Both for CUDA GPU and Xeon PHI, device-side memory is managed by a custom dynamic memory allocator. The allocator is part of the `aphes` runtime

[4]Additionally there is a special identifier that, sent here, signals program termination.



```
void Add(char A[32], char *C) {
  int *B = A+4;

  for (char i = 0; i < 16; ++i)
    C[i] = A[i] + B[i];
}
```

Figure 6.6: Example: Allocation mapping for three arrays.

and maintains a single continuous chunk of memory. In the prologue, in which the sizes of all required array slices are known dynamically, the allocator is invoked to remap all arrays accessed in the target region into the managed chunk, growing it if required. If host pointers point to overlapping regions of memory, the allocator automatically merges their mappings accordingly.

Figure 6.6 shows an example of this process. The allocator has allocated a contiguous array of 40 bytes. Mapping the arrays `A`,`B`, and `C` into this memory occupies only 34 bytes. Even though the size of `A` is 32 bytes, and the size of `C` is unknown, the loop will access only 16 bytes of `C`, and 20 bytes of `A`, namely once 16 bytes through `A` directly at offset 0, and once 16 bytes indirectly through `B` at offset 4. To compute the mapping, the allocator represents every array A_i as a triple $(Addr_{A_i}, Size_{A_i}, Offset_{A_i})$. $Offset_{A_i}$ is initially 0. Sorting these triples in ascending lexicographic order, subsequent tuples for A_i and A_j are merged if and only if $Addr_{A_i} + Size_{A_i} > Addr_{A_j}$. If the inequality holds, then A_i and A_j overlap in memory. Upon merging, we set $Offset_{A_j} = Addr_{A_j} - Addr_{A_i}$ and $Addr_{A_j} = Addr_{A_i}$. The merged triples can then be trivially mapped onto the contiguous space, growing the allocation if necessary.

Lastly, the runtime system's tuning interface implements interoperation with the `libtuning` library. It maintains a registry of instantiated and running tuners, and provides an interface to that registry to the parallelized program. Finally, it also implements the algorithm described in Section 6.2.2.4 to compute loop iteration bounds for the target region execution.

6.3 Offloading Heuristics and Runtime Predictors

As we highlighted in Chapter 4, the vast majority off approaches to automatic parallelization or automatic offloading rely on heuristics, cost models, and machine and performance models to make tuning decisions. Although the goal of this thesis is to demonstrate the power of black-box online autotuning applied to the problem, the benefits of the models developed in the past should not be ignored. With `aphes` and `libtuning` we combine both techniques. The `libtuning` autotuner allows clients to reject configurations without actually measuring them. In `aphes`, we use this mechanism to bridge between static performance models and autotuning. For this purpose, `aphes` includes a small embedded DSL compiler developers may use to implement rejection rules. An example for such a rule can be as simple as

$$C_{P_i} == 1 \to \texttt{ComputeVolume}_C > \mathit{Threshold}.$$

`ComputeVolume` here is a performance model predicting the number of *dynamic instructions* executed in the source region. Assuming every instruction's execution takes the same amount of time, the compute volume is an accurate estimate of the time required to execute the source region. When the `ComputeVolume` models the target region on the other hand, it additionally depends on the tuning parameters. That allows us to make a-priori runtime predictions for the transformed and tuned program. The heuristic implemented by the simple rule above can intuitively be interpreted as “Only execute this region on the GPU if in this configuration it will perform enough computations”. As a proof-of-concept, we implemented the rule above in the `aphes` compiler.

Besides the `ComputeVolume`, `aphes` embeds multiple similar analytical models into the application during transformation. That includes the number of `load` and `store` instructions executed, and the `ArithmeticIntensity`. The latter describes the ratio between the `ComputeVolume` of the source region and the amount of memory it accesses. The `ArithmeticIntensity` is thus a metric for the number of operations performed per byte of memory transferred.

In general, the analytical models are *parametric*. For example, the number of `load` instructions executed by the source region depends on the number of loop iterations. This number generally depends on the input of the program or the current region. Consequently, `aphes` cannot evaluate the models statically and has to defer that to the program's execution. For every model, `aphes` generates code to compute its value at program runtime. The models are then evaluated at runtime as needed. Additionally, because they are parametric we can exploit the models for a second important purpose. The `ComputeVolume` in combination with the numbers of `loads` and `stores` provide an approximation of the runtime of the parallelized loop. This means that the parameters in these expressions directly influence the loop runtime. Because the parameters by construction are program variables, they are perfect candidates to serve as indicators to hybrid autotuning. Once the models have been computed, `aphes` thus extracts all their parameters and registers them with the autotuner accordingly.

Currently, we support two mechanisms for deriving analytical models from the input program: based on the polyhedral model, or based on classical interprocedural data flow analysis if the source region is not representable polyhedrally. In this case, we count the dynamic instructions of a region recursively. For instance, the instruction count of a loop is the instruction count of its body multiplied by the number of loop iterations. Often, however, this number does not satisfy the static control criterion. That is, it is neither constant nor a provably invariant variable. In this case, we underestimate the instruction count. As a consequence, evaluating the model does not yield an exact value, but a lower bound on the specific count, which is sufficient for our users of the models. We perform this approximation whenever exact counting is impossible, in particular for `for` and `while` loops with undecidable iteration counts, `if` conditionals with different instruction counts in their branches, and calls to functions that cannot be analyzed interprocedurally. If, however, the polyhedral model is applicable to the source, approximating the count is unnecessary and we can determine exact instruction counts. Using `libbarvinok`, we directly obtain the analytical models from the polyhedral description of the source and target regions. This library provides an implementation of Barvinok's algorithm, which counts the number of points in an integer polyhedron efficiently.

Generating code for the analytical model expressions is straightforward. Both the data flow analysis and the polyhedral model produce piecewise polynomial expressions, parametric in the program arguments and, if modeling the target region, also the tuning parameters.

6.4 Summary

This chapter presented the parallelizing compiler component of the APHES framework. It is implemented as an LLVM-based tool that consumes a program in LLVM intermediate representation and produces an executable binary. If parallelization was successful, the resulting program automatically offloads computations to multiple platforms and distributes the work according to directions by the autotuner presented in the previous chapter.

The program analysis and transformation is built upon two techniques: Using the polyhedral model or alternatively using dependence testing and ad hoc transformations. The polyhedral-based transformation first uses the Pluto algorithm in PPCG to optimize the schedule of the transformed loop to maximize parallelism. We then modify the schedule to first partition the work and second map each partition to the target platforms. Partitioning and mapping both are governed by tunable parameters controlled by the `libtuning` autotuner. Lastly, using Polly-ACC we generate code from the transformed schedule. The dependence testing-based parallelization scheme is applied if the polyhedral model fails to represent the loop. Using standard techniques we detect data parallelism interprocedurally and transform parallel loops using multiple platform-specific target plugins. In addition to the partitioning of the work, both the polyhedral parallelization as well as the target plugins register additional platform-specific parameters with the autotuner instance.

Chapter 7

Experimental Evaluation

In this chapter we present the experimental evaluation of the `APHES` framework. We investigate the validity the theses posed in Chapter 1 in three separate experiments. The validation of thesis $\mathbf{T_1}$ comprises two distinct aspects. Thesis $\mathbf{T_1}$ states that cooperative parallelization accelerates programs and outperforms the state of the art. First, we demonstrate that the approach followed by the current state of the art is insufficient to achieve the project goals. The current state of the art in cooperative multi-platform execution are Qilin, CHC, and libHawaii. All three approaches use a linear regression model for the cooperative execution. Section 7.2 presents benchmarks using `APHES`' ad-hoc parallelization to analyze whether the linear regression can accurately represent the cooperative execution problem. Second, in Section 7.3 we use polyhedral cooperative parallelization on Polybench, a widely used scientific benchmark. We present the performance improvements achieved with hybrid tuning, in particular using hierarchical search.

We then analyze the behavior of hybrid autotuning in detail to validate thesis $\mathbf{T_2}$. Thesis $\mathbf{T_2}$ claims that hybrid autotuning improves on the amortization time of the state of the art in autotuning. Our tuning approach combines two ways to obtain a new configuration for a given dynamic context state: Search and prediction. Of the two alternatives, amortization time is predominantly determined by search. Although the quality of the predicted configurations certainly has an influence on amortization time, that particular aspect is covered specifically by thesis $\mathbf{T_3}$. Validating thesis $\mathbf{T_2}$ thus requires analyzing the behavior of the search. The autotuner decides stochastically which of the two alternatives to query. That means that we can analyze both aspects of hybrid tuning in isolation. In Section 7.3 we investigate the convergence behavior of the tuner and compare it to state-of-the-art autotuning. To analyze the amortization behavior, both the configurations found through search as well as the time required to complete a given number of search steps is compared.

In Section 7.4 we investigate thesis $\mathbf{T_3}$. The thesis states that our tuner successfully predicts configurations for unknown context states. For a subset of the benchmarks used in Section 7.3 we analyze and compare the predictions of our models for varying tuning contexts. The contexts are varied by using a set of different tuning kernel inputs generated randomly.

Section 7.5 summarizes the results and reviews the thesis objectives with respect to our findings. In the following section, we begin by introducing the main set of benchmarks used throughout this chapter.

7.1 Benchmarks

The benchmarks used in this evaluation were obtained from the PolyBench/C 4.2 benchmark suite[1]. PolyBench is a collection of small programs for numerical computations with static control flow. Static control is a prerequisite of the polyhedral model. From this suite we used the `blas` and `kernels` sub-packages of the linear algebra benchmarks. From the 13 benchmark programs in those packages we selected 11, which are summarized in Table 7.1. The `symm` and `trmm` were excluded for reasons discussed further below.

The selected benchmarks perform various vector and matrix operations. Although these operations have a limited variance in the data access patterns, they cover an interesting range of computational patterns. On the one hand there are benchmarks with both quadratic computational and spatial complexity. The benchmarks including general matrix multiplication such as `gemm`, `2mm`, or `3mm` have cubic computational complexity and quadratic spatial complexity. Moreover, `syrk` and `syr2k` are particularly interesting because they have triangular iteration domains. Consequently the partitions created by the parallelizer become asymmetrical. It is therefore not only the size of the partition anymore that determines its computational cost, but also the specific iteration range it covers.

For the polyhedral parallelization evaluation in Section 7.3 we made minor modifications to the selected benchmarks to account for simplifications made in the implementation of the `APHES` prototype. These involve two changes. First, we modified the memory allocation routines of PolyBench to use page-locked (pinned)

[1]Available at https://sourceforge.net/projects/polybench/. Last accessed: February 28th, 2019

Table 7.1: The PolyBench benchmarks used in this evaluation

Benchmark	Description
`2mm`	Two matrix multiplications ($D = A \cdot B, E = C \cdot D$)
`3mm`	Three matrix multiplications ($E = A \cdot B, F = C \cdot D, G = E \cdot F$)
`atax`	Matrix transposition and vector multiplication ($A^T AX$)
`bicg`	The BiCG sub-kernel of the BiCGStab solver
`doitgen`	Quantum chemistry multi-resolution analysis kernel
`gemm`	General matrix multiply ($C = \alpha A \cdot B + \beta C$)
`gemver`	Vector Multiplication and Matrix Addition
`gesummv`	Scalar, Vector and Matrix Multiplication
`mvt`	Matrix Vector Product and Transpose
`syr2k`	Symmetric rank-2k operations
`syrk`	Symmetric rank-k operations

memory on the host. This modification drastically simplifies the implementation of the bookkeeping required in the `aphes` runtime system. It allows us to use CUDA's own asynchronous memory transfer mechanisms without having to manage threads for this purpose. Note however that this change affects performance: Asynchronous transfers from and to page-locked memory enable the device driver to perform Direct Memory Access (DMA) transfers without having to copy the data to a page-locked buffer first. Data transfers after our change can thus be faster, because they eliminate a copy of the data. Nevertheless, all reference compilers compared against in Section 7.3 are applied to the modified benchmark versions. The potential performance impacts of the change thus do not invalidate the comparisons and conclusions drawn in this evaluation. The second modification makes minor changes to the loop order in the benchmark kernels. The current prototype implementation only generates a single GPU kernel from a parallel region when using polyhedral parallelization. Some kernels' loop nests involve data dependences which require creating multiple GPU kernels in sequence. To account for the limitation of the implementation, we break the loop nests into multiple regions manually by fusing/fissioning loops and inserting memory fences. Fusing and fissioning are operations also performed by the Pluto optimizer, and the memory fences are no-ops removed during compilation. Nevertheless our change might prevent optimizations that the compiler could apply on the original regions.

However, no missed optimizations were observed and there was no discernible performance difference for the reference compilers compared against in Section 7.3. The modification causes the `aphes` compiler to be unable to find and parallelize a meaningful loop within the `symm` kernel. The kernel was therefore excluded from the evaluation. Finally, we also wrapped the calls to every benchmark's kernel into a fixed-length loop to create a tuning loop. This modification forced us to exclude the `trmm` kernel, because the repeated application of the kernel operation on the same inputs causes the floating point values to denormalize, i.e., assume the special value `inf`. Denormal floats can have a substantial impact on performance [DK06] that varies across platforms. Since our change would thus cause a change in the general behavior of this benchmark, we removed it from our list.

In addition to PolyBench we use two different benchmarks for the analysis in Section 7.2 of the linear models proposed by Qilin, CHC, and libHawaii. The benchmarks are `BlackScholes` and `Binomial`, which were both used in the original articles presenting Qilin and CHC.[2] In total, four benchmark programs are used in both articles, the first of which is `gemm`, which is also used by libHawaii[3] and is already included in the PolyBench benchmarks discussed above. Of the remaining three, only `Binomial` and `BlackScholes` were parallelizable by `APHES`. Both benchmarks compute an options pricing model and are available as CUDA programming examples as part of the CUDA SDK.[4]. They contain a sequential reference implementation which we use as input to `APHES`. The `BlackScholes` benchmark already contains a natural tuning loop: The benchmark computes the Black-Scholes options pricing formula for 4 million options for 512 iterations. The 512 iterations pose the tuning loop. Because the `Binomial` benchmark does not contain such a loop, we added one manually.

7.2 Ad-hoc Parallelization

We analyzed the performance of the PolyBench benchmarks parallelized with `APHES`'s ad-hoc parallelization. Unfortunately, that experiment exposed a major limitation of the ad-hoc parallelization approach. Being limited to transforming outer loops only, there is little parallelism the approach can extract from the

[2]See references [LHK09] and [LRG12]
[3]See reference [RDP14].
[4]https://docs.nvidia.com/cuda/cuda-samples/index.html Last accessed: March 19th, 2019

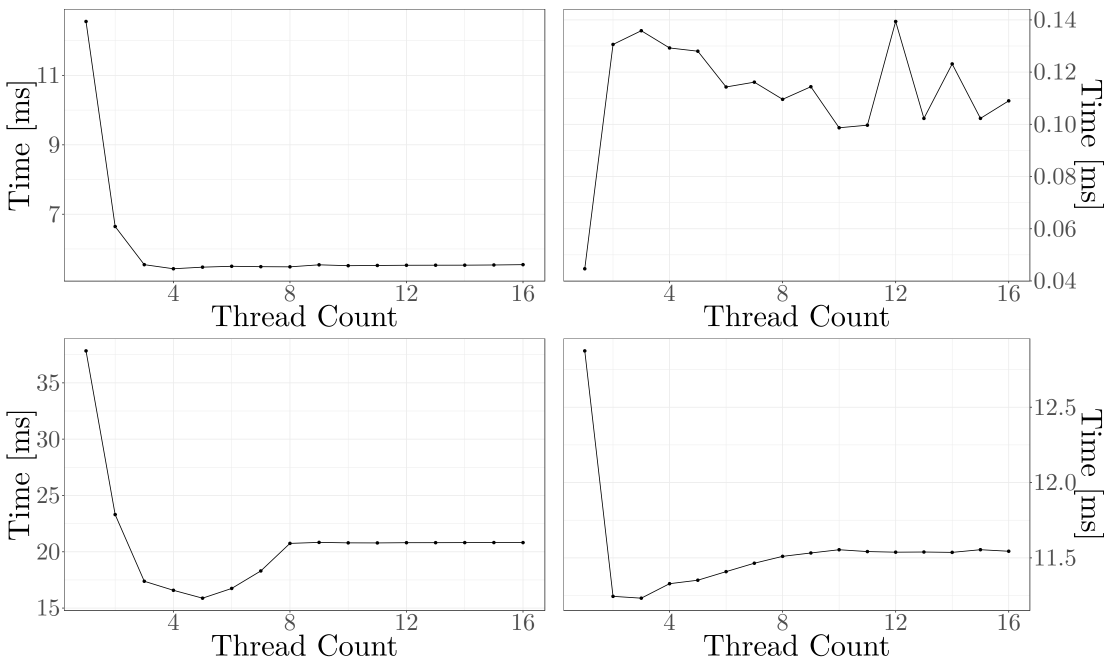


Figure 7.1: Relationship between runtime and OpenMP thread configurations for the four loops of the `gemver` benchmark.

PolyBench benchmarks. Without exception, single-platform parallelization for the OpenMP target achieved the best performance. Nevertheless, we can use the results to investigate the linear modeling approach used by the current state of the art in cooperative heterogeneous execution. Both Qilin and CHC which we introduced in Chapter 4 use a linear regression model to determine the workload distribution across the CPU and GPU platforms. Both approaches differ slightly in how they compute the regression: Qilin samples multiple subsets of the workload on both platforms and measures the execution time, whereas CHC measures only the biggest and smallest possible workload for each platform. The libHawaii autotuner that was also introduced in Chapter 4 uses a control theory approach instead of linear regression. Nevertheless, since the underlying assumption is still a linear model, the following discussion also applies to libHawaii.

Assuming a linear relationship between runtime and workload distribution is sensible when ignoring platform-specific parameters. However, when such parameters as for example the CPU and GPU thread counts are also considered, the PolyBench experiment showed that the linear model is insufficient. Figure 7.1 shows runtime results for the PolyBench `gemver` kernel which has been parallelized for

OpenMP using `APHES`. The `gemver` kernel performs a matrix-vector multiplication followed by a matrix addition. It is composed of four consecutive parallelizable loops. Using the ad-hoc parallelization mechanism in `APHES`, we transform all four loops, but this time for the OpenMP target only. Then, we measure the runtime of the individual loops for thread counts ranging from 1 to 16. The first loop to the top left behaves as one would naively expect: Doubling the number of threads from one to two to four each time roughly halves the runtime. At four threads, the performance peaks. The second loop to the top right behaves contrarily: Because it is so cheap to compute, consuming only a 20th of a millisecond, using more than one thread slows down the execution by a factor of up to $3\times$. The third loop to the bottom left on the other hand shows the most curious progression. The performance is maximal at five threads, after which the performance decreases up until eight threads, after which the runtime is level. There are two important aspects here: First, both using too many or too few threads hurts performance. Second, leveling off after a thread count of eight is an effect of the OpenMP implementation. When exceeding the number of available hardware threads, which is eight on this machine, the `libgomp` OpenMP library used by `APHES` changes its scheduling behavior. This is however an implementation detail of that particular library and not standardized. The effect can thus not be relied upon. The fourth `gemver` loop, lastly, looks again similar to the third, in that it increases performance up until three threads and worsens beyond that number.

Consequently, we see that we cannot rely on linear regression to optimize thread counts alongside the workload distribution. All three cooperative parallelization approaches use a fixed configuration for the platform-specific parameters. With Qilin and libHawaii, defining these parameters is up to the application developer, whereas CHC uses a fixed CPU thread count of $C - 1$ where C is the number of hardware threads. As CHC takes a complete CUDA program as input, the GPU thread configuration is also left to the application developer.

To analyze whether a greedy thread configuration akin to CHC is viable for our approach, we apply the ad-hoc parallelization mechanism to two of the benchmarks also used by Qilin and CHC, `Binomial` and `BlackScholes`. After parallelizing the applications with `APHES`, we measured the runtimes of the resulting tuning kernels for a set of manually defined configurations. Since we configure parameters manually, it is sufficient to set the number of tuning loop iterations of the `Binomial` benchmark to ten, which serves to stabilize our measurements. We ran the bench-

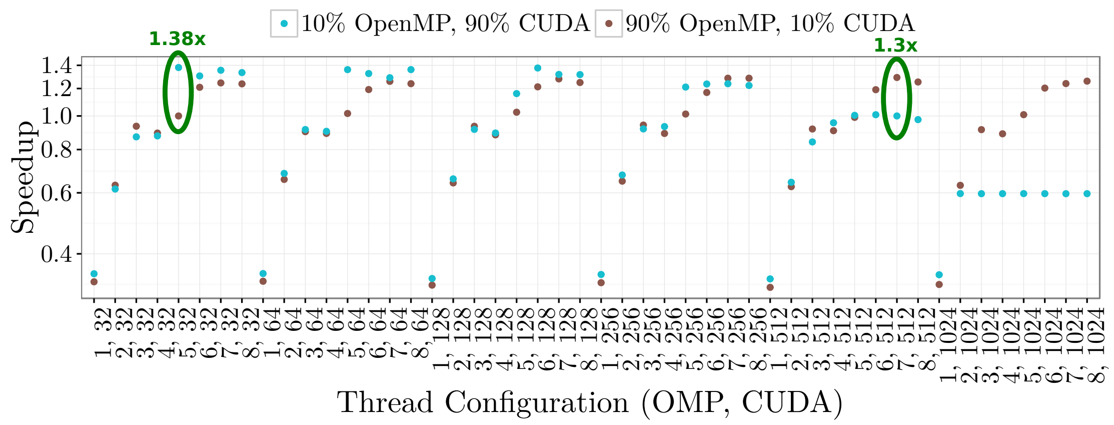


Figure 7.2: `Binomial`: Comparison of greedily selected thread configurations. For the 10%:90% distribution, the speedup reference configuration is $(5, 32)$, for the 90%:10% distribution the reference is $(7, 512)$. The references are the optimal configurations for the respectively other distribution.

marks on a machine equipped with an Intel Core i7 at 3.5 GHz with four physical and eight virtual cores and an NVIDIA GTX 970.

We measure the following parameter configurations. For the workload split, we assign 10% of the loop iterations to OpenMP and 90% to CUDA and vice versa. Those configurations correspond to CHC's approach to building the linear model. On the CPU, the OpenMP thread counts range from one to eight on the CPU. The CUDA blocks contain 32, 64, 128, 256, 512, or 1024 threads, while there is one CUDA thread per work-item (i.e., per loop iteration).

Exploring the configurations exhaustively, we find that for the `Binomial` benchmark for the 10%:90% workload distribution, the optimal configuration is $(5, 32)$, (i.e., 5 OpenMP threads, 32 CUDA threads per block). For the 90%:10% distribution the optimum is at $(7, 512)$ on the other hand. While the second one with an OpenMP thread count of seven could have been produced by CHC, the first one could not. Worse, greedily selecting either can have a substantial negative performance impact on other workload distributions. We show that comparison in Figure 7.2. For both workload splits, the plot shows the speedup achieved by all sampled thread configurations over the configuration that was optimal for the *other* split. The speedup is computed for the median runtimes of the last five tuning loop iterations. The plot highlights the two optima: For the 10%:90% distribution, using the thread configuration $(5, 32)$ over $(7, 512)$ increases perfor-

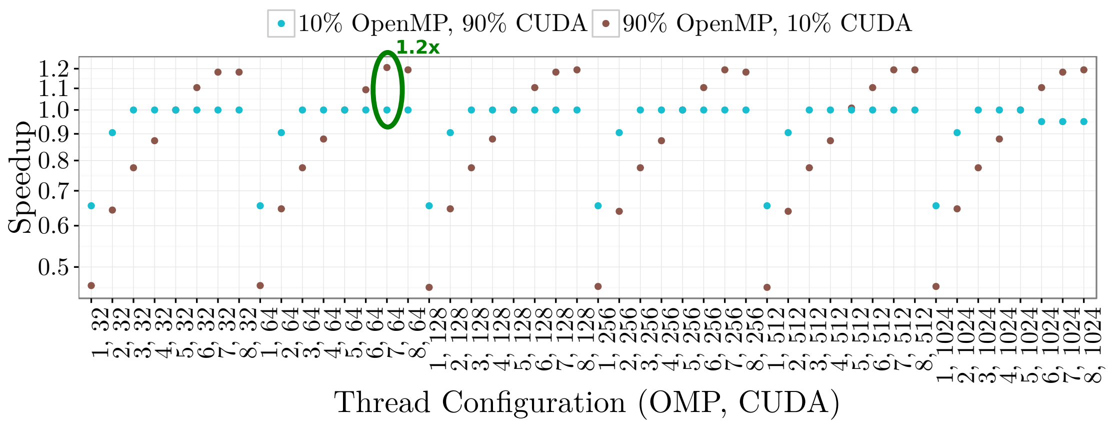


Figure 7.3: `BlackScholes`: Comparison of greedily selected thread configurations. For the 10%:90% distribution, the speedup reference configuration is $(7, 64)$, for the 90%:10% distribution the reference is $(3, 32)$. The references are the optimal configurations for the respectively other distribution.

mance by almost 40%. Vice versa, using the configuration $(7, 512)$ over $(5, 32)$ for the 90%:10% distribution gives us a performance boost of 30%.

We see complementary results for the `BlackScholes` benchmark. For the 90%:10% distribution, the optimal thread configuration is $(7, 64)$. Interestingly, for the 10%:90% distribution, there is no single optimum when accounting for noise. In fact, the majority of configurations using three OpenMP threads or more have the same performance. The absolutely best configuration was $(3, 32)$ which we use in the comparison. The speedup results are shown in Figure 7.3. The maximum speedup for the 90%:10% split is 1.2 over the configuration $(3, 32)$. However, would we have chosen the reference configuration differently, for example like CHC using seven CPU threads, we would see a much smaller performance gain. As we can see in the plot, configurations using seven or eight CPU threads are all within a 3% range of the optimum.

In summary, we have seen that the performance observed for a specific workload distribution is sensitive to the configuration of platform specific parameters. Using exhaustive exploration we found a performance gap of up to 40% that could arise when using a heuristically or fixed greedily determined thread configuration. We also see that even if a linear regression model can predict an optimal workload split when thread configurations are fixed, the result is not globally optimal when they are not. This insight motivates the use of a more sophisticated optimization method such as autotuning. Moreover, the results offer evidence towards the validity of thesis $\mathbf{T_1}$, demonstrating that parameter tuning beyond the workload distribution is necessary to maximize performance.

7.3 Polyhedral Parallelization

In this section we present performance results for a number of scientific benchmarks. We transform the benchmarks using cooperative parallelization and hybrid autotuning. We discuss the performance results in comparison against three single-platform baselines. Subsequently, we analyze hierarchical search in detail and compare it against OpenTuner, a state of the art autotuner.

7.3.1 Performance Results

In the following we present the performance improvements we achieved on the PolyBench benchmarks. For this purpose we compiled four versions of each program. Besides our compiler, we used three reference compilers as baseline: Clang using the polyhedral optimizer (Polly), using the polyhedral OpenMP parallelization (Polly-OMP), and using the polyhedral GPU parallelization (Polly-ACC). For the baselines we use the O3 optimizer level and the default settings for their tunable parameters, such as the OpenMP thread counts. The input sizes of the benchmark programs are fixed per default and can be set during compilation. PolyBench provides five default configurations labeled `MINI` through `EXTRA_LARGE`. We chose the largest configuration for every benchmark. For the `gemm` kernel for instance this setting defines input matrices of 2000-by-2300 and 2300-by-2600. To autotune the programs produced by the `aphes` compiler we set the fixed length of the tuning loop to 100. For the reference programs, the tuning loop was set to a length of 10 to allow for device initialization and cache warm-ups. The median observed kernel runtime is used as the reference result. We ran all benchmarks on a single cluster node with 24 Intel Xeon CPUs at 2.6 GHz and equipped with an NVIDIA P100 GPU.

In Figure 7.4 we show the observed kernel runtime speedups of our autotuned program over the three reference versions. As baselines we use the median runtimes of twenty repetitions of running the reference programs. The runtime samples for the tuned program are the those of the best configuration found within the 100

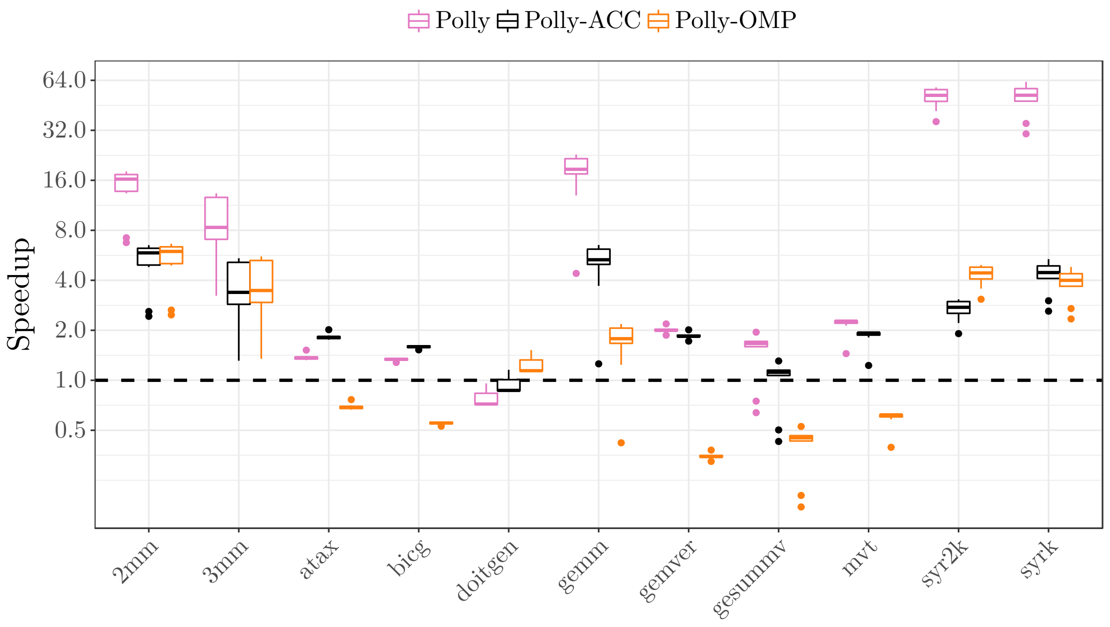


Figure 7.4: Speedup results for the automatically parallelized and tuned PolyBench programs over three baseline compilers: Polly, Polly-ACC, and Polly-OMP.

tuning iterations. The box plots were generated from twenty runtime samples for the tuned program. Both for the tuned and reference versions "runtime" here means the time required to execute the kernel once. The time is reported by the PolyBench programs directly, and includes all overhead of the autotuning, data transfers, and synchronization.

Polly-ACC is outperformed by the solution produced by our approach for all but two benchmarks, as is visible in the figure. In the remaining two the performance of the tuned binary is on par, however. Our maximum speedup over Polly-ACC is 5.9× (median) for the `2mm` benchmark. For `doitgen` on the other hand we cause a minor slowdown of 17%. This is because that benchmark does not benefit from parallelization in general: The kernel is composed of four nested loops, of which the outer two are data parallel. At 220 and 250, the iteration counts of those two loops yield too little parallel work to benefit from GPU offloading: The best performance of `doitgen` is achieved by the Polly baseline. The additional decrease in performance of our approach with respect to Polly-ACC is caused by the extra overhead added by the multi-platform management.

In all cases except `doitgen` our approach outperforms Polly substantially. We achieve a speedup of up to 56× for the `syrk` kernel. For `doitgen`, we incur an 18% slowdown, which again is due to the limited amount of parallelism in that benchmark. Our speedup compared to Polly-OMP is generally the lowest, except for `syr2k` and `doitgen`. We in fact cause slowdowns for five benchmarks. The highest speedup of 5× (median) over Polly-OMP is achieved for the `2mm` kernel. For the `syr2k` and `doitgen` benchmarks, Polly-ACC is able to outperform the OpenMP parallelization.

In summary, we have seen that our approach is able to improve performance substantially over single-target compilers. However it is also capable of introducing large slowdowns. In all cases where this occurred, the reason is that the benchmarks exhibit a small degree of data parallelism coupled with little work per parallel loop iteration. That is the case for `atax`, `bicg`, `gemver`, `gesummv`, and `mvt`. The exploitable data parallelism allows Polly-OMP to accelerate the benchmark, but since the amount of work per work item is small, the benchmarks do not profit from GPU execution. Our tuner maps all partitions onto a CPU successfully. Nevertheless there are still always multiple partitions being mapped, which means we incur multiple times the thread management overhead of Polly-OMP. By setting the partition splits appropriately (i.e., to zero), it is generally possible for the tuner to reduce the number of partitions. The tuner is unable to find this particular configuration because there is a discontinuity of the search space at the boundaries of this parameter's value range. The parameter assumes values from the interval $[0, 1)$. Values slightly different from zero however lead to vastly different program behavior than a value of zero: The number of platforms executing the program changes. Effectively, zero is thus a special value that behaves like a nominal parameters controlling the number of active partitions. Instead, the number of partitions could be controlled by a dedicated tuning parameter, expressing the effect more explicitly to the tuner. We expect that this work-around would eliminate the problematic overhead. A similar problem causes the performance degradation in the `doitgen` benchmark compared to the Polly baseline. The benchmark does not profit from parallelization, but our tuner is unable to execute the original code because the prototype compiler completely replaced it. This could be repaired similarly to above: Retain the original code and introduce another parameter to select between the original and the transformed execution path.

7.3.2 Detailed Analysis of Hierarchical Search

Having seen that our tuner is able to find adequate configurations in the previous section, we investigate its effectiveness and efficiency in detail in the following. Being effective requires that configurations found are as close to the optimum as possible. Efficiency refers to the time the tuner requires to get to its final configuration. Unfortunately there is no ground truth we can compare against. The globally optimal configuration is unknown and exhaustively searching for it is infeasible. Instead, we use OpenTuner [Ans+14] to provide reference data. The comparison between hierarchical search and OpenTuner was first presented in our publication [PGT19], but is presented here in greater detail.[5] OpenTuner is a general purpose autotuning tool written in Python. Application developers tune their program by providing a `MeasurementInterface` implementation. The OpenTuner driver iteratively passes configurations to that interface, for which it is expected to return a measurement value. Configurations and measurements are stored in a central database. OpenTuner's unique feature is its ability to run multiple search algorithms in cooperation. At every iteration, a search algorithm is selected to provide the next configuration. Measurement results are then shared across all algorithms. Instead of a specific search algorithm or implementation the application developer chooses a *set* of them or uses the predefined default. The set can also be generated randomly. Because the tuner is not specifically built as an online tuner, the search is usually run with a time budget or for a limited number of iterations.

To compare hierarchical search and OpenTuner, we use OpenTuner to optimize the PolyBench benchmark programs that were used to produce the performance results discussed above. Unfortunately, OpenTuner cannot be integrated into the parallelized programs like `libtuning` because of its Python implementation that is designed to act as a driver for the program to be tuned: The intended workflow is to produce a configuration in every tuning iteration, to run the program with that configuration, and return a runtime measurement. To optimize the PolyBench programs, we thus use OpenTuner as a driver for the binaries: We implement a Python application containing the OpenTuner tuning loop. In each tuning loop iteration, the application configures and runs the binary produced by our compiler. For this purpose we exploit a feature of `libtuning` that allows loading

[5]The experimental design and data were contributions to the article by the author of this dissertation.

parameter configurations from a file whenever the program is started. The loaded configuration is then kept unchanged for every iteration of the tuning kernel. Using this mechanism, OpenTuner can evaluate a configuration by writing it to the file and executing the benchmark program, which then loads the configuration file and prints out the kernel timings. Three kernel timings are produced by setting the tuning loop length accordingly. We observed that the first timing is always substantially worse than the following ones, which we attribute to the necessary initialization of the GPU device and to cache warming effects. The first timing is hence discarded. OpenTuner uses the average of the remaining ones as the runtime result for the sampled configuration.

While we are forced to take these measures to be able to optimize our parallelized programs with OpenTuner, they cause multiple subtle differences in how configurations are executed and in the performance results the searches can observe for the same configuration. These differences bias the comparison towards OpenTuner, with one exception: In half of the PolyBench programs, the `aphes` compiler generates multiple parallel regions in sequence, each of which is associated with its own instance of the `libtuning` autotuner. When optimizing, each instance observes only the runtime of its associated region. Since OpenTuner is not integrated into the application, however, it is forced to optimize all regions together, thus solving a larger tuning problem. Nevertheless, in the remaining half of the programs there is only one parallel region, allowing for a one-to-one comparison between OpenTuner and hierarchical search. The programs with one parallel region are `doitgen`, `gemm`, `gesummv`, `syr2k`, and `syrk`. Furthermore, two additional major differences exist that strongly favor the OpenTuner search. Firstly, the performance measurements observed by `libtuning` always include all overheads of the search and parameter updates, which for OpenTuner is *not* included in the timing results. We cannot avoid any effect this bias potentially has on the experiment. Secondly, OpenTuner is able to not only avoid penalties of its own overhead, it also avoids some overhead that is caused by the parallel execution within the optimized programs: Whenever a GPU kernel is executed for the first time, it is first Just-In-Time compiled for the GPU device. The cost associated with that is not observed by OpenTuner because the first runtime measurement containing this cost is discarded. With `libtuning`, we do not discard any configurations by default and thus incur the full kernel initialization overhead. We found the effect of this bias to be substantial, in particular when the additional cost is large compared to the actual execution time of

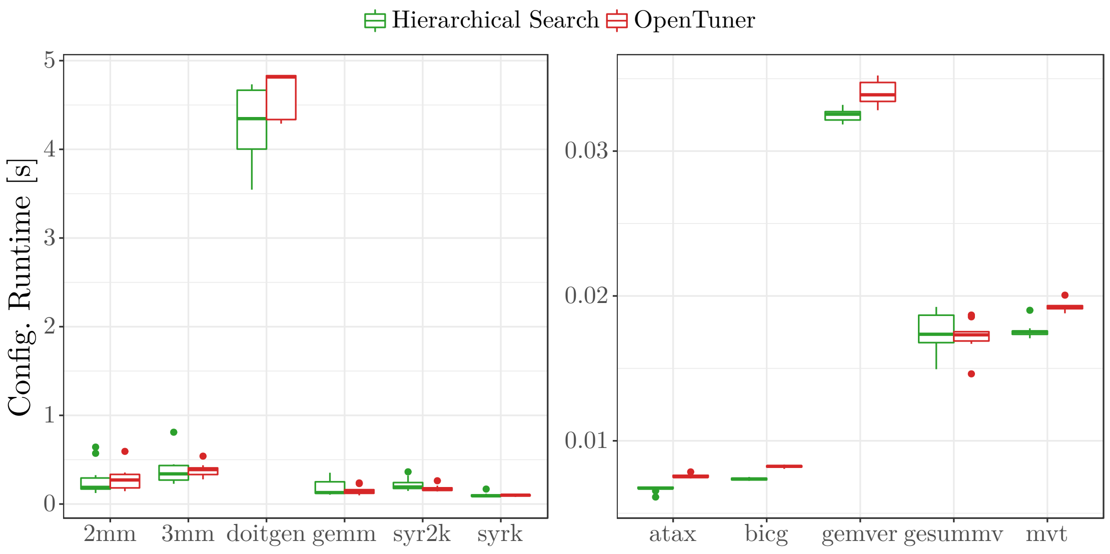


Figure 7.5: Comparison between the configurations found by hierarchical search and OpenTuner after 100 search steps. The plot shows the best runtimes observed during these steps.

the kernel. Therefore, we take steps to prevent this disparity between OpenTuner and hierarchical search in the experiment we conduct: Using the Stabilization mechanism in `libtuning` (see Section 3.2, page 41), we configure the autotuner to sample each configuration twice and discard the first result. As a consequence, we also double the length of the tuning loop when tuning with hierarchical search: For OpenTuner, the PolyBench programs are tuned for 100 iterations, whereas 200 iterations are used for hierarchical search, which corresponds to 100 search steps. In both cases, the search is repeated ten times. To compare the two search algorithms, we focus on two aspects: First, we consider the tuning result produced by either. For either search, we consider as tuning result the best configuration found during the search. Second, we analyze the total time required to complete a fixed number of search steps.

We first consider the best configurations found by the search algorithms. Figure 7.5 depicts the runtimes of the best configurations among the tuning runs of the respective searches: We see that the two algorithms find configurations that for half the benchmarks achieve comparable performance. The data further shows that hierarchical search's tuning results outperform OpenTuner by a small percentage in the `atax`, `bicg`, `gemver`, and `mvt` benchmarks. What these four have

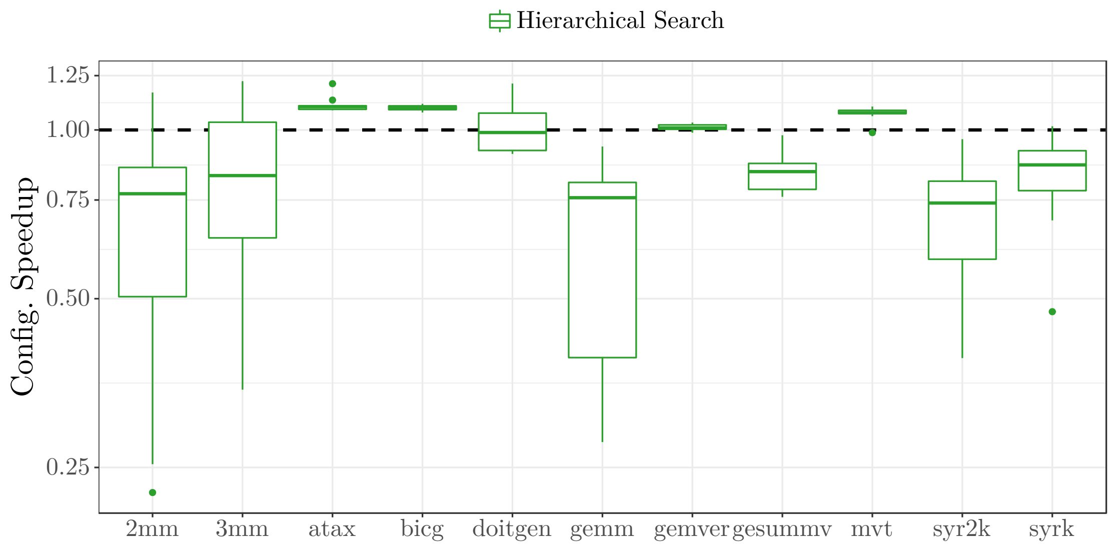


Figure 7.6: Runtime of the configurations found by hierarchical search normalized to the best configuration found by OpenTuner across all tuning runs.

in common is that they are both cheap (on the order of milliseconds or tens of milliseconds) and contain multiple parallel regions in sequence. Because the benchmarks are cheap, the search space contains many bad configurations, in particular those using the GPU. Additionally the multiple parallel regions render the search space more complex for OpenTuner, as discussed above. On the other hand, the only benchmark where the OpenTuner results are overall ahead is `syr2k`. For this benchmark we observed that OpenTuner finds configurations close to its optimum after the first quarter of the search, whereas hierarchical search takes at least twice as long, if it finds those configurations at all. This shows that OpenTuner's more eager exploration is helpful and necessary to find the optimum in this particular benchmark. To visualize the worst-case loss in performance we incur with hierarchical search, Figure 7.6 shows the performance of the search results relative to the absolute best configuration found by OpenTuner across all of its tuning runs. For five benchmarks, namely `atax`, `bicg`, `doitgen`, `gemver`, and `mvt`, the performance is on par or better by a small margin then the OpenTuner result. For the remaining benchmarks, the configurations found by OpenTuner outperform those of hierarchical search. The geometric mean of the relative performance is 0.85 (the median is 0.96), meaning that in the worst case, hierarchical search results are on average 15% slower than those of OpenTuner.

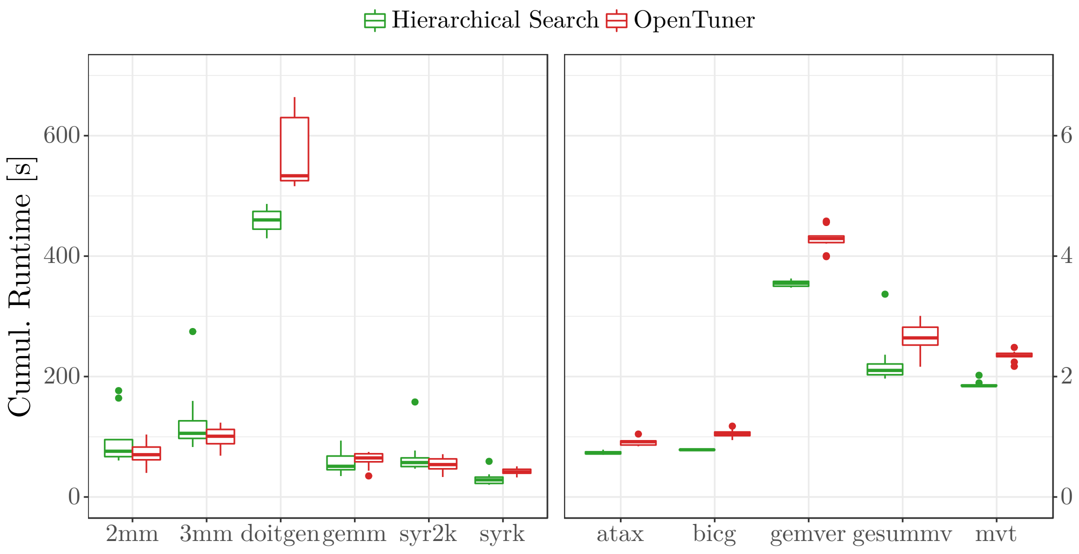


Figure 7.7: The time required to complete 100 search steps using OpenTuner or hierarchical search. Both search techniques require the same amount of time for the `2mm`, `3mm`, `gemm`, and `syr2k` benchmarks. Hierarchical search is faster in the remaining ones.

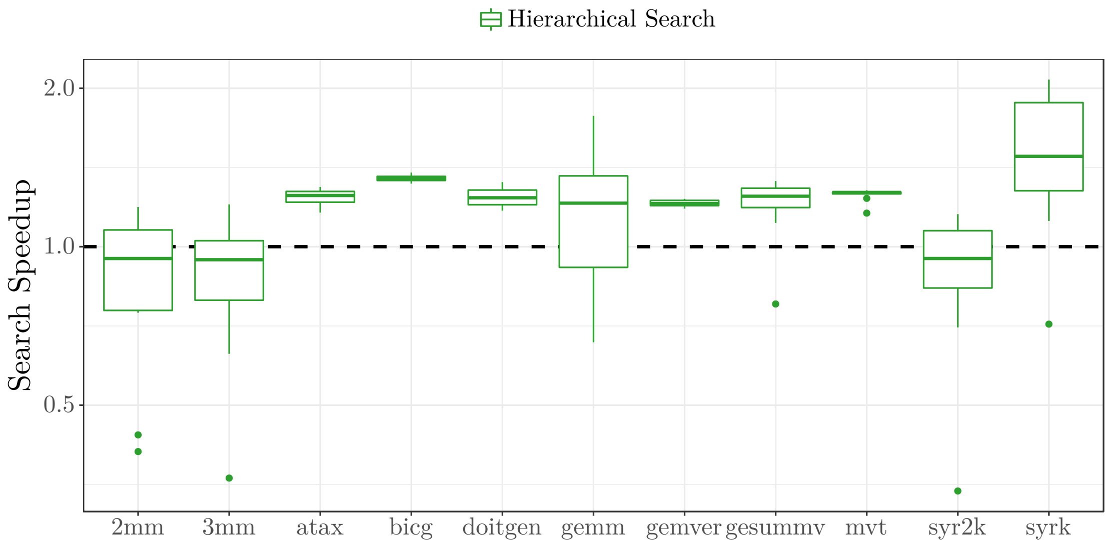
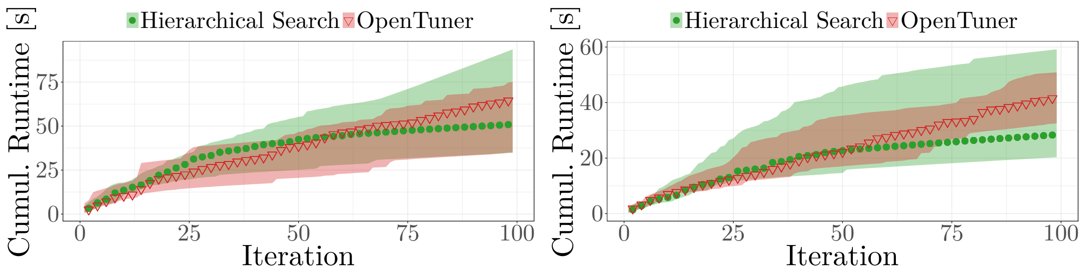


Figure 7.8: The time required to execute 100 search iterations using hierarchical search relative to the average time required by OpenTuner. The median speedup is 1.22.



(a) `gemm` **(b)** `syrk`

Figure 7.9: Execution timeline of 100 search steps with OpenTuner and hierarchical search. Dots are median times required to reach an iteration, the ribbon boundaries are the minimum and maximum. [PGT19]

The key claim of hierarchical search is that it accelerates the search. To assess that claim we compare the cumulative runtime of the configurations sampled by OpenTuner and hierarchical search. Figure 7.7 shows the comparison. The plot shows the cumulative runtime required to complete 100 search steps. It only includes kernel runtimes including all data transfers, the thread management, and for hierarchical search all overhead of the search implementation. The plot shows that hierarchical search completes 100 iterations faster than OpenTuner for seven benchmarks. For `2mm`, `3mm`, `gemm`, and `syr2k`, no acceleration is achieved.

For a better visualization of the relative search performance, we refer to Figure 7.8. This plot shows the speedup of hierarchical search over OpenTuner's average search time. As before, the runtime measurements for our tuner all include all tuning overhead, which is excluded in the OpenTuner measurements. The best performance is achieved in the `syrk` benchmark, for which we reach a median speedup of up to 1.49× and a maximum speedup of 2×. The `2mm`, `3mm`, and `syr2k` benchmarks on the other hand exhibit the worst relative performance with median speedups between 0.94 and 0.95. Overall, the median speedup of hierarchical search compared to OpenTuner is 1.22 (the geometric mean is 1.13). In other words: On average, hierarchical search reduces the search time by over 10% compared to OpenTuner, and by over 30% in the best case.

To illustrate the differences in the search behavior between OpenTuner and hierarchical search, Figure 7.9 shows the cumulative runtimes for two of the benchmarks in greater detail. The selected benchmarks are `gemm`, as a representative for those cases where our search did not accelerate the search, and `syrk` for which we achieved the best results. Both benchmarks contain only a single parallel region and therefore allow for a direct comparison between hierarchical and OpenTuner search. The dots in the plots denote the median time required to reach the iteration on the abscissa, the ribbon boundaries are the minimum and maximum. The data points at iteration 100 are hence the basis for the plots in Figure 7.8. For `gemm` we see at the lower ribbon boundary OpenTuner searches which quickly find configurations with excellent performance and then retain those. After about iteration 16, the lower ribbon boundary is made up of just two search instances, which trade places in iteration 72. Within both instances we see little variance in the kernel runtimes, which indicates that the search ensemble behaves greedily for the majority of the iterations, varying configuration values only slightly, and only rarely makes larger exploratory steps. Contrasting the two searches in the plot with each other we furthermore see the difference in the search behavior: While OpenTuner may take exploratory steps at any iteration, hierarchical search is explorative in the beginning and then becomes more greedy over time. The latter effect is directly controlled by the γ_ϵ parameter, controlling the stepwise decrease of the ϵ in the ϵ-Greedy search. Another interesting effect we see in the hierarchical search curve is the upward bend the upper ribbon boundary exhibits around iteration 70. This effect is produced by a search instance that becomes stuck in a local extremum that it cannot escape, even though that extremum has a worse performance value than configurations sampled previously. Our implementation takes no measures to avoid such situations, but the effect can be easily mitigated in a practical deployment: Once the tuner has converged, the best know configuration can be applied instead of the one sampled last. Comparing `gemm` with `syrk`, we see that the median and lower ribbon boundaries of hierarchical search progress almost identically. The upper ribbon boundary of `syrk` is formed by the outlier visible in Figure 7.8. Here, hierarchical search converges to a local extremum again. Hierarchical search is unable to escape the local extremum, causing the poor cumulative runtime. However, unlike above the runtime at the extremum is not any worse than the configurations sampled before. To mitigate this issue, it is therefore necessary to re-run the search to improve the configuration, but in the context of hybrid tuning that already happens automatically: When the hybrid tuner chooses to explore, a new search is started. Considering the OpenTuner progression for the `syrk` highlights OpenTuner's explorative nature. Although we see both ribbon boundaries and the median level off briefly at around 25 iterations, they shortly after all rise steeply. The tuner leaves good configurations behind to

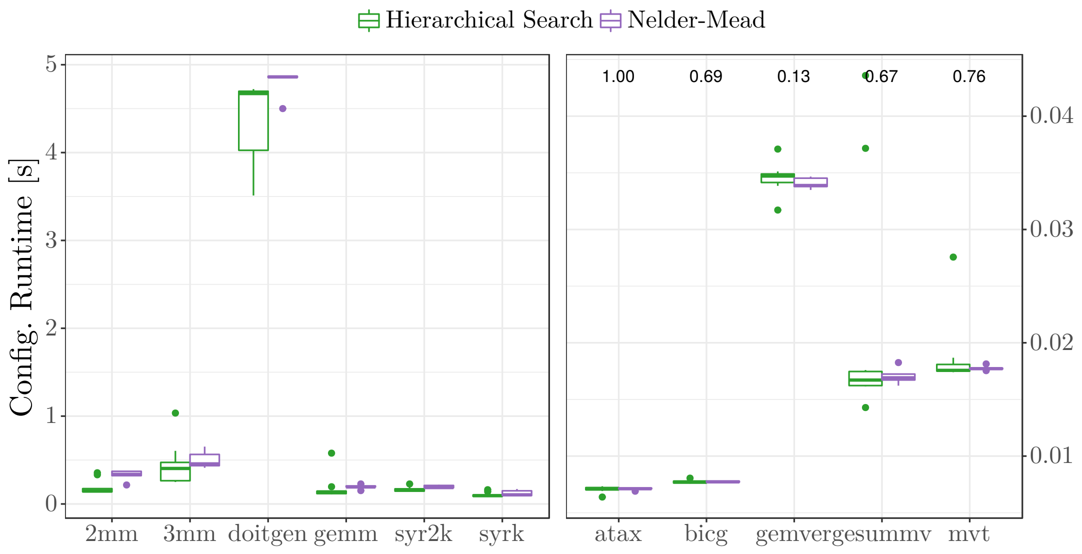


Figure 7.10: Comparison between the configurations found by hierarchical search and Nelder-Mead search 100 search steps. The plot shows the best runtimes observed during these steps.

eagerly explore more of the space, whereas hierarchical tuning more conservatively zeros in on good configurations.

In addition to OpenTuner, a comparison between hierarchical search and our own implementation of the Nelder-Mead algorithm is presented. This comparison provides a direct insight into the effect of the search space decomposition: The Nelder-Mead implementation is also used in the decomposed space in the non-nominal nodes. We can thus see how the dimensionality reduction and the mixed search algorithms simplify the search. A search using only Nelder-Mead corresponds to a hierarchical search where the difference between nominal and non-nominal parameters as well as parameter dependence constraints are ignored. In other words, a Nelder-Mead search using our implementation is identical to a hierarchical search whose search space graph consists of a single non-nominal node. We include this comparison to directly show the effect of our search space decomposition. The Nelder-Mead data discussed in the following was obtained from five tuning runs. Unlike in the performance comparison against OpenTuner, we abstain from using the double-sampling trick for hierarchical search here. Since Nelder-Mead is implemented in `libtuning`, we can use it to optimize the parallelized regions directly. Hence, it is unnecessary to repeat and discard measurements

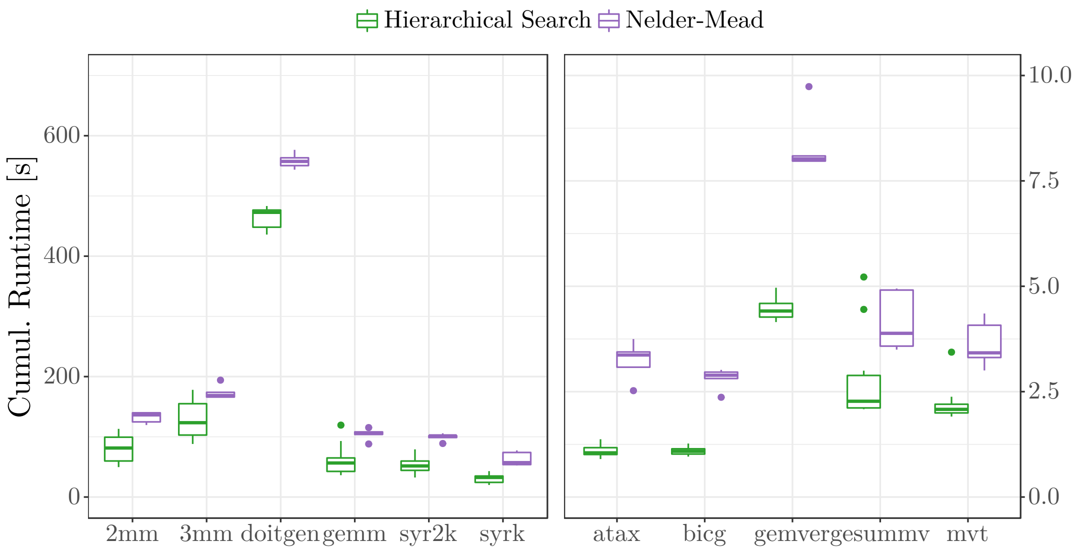


Figure 7.11: The time required to complete 100 search steps using Nelder-Mead or hierarchical search. Hierarchical search completes its steps faster than Nelder-Mead without exception.

during hierarchical searches. Figure 7.10 compares the performance of configurations found by both hierarchical search and Nelder-Mead. Except for four cases, the performance is on par. The exceptions are `2mm`, `doitgen`, `gemm`, and `syr2k`. This difference is most likely caused by Nelder-Mead being unable to complete its search within the allotted frame of 100 iterations: We observed that in multiple of the tuning runs for those benchmarks the Nelder-Mead search has not converged within 100 steps, and that it takes about twice that number for the search to reach a stable point with little runtime variance. Having shown that hierarchical search does not produce configurations worse than the original Nelder-Mead, we also compare the cumulative runtime of the searches. The runtimes are presented in Figure 7.11. Without exception, hierarchical search completes faster than Nelder-Mead. The median speedup over the average Nelder-Mead search time is 1.9 (the geometric mean is 1.8): Hierarchical search reduces the search time almost by half. This result shows that by not using Nelder-Mead to optimize nominal parameters and by reducing the space complexity, the search can be greatly accelerated.

The comparison against OpenTuner revealed that the search space we are facing in the `gemm` benchmark is especially interesting, especially since hierarchical search is unable to accelerate the search. Because of this we used `gemm` as the basis for a

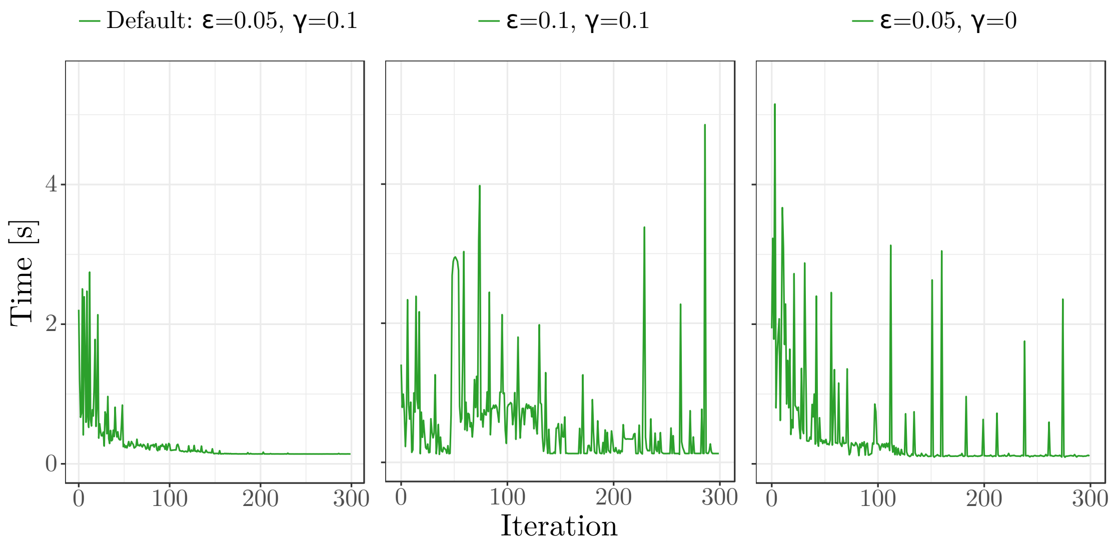


Figure 7.12: `gemm`: Sensitivity of hierarchical search to variation of the hyperparameters: A large ϵ value encourages exploration at the cost of convergence. Disabling the ϵ-decay by setting $\gamma_\epsilon = 0$ produces spikes when sub-optimal configurations are tried more frequently. (Based on [PGT19])

deeper analysis in our publication [PGT19], which we will discuss in the remainder of this section. The following experiments were executed on hardware effectively identical to that used for the experiments above, albeit offering a lower clock speed per core by 100MHz. At the hand of `gemm` we investigate both the influence of hyper-parameters in hierarchical search as well as the effect of the rejection rules.

The most impactful hyper-parameters are those of the ϵ-Greedy algorithm used in the nominal nodes in the search space graph: The ϵ which controls its propensity to explore, and the γ_ϵ which defines the change of ϵ at every tuning step. Figure 7.12 demonstrates the effects of setting adverse values on the runtimes observed during tuning. Note that the traces shown in the plot pertain to single tuning runs. We do not show aggregates of multiple repetitions here, because aggregation smoothes out the spikes and thus hides the performance impact caused by the ϵ-Greedy algorithm choosing random configurations. As a consequence, the shown results and their discussion need to be considered an illustrative example. The curve on the left shows a tuning curve using our default parameter values $\epsilon = 0.05, \gamma_\epsilon = 0.1$. In the middle, we set the ϵ parameter to 10%. Although we can see a macroscopic decreasing trend not unlike in the default case, it is distorted

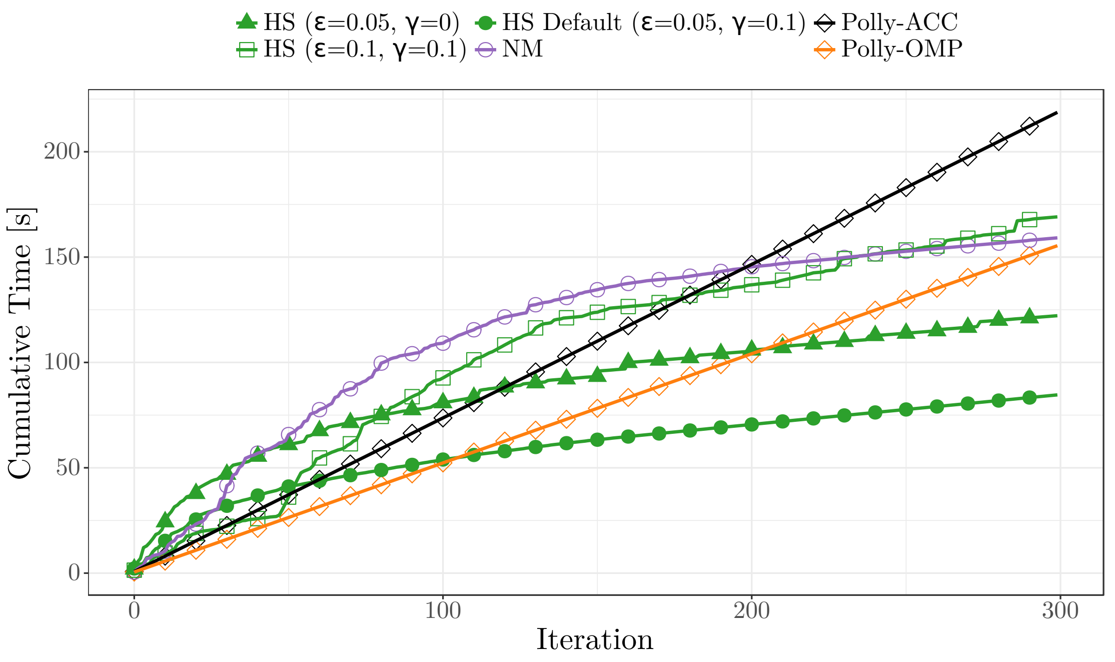


Figure 7.13: `gemm`: Cumulative time traces for the different parametrizations of hierarchical search (HS) compared with classical Nelder-Mead (NM) and the two parallelizing reference compilers. (Based on [PGT19])

by frequent spikes. Because of the larger parameter value the ϵ-Greedy algorithm is both more likely to choose a random configuration and takes longer to reduce the ϵ to turn into a greedy algorithm. Lastly, in the plot on the right, we use the default $\epsilon = 0.05$ but set γ_ϵ to zero. The plot therefore shows the behavior of the original ϵ-Greedy algorithm as it is used in Reinforcement Learning for instance. In essence, we see the same decreasing trend as in the leftmost plot. Unlike the default configuration however we see spikes on the tail.

In Figure 7.13 we show the effect of the parameter settings with respect to cumulative search time. The curves show the accumulated kernel runtimes (including overhead) of the tuning traces in Figure 7.12 over the progression of the tuner. As baselines we included the two parallelizing reference compilers used in Section 7.3.1, as well as our implementation of the classical Nelder-Mead search. The Nelder-Mead curve was obtained by optimizing the parallelized benchmark using only that search algorithm by treating nominal parameters as if they were simple integer intervals. Comparing the Nelder-Mead curve with the hierarchical search curve using our default configuration, we see that our approach completes

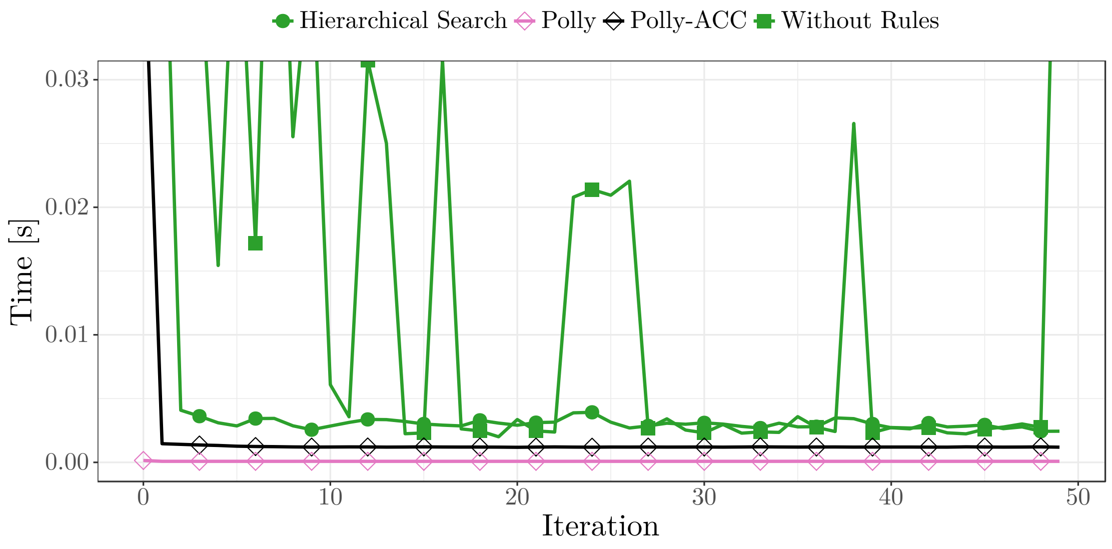


Figure 7.14: `gemm`: Search behavior using small matrix sizes using rejection rules, compared to the reference compilers and search without the rules. In the first iteration, the GPU device is being initialized for both our approach and Polly-ACC, causing the spike. (Based on [PGT19])

100 iterations in half the time. Moreover, we see that our approach outperforms Nelder-Mead with less optimal settings, with the exception of the $\epsilon = 10\%$ case, for which the two searches change ranks at around iteration 250. All search algorithms find configurations that offer speedup over both reference compiler baselines. This demonstrates that our search space decomposition can be beneficial even with unfortunate parameter settings, but the hyper parameters still offer room for optimization.

Finally, we also use the `gemm` benchmark to show the effect of the manual rules which are used to reject tuning configurations in the nominal search space graph nodes. Because the inputs we used for the PolyBench programs were sufficiently large in the experiments discussed so far, no configuration was ever rejected. To enforce rejection, we compiled and ran `gemm` with a smaller configuration. We used the `SMALL_DATASET` default setting. In Figure 7.14 we show the effect the rules contribute. The plot shows single traces using hierarchical search with and without rules. Again, we analyze individual traces instead of aggregates to not average out the negative effects the single traces emphasize. For comparison, we also include the Polly and Polly-ACC baselines. We see that because of the small input size, using the unparallelized kernel version offers the highest performance. Even the single-platform Polly-ACC baseline is substantially faster than our solution. This is because our compiler prototype does not implement a fallback to the original unparallelized kernel. The tuner is forced to always select a parallel version. However, the rejection rules help mitigate this issue. Because of the input size they reject configurations that offload computations to a GPU. Without the rules, the tuner samples configurations which are off the chart here and cause slowdowns of 100× and more. The Polly-ACC curve exposes an interesting detail: The reference compiler adds a similar rejection rule, and it is triggering for this benchmark as well. The Polly-ACC rule is much less sophisticated and much easier to compute than ours. Therefore, the overhead that the Polly-ACC plot shows over the Polly baseline can be seen as a lower bound on the overhead introduced by our rejection rules. The upper bound on the overhead is hard to determine. In the second iteration of the trace in Figure 7.14, which is also off the scale, we identify one particular problem of the rejection mechanism. When a configuration is rejected, the search algorithm produces a new one, which for ϵ-Greedy means drawing a new random sample. The re-sampling continues until an acceptable configuration is found, and causes a slowdown of 600× over the Polly baseline.

7.3.3 Discussion

The results presented in this section have shown that the `APHES` framework is able to automatically accelerate data parallel programs. Offloading to multiple parallel platforms outperforms single-platform and non-parallelizing reference compilers. Moreover, we have shown that hierarchical search is able to accelerate the search for configurations. The automatic reduction of the search space size and the use of individual search algorithms for different parameter classes accelerate the search by a factor of up to two and by 1.13 on average. This acceleration was shown against OpenTuner, a current state-of-the-art autotuner. The configurations found by our approach are on par with those found by OpenTuner. Because of that and because of the acceleration of the search we can conclude that hierarchical search is able to improve amortization time. The rejection rules which aim to avoid sampling predictably bad configurations of the nominal parameters have shown to be successful. Despite that, the rules introduce a noteworthy overhead when the parallel workload is insufficient. Being unable to execute the original unparallelized version of the code poses another limitation of our prototypical implementation. Consequently, our compiler can cause degraded performance when the parallel region operates on too small inputs. For a production deployment our compiler must be able to both fall back to the original code and to handle rule-based rejection more intelligently. A possible solution for the latter would be to identify the parameter values responsible for triggering the rule and exclude the violating values from the continued search.

We conclude that the findings presented in this section substantiate thesis $\mathbf{T_2}$ and the open aspects of thesis $\mathbf{T_1}$.

7.4 Hybrid Autotuning

In this section we present experimental results for an analysis of the predictive component of the hybrid autotuning approach. Because our autotuner is using a stochastic policy to decide between search and prediction, we are able to analyze the two components in isolation. The results we present generalize easily to full hybrid autotuning in production by taking the probabilistic behavior into account. Since we have analyzed hierarchical search in the previous sections extensively, we focus on predictive autotuning in this section. We analyze the predictors with respect to the quality of the prediction, i.e., we compare the performance of the configurations found by either prediction or search. All experiments were run on a machine equipped with an Intel Core i7 at 3.5 GHz with four physical and eight virtual cores and an NVIDIA GTX 970.

In the following sections we first discuss the benchmarks we used as well as our procedure to generate training and validation data for our predictors. Subsequently our experiment results are presented and discussed.

7.4.1 Benchmark Selection

This section focuses on the programs of PolyBench that use a single autotuner instance, namely `doitgen`, `gemm`, `gesummv`, `syr2k`, and `syrk`. The remaining programs all contain at least two parallel regions, each of which is associated with an individual tuner instance. Since we are looking at the behavior of individual tuning procedures, the choice simplifies the analysis and the presentation of results greatly. Moreover, the selected benchmarks are diverse enough to allow for representative conclusions: For `doitgen` and `gesummv`, the heterogeneous parallelization was unable to improve the performance (cf. Section 7.3.1). For `gemm`, `syr2k`, and `syrk` the acceleration was excellent on the other hand. In the `doitgen` benchmark our tuner was also unable to achieve a substantial speedup over the OpenTuner search (cf. Section 7.3.2). Therefore, the selected benchmarks capture the extremes.

7.4.2 Experiment Setup

The aim of hybrid autotuning is to improve online autotuning in the face of a dynamic tuning context. To create a dynamic context, we subject the PolyBench benchmarks to different sets of inputs. The input sets further need to contain both inputs that are similar and inputs that are different. To generate the input set, we select inputs randomly with the help of Latin Hypercube sampling[6]: Each benchmark defines several default inputs ranging from `MINI` to `EXTRA_LARGE`. These span a cuboid, which we subdivide into 20 sub-ranges along every dimension. For `gemm`, for instance, the cuboid is three dimensional (for the row and column numbers of the multiplied matrices). There is only a single input for `gesummv`. From the 20 sub-ranges we select five according to Latin Hypercube sampling. Within each selected sub-range we then select 20 points at random. This scheme thus produces 100 inputs for each benchmark, out of which twenty at a time are geometrically close to each other.

Using all benchmarks and inputs we compute search-based tuning baselines to serve as training and validation data. For every benchmark and input, we use hierarchical search five times to optimize the parallel execution over 100 tuning iterations. The baseline data comprises every sampled configuration and its benchmark kernel runtime. The best configuration observed during the five tuning repetitions also serves as a virtual performance roofline. Of course that is only an approximation of the true roofline, which we cannot know without an exhaustive search.

To be able to analyze the quality of the prediction independent of the implementation of the predictor, we perform the experiment similar to the OpenTuner comparison in Section 7.3.2: The predictors are implemented as a Python program external to `libtuning` and function as drivers for the benchmark programs. After

[6]Latin Hypercube sampling [Par94; MBC79] creates space-spanning samples by evenly subdividing every dimension of the sampled space and then placing samples so that there are never two within the same subdivision along any space axis

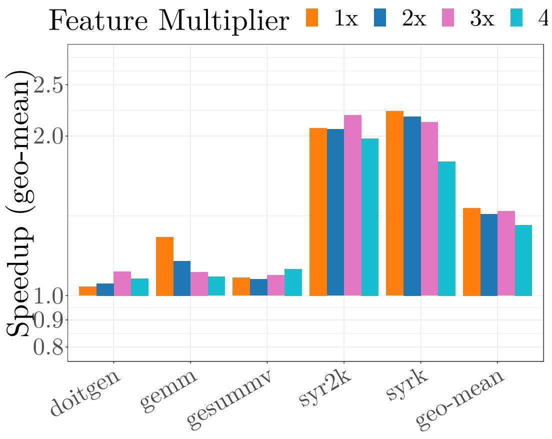
training based on the baseline training data, the predictors produce configurations for the test inputs and execute the benchmark programs using those configurations. As in the OpenTuner experiment, the benchmark programs' tuning loop lengths are set to a fixed value of five. The first two runtime measurements of the PolyBench kernels are discarded and the average of the remaining three is used as the benchmark result.

To train and evaluate the predictors we apply ten-fold cross-validation: The input set for every benchmark is shuffled and then split into ten equally sized partitions. Each partition is used as a test sample while the remaining nine are used for training. We repeat this process five times. In total we thus train and validate 50 times. Every training iteration is based on 90 inputs, whereas validation is based on 10 inputs, producing 500 data points per benchmark and predictor. For validation, we further use the best baseline configuration for each input as reference.

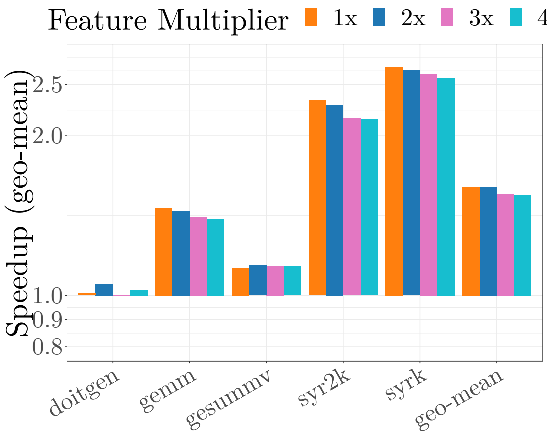
7.4.3 Results

In this section we compare three different variants of the predictor component of the hybrid autotuning approach. The first is the tabular Nearest-Neighbor (NN) predictor. For every benchmark and input its table records the best-performing configuration. To make a prediction for an unknown input, it returns the table entry for the known input that is closest to the unknown one with respect to Euclidean distance. The remaining two predictors are variants of the Greedy-GQ function approximation algorithm. They differ in the features modeling the inputs and configurations. The first one, which we henceforth call RBF-Model, uses Radial Basis Functions (RBF) as features. The second one, called NRBF-Model, uses the normalized variation of the RBF features.

The first step to construct the function approximation models is to define the features. Recall the general definition of the RBF: $\varphi_i(x) = -\omega||x - \mu_i||$ where $x = (s, a)$ is the concatenation of the input and parameter configuration vectors (or state and action vectors in RL terminology). The degrees of freedom are the number of features to use, the centers of the individual basis functions, and their widths. The NRBF features expose the same degrees of freedom. We set the width to 1 for all i. For the centers, we resort to Latin Hypercube sampling again to place the basis functions quasi-randomly in the space. Because the features quantify the inputs and the parameter configuration the number of features should not be



(a) Speedups of the RBF-Model over searching for 100 iterations.



(b) Speedups of the NRBF-Model over searching for 100 iterations.

Figure 7.15: Performance results for the RBF- and NRBF-Model for varying features counts.

fixed. We define the number therefore to be a multiple of the number of inputs and parameters. As multipliers we chose 1×, 2×, 3×, and 4×.

The effect of the different multipliers is shown in Figure 7.15. The plots show the geo-mean speedups of the predictor over the hierarchical search. Speedup here is with respect to the time required to complete the 100 iterations of the search baselines. For the search, that time is the sum of the individual kernel runtimes per sampled configuration. For the predictor, that time is 100 times the runtime of the predicted configuration. Interestingly we see that there is no clear winner among the different feature sets across both models. For the RBF-Model, the feature set with the best overall average, given by the 1× multiplier, achieves a speedup of 1.46, which is 2 percentage-points ahead of the runner-up. For the NRBF-Model, the best feature set is created using the 2× multiplier. With a speedup of 1.6 it outperforms the second best set by a margin of only 0.09 percentage-points. For the following analyses we use the respective best feature set in both models.

Having determined an appropriate feature set for the function approximation models, we now compare the different predictors. In Figure 7.16 we show the speedups of the three predictors over hierarchical search for the 100 iteration frame of reference. As expected, the NN predictor achieves the highest acceleration of 1.93 over tuning. In comparison, the RBF- and NRBF-Models achieve 1.46 and 1.6 respectively.

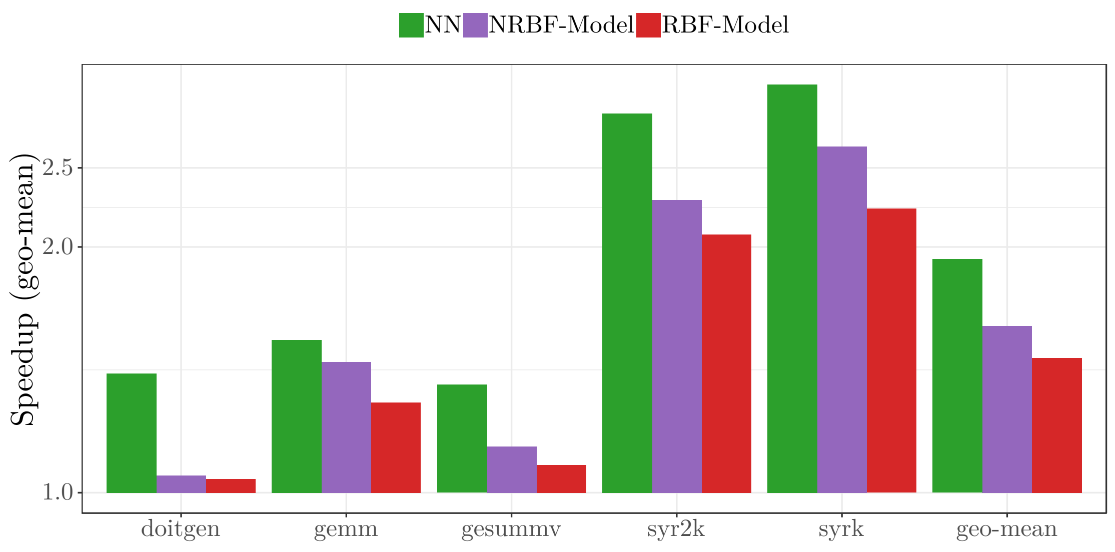


Figure 7.16: Speedup of using the predicted configuration for 100 iterations over performing search-based tuning

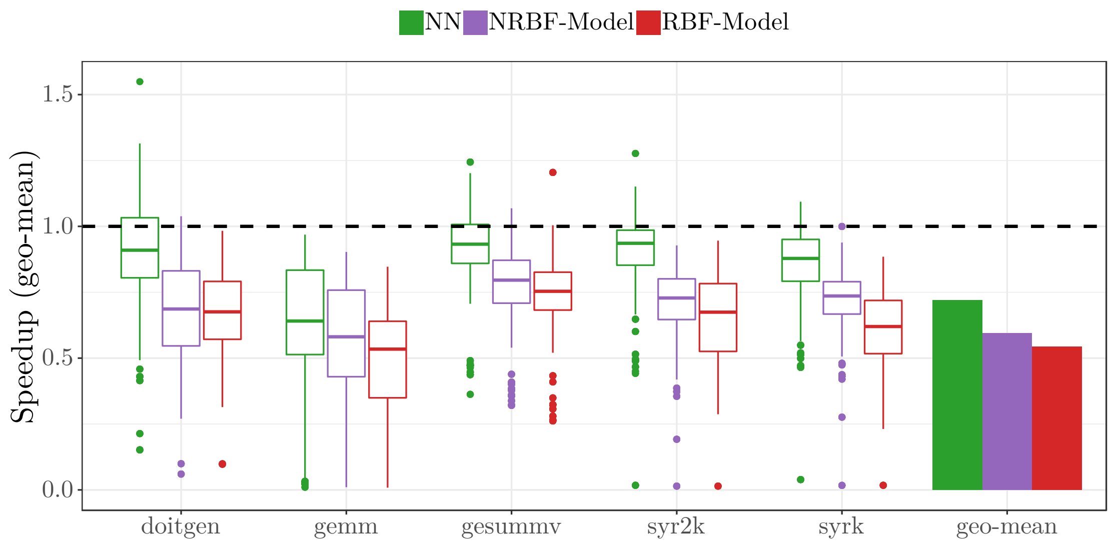


Figure 7.17: Speedup comparison between the prediction results and the best search results.

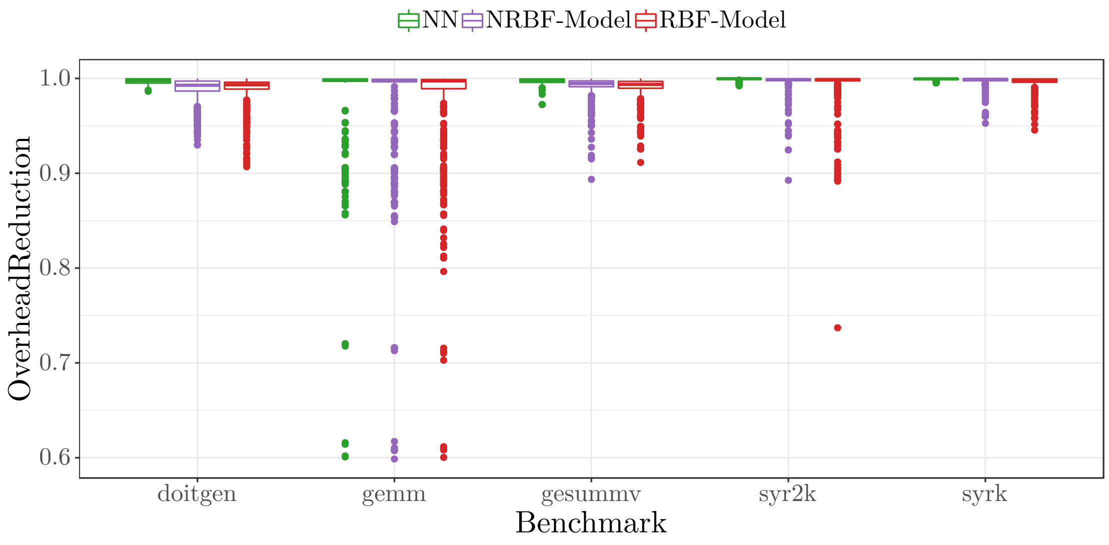


Figure 7.18: Reduction of the search overhead using hybrid tuning.

To assess the quality of the predictors we further compare the prediction results with the best search results. In Figure 7.17 we show the speedup of the predicted configurations over the best configuration found during searching. The vast majority of predicted configurations is slower then what the autotuner found using the hierarchical search. Interestingly, there is a small number of points greater than one, which refer to predictions that produced a configuration actually faster than those found during searching. On average, the NN predictor reaches 71.9% of the search result's performance. The RBF- and NRBF-Models fall behind with 54.4% and 59.6%. While this indicates that we can find substantially better configurations using searching instead of prediction, it does not yet show the benefit of hybrid tuning. The primary goal of our technique is to reduce the *overhead* introduced by search-based autotuning. To evaluate this reduction, we define the overhead as the increase in cumulative runtime compared to using the best known configuration for 100 iterations. Note that this is a different baseline than the one used in Figure 7.17, which was selected only from the configurations seen during the searches, not during prediction. Otherwise, when the predictors find a better configuration than what searching produced, the overhead would be negative. The results we show here are thus slightly biased towards the search, because they explicitly exclude the predictor's ability to find better results than the search. Figure 7.18 shows the overhead reduction metric. We see that even in the worst case

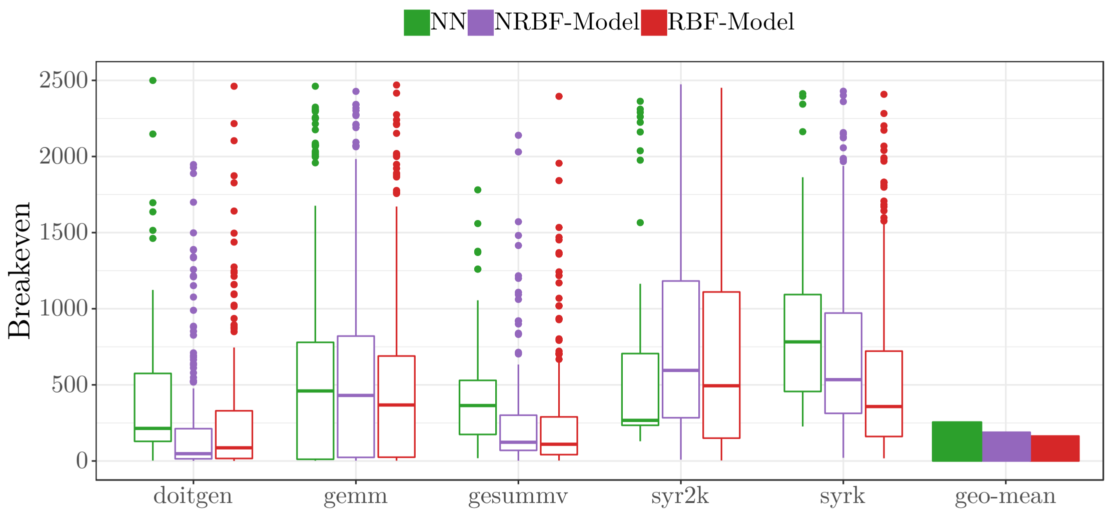


Figure 7.19: The tuning iteration at which the better configuration found by search-based tuning outweighs the overhead reduction gained by prediction-based tuning.

(`gemm`), the overhead reduction is 60%. The geo-mean reduction even is above 98.7% and the median is above 99.6%: Half the time, hybrid autotuning eliminates the entire overhead search-based tuning would incur over 100 iterations, even though search-based tuning finds better configurations.

This conclusion however is only valid for the frame of 100 iterations we looked at: After 100 iterations the search has converged to a configuration that, as we have seen, is generally better than that of the predictor. As the number of iterations considered in the frame of reference goes to infinity, the relative overhead of the search goes to zero. Therefore, there must be a specific number of iterations after which the overhead reduction becomes negative. That means, at some point the better performance of the configuration found by the search will outweigh the benefit of the predictor. If we assume for simplicity that after the 100 iterations the cumulative runtime grows linearly we can compute that break-even point. The cumulative runtimes of the search-based tuning and the prediction-based tuning can then be approximated as linear functions. Define $T_{tuner}(i) = t_{tuner} \cdot i + T_{tuner,100}$ for all tuning methods, where t_{tuner} is the runtime of the (converged) tuners' configuration and $T_{tuner,100}$ is the cumulative time taken for the first 100 iterations. Then, the break-even point is the iteration at which the linear functions of the search-based tuning and the prediction-based tuning intersect. We show the result of this analysis in Figure 7.19. The plot includes only roughly 46% of the data the previous analyses were based upon. The majority (45.3%) was removed because the break-even point was negative: With the exception of five data points, the configuration found by the search was never able to outweigh the overhead reduction. For those five however, which stem from the `doitgen` benchmark search using the RBF- and NRBF-Models, search was already faster than prediction over the 100 iterations. The remaining 8.6% were removed for presentation purposes, because their break-even point was in excess of 2500. The median of those 8.6% is at 4933 iterations, the average is at 37930.39 iterations. The 35% of the data shown here thus pose those data points where hybrid tuning performs worst. The geo-mean break-even point for NN is at 256 iterations. For the RBF- and NRBF-Model it is at 164 and 189, respectively. Including also the data points that were removed for presentation purposes, the geo-mean break-even points are 606, 263, and 306, respectively.

7.4.4 Discussion

In this section we presented a performance evaluation of our hybrid tuning approach. The results have shown that the prediction component of the hybrid tuner can reduce over 99% of the overhead of performing a search for 100 iterations. On average, this means prediction can accelerate these 100 iterations almost 2-fold. The nearest-neighbor predictor has shown to outperform the function approximation predictor by a large margin in all experiments. Although the results for the RBF- and NRBF-Models were positive nonetheless, their benefits in space and runtime complexity do not outweigh the performance difference.

The predictors produce slower configurations than search-based tuning. For NN, the average slowdown was 28%, for the RBF-Model it was 46%, and 40% for the NRBF-Model. Therefore, we found that there is a limit to the overhead reduction that prediction-based tuning can provide. When the tuning kernel is repeated often enough without any change in the context after the search has converged, the search overhead and thus the potential for reduction go to zero. At some point, searching will then provide better overall performance since it is able to find better configurations. For the benchmarks we analyzed, we saw that this occurs in 54.2% of the cases. For those, the break-even was on average above 263 iterations. For the NN predictor it was on average as large as 606 iterations.

These results substantiate our thesis $\mathbf{T_3}$: The predicted configurations are adequate at 72% of the performance achieved by search, but it takes over 600 iterations on average for the search to amortize against the prediction result.

7.5 Summary

In this chapter we evaluated the performance improvements the `APHES` framework offers. The results have shown that our hybrid tuning approach can be used to optimize automatically parallelized programs for cooperative heterogeneous execution. Using hierarchical search, hybrid tuning is able to effectively explore the search space and to exploit a model trained online during the search.

We have shown that autotuning is necessary to enable optimized cooperative heterogeneous execution. Using the performance model employed by the most closely related prior research, we have shown that the model is too simple to provide adequate results. In the presence of additional tunable parameters the linearity assumptions of the model are violated.

With hybrid tuning, programs parallelized by our `aphes` compiler have shown to outperform two state-of-the-art single-platform parallelizers. Compared to current state-of-the-art autotuners, the hierarchical search algorithm reduces the search time by over 10% on average, and up to a maximum of 30%.

Exploiting the models which are automatically trained while exploring, hybrid tuning reduces the impact of the search overhead further. The models produce a configuration at the first tuning iteration based on indicators that describe the current application and system state. Known states can thus be recognized. Using either a tabular nearest-neighbor approach or function approximation, adequate configurations can be predicted for unknown states.

During the evaluation, however, the function approximation approach has shown weaker performance compared to the tabular approach. While the tabular predictor is more expensive to store and to query, its predictions exhibit better performance by up to 47 percentage points on average.

Chapter 8

Conclusion and Outlook

In this dissertation we presented the APHES framework for automatic parallelization for heterogeneous systems. To leverage multiple parallel platforms simultaneously, parallelized programs automatically distribute data parallel work. The work distribution as well as platform-specific tunable parameters are controlled by libtuning, our novel online autotuning library. Parallelized programs are automatically instrumented to interact with the autotuner by the aphes compiler. Together, libtuning and the aphes compiler form the APHES framework.

With libtuning, we have designed a general purpose online autotuning library. It implements our novel tuning technique, hybrid autotuning. Hybrid tuning combines empirical search with online learning. Models constructed while searching can then be *exploited* to select configurations for given program inputs and application and system states. Inputs and states are quantified through indicators, which are metrics defined by an application developer or, within the APHES framework, by the compiler. Changing indicators point to a change in the inputs or in the application or system state. When changes occur, the tuner autonomously chooses whether to explore the space or to exploit the model.

For exploration, the libtuning autotuner uses a novel search algorithm we call hierarchical search. Hierarchical search decomposes the search space by exploiting inter-parameter dependencies and when parameters are nominal. Reducing the high-dimensional search space to several spaces of smaller dimensions simplifies the search in each. Moreover, every space can be explored using different search algorithms. In particular, hierarchical search uses the ϵ-Greedy algorithm to optimize nominal parameter spaces, and Nelder-Mead in the non-nominal ones. For the nominal spaces, libtuning further supports retrieving the next configuration without measuring the last. That enables avoiding bad configurations that can be identified based on domain knowledge a-priori.

The `aphes` compiler is a parallelizing compiler targeting data parallel regions of input programs. The regions are parallelized for multiple parallel targets such as OpenMP or CUDA. It implements two transformation pipelines. The main pipeline is based on the polyhedral model. The polyhedral model is a mature analysis and transformation framework for loop nests. In particular, it supports data locality optimization and dependence testing, which enable automatic parallelization. The `aphes` compiler uses the model to generate code for OpenMP and CUDA, and to distribute the workload across the platforms. The `libtuning` autotuner is used to optimize the distribution of work as well as platform-specific parameters such as OpenMP threads, the amount of work per GPU thread, or whether to use the GPU's shared memory. The compiler automatically extracts indicators from the program by identifying variables that influence the amount of work processed in the parallelized loop. Using the extracted indicators, hybrid tuning can build the prediction models.

Because the polyhedral model has restrictive requirements on the loops it can analyze, we have implemented a second fallback pipeline. The second pipeline provides limited ad-hoc outer-loop parallelization targets based on the program dependence graph. Interprocedural analyses and less restrictions on the control flow enable transformation of programs that are not representable in the polyhedral model. The limitation to outer-loop parallelization however reduces the possible acceleration.

Our experimental evaluation has shown that with the `APHES` framework, automatic multi-platform parallelization is possible. The generated programs outperform programs parallelized with two state-of-the-art single-platform compilers. However, the experiments also exposed the approach's limitations: Outer-loop-only ad-hoc parallelization cannot offer significant speedups in general. Multi-loop parallelization is necessary, but the polyhedral model is subject to restrictions which impede its applicability. Hybrid tuning has demonstrated to reduce the time required for exploration by over 10% on average and to build adequate models. The best performing model, a tabular nearest-neighbor approach, has shown to shed at least 98% of the overhead an exploration would incur. Function-approximation models which are cheaper to store and query, reduce over 84% of the overhead. These results demonstrate that `libtuning` provides an autotuner that enables always-on online tuning. Because of the accelerated search and the model predictions, it poses a tool that can be deployed with applications into production environments.

In the following, we revisit the objectives of this dissertation and discuss the theses we posed in Chapter 1. Lastly, we propose future directions and research questions that arise from our findings.

8.1 Thesis Objectives

In this dissertation, we evaluated our prototypical implementation of the `APHES` framework to investigate the theses posed in Section 1.2.

Thesis $\mathbf{T_1}$: Optimized cooperative parallelization accelerates programs, exceeding the performance of single platform parallelization. Using polyhedral model-based parallelization, our benchmarks have shown a substantial increase in performance compared to two state-of-the-art single-platform parallelizing compilers. We were able to achieve speedups of up to $6\times$ for both references. However, for benchmarks with a limited amount of computations our transformations slowed down the execution. The cause of the decrease in performance was the management overhead for the parallel execution. We did not allow our autotuner to execute unparallelized or single-platform code, which would eliminate this particular problem. The ad-hoc parallelization method was unable to produce substantial speedups. Because it is restricted to transforming outer loops, the amount of parallelism the method can exploit is severely limited. With ad-hoc parallelization, we were able to demonstrate that the performance models used by current state-of-the-art multi-platform parallelization tools are insufficient when additional platform specific parameters are taken into account.

Thesis $\mathbf{T_2}$: Hybrid autotuning determines configurations of comparable quality with better amortization time than the state of the art. Our experiments demonstrated that hybrid tuning and hierarchical search in particular are able to reduce the search time by up to 30% and by over 10% on average. Because of the reduced complexity of the search space, hierarchical search finds an adequate configuration in a shorter time than state-of-the-art autotuning algorithms. The configurations found by hierarchical search are still on par with other autotuners. Although the prediction component of hybrid autotuning produces worse configurations than the search, it does not require sampling configurations and avoids that particular overhead. The amortization time is thus improved compared to the state of the art.

Thesis T_3: Hybrid autotuning predicts adequate configurations for given inputs based without negatively impacting amortization time. The evaluation has shown that both the nearest-neighbor and function-approximation predictors are able to avoid large portions of the overhead that emerges during heuristical searches. The configurations produced by prediction are adequate and they achieve up to 72% of the performance of configurations found through searching on average. However, although the performance is worse, it takes up to more than 600 iterations on average for the search to make up the difference. Only after that break-even point any negative impact of the prediction result on amortization time becomes apparent.

8.2 Outlook

The positive results we were able to achieve with our approach give rise to additional research questions worth exploring. In particular, our autotuner has demonstrated to be usable for always-on tuning scenarios. But to offer the best possible performance results, great care is still required from the application developer. That burden on the developer can and should be lightened. Similarly, we have demonstrated the power of using prediction models along heuristic search in autotuning. Especially in light of the recent interest and advances in machine learning, we believe that our approach can grow beyond its current abilities.

Automatic Indicator Extraction Currently, indicators need to be manually registered with the autotuner. When using the `aphes` compiler to instrument an application for tuning, this process can be automated because the compiler has a good understanding of performance-critical program properties. For other applications, however, there is no automatic solution, which bears the potential for future research: Using program analysis, can an automatic tool extract indicators from programs instrumented with the tuner?

Automatic Parameter Dependencies Like the indicators, parameter dependencies must be stated explicitly by the application developer. Not only is this tedious, it can also introduce errors: If a parameter dependency is defined that is incorrect, the autotuner's search can produce much worse results. There is room for automation: The autotuner is able to observe the relationship between parameter configurations and runtimes. Is it possible to deduce any dependencies automatically from observations, potentially even ones the developer was unaware of? Can this analysis be realized without sacrificing the benefits parameter dependencies provide?

Portability of Hybrid Tuning Models Hybrid autotuning produces prediction models for indicators that currently represent only the dynamic application state. Obviously, indicators can be included that additionally model static and dynamic system properties, as well as other static application properties. Potentially, this opens up the possibility to create *portable* models. Can models be trained on one system and then be exploited on a different one? Can models make predictions across different applications?

List of Figures

List of Tables

Bibliography

[AAH15] Cfir Aguston, Yosi Ben Asher, and Gadi Haber. “Parallelization Hints via Code Skeletonization”. In: *IEEE Transactions on Parallel and Distributed Systems* 26.11 (Nov. 1, 2015). ISSN: 1558-2183 (cit. on p. 67).

[Aho+07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, eds. *Compilers: Principles, Techniques, & Tools*. 2nd ed. Pearson, Addison Wesley, 2007. ISBN: 978-0-321-48681-3 (cit. on p. 29).

[Alm+04] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. “Finding Effective Compilation Sequences”. In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES ’04. ACM, 2004. ISBN: 1-58113-806-7 (cit. on p. 72).

[Ami+12] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian Mcmahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. “Par4All: From Convex Array Regions to Heterogeneous Computing”. In: *Second International Workshop on Polyhedral Compilation Techniques 2012*. IMPACT ’12. 2012 (cit. on p. 59).

[Ans+09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. “PetaBricks: A Language and Compiler for Algorithmic Choice”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. ACM, 2009, pp. 38–49. ISBN: 978-1-60558-392-1 (cit. on p. 76).

[Ans+11] Jason Ansel, Maciej Pacula, Saman Amarasinghe, and Una-May O’Reilly. “An Efficient Evolutionary Algorithm for Solving Incrementally Structured Problems”. In: *Proceedings of the 13th Annual*

Conference on Genetic and Evolutionary Computation. GECCO '11. ACM, 2011, pp. 1699–1706. ISBN: 978-1-4503-0557-0 (cit. on p. 76).

[Ans+12] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. "Siblingrivalry: Online Autotuning Through Local Competitions". In: *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '12. ACM, 2012, pp. 91–100. ISBN: 978-1-4503-1424-4 (cit. on p. 51).

[Ans+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "OpenTuner: An Extensible Framework for Program Autotuning". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*. PACT '14. ACM, 2014, pp. 303–316. ISBN: 978-1-4503-2809-8 (cit. on pp. 52, 138).

[Ans14] Jason Ansel. "Autotuning Programs with Algorithmic Choice". Dissertation. Massachusetts Institute of Technology, 2014 (cit. on p. 76).

[Aru+17] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. "A Brief Survey of Deep Reinforcement Learning". In: *IEEE Signal Processing Magazine* 34.6 (Nov. 2017), pp. 26–38. ISSN: 1053-5888 (cit. on p. 18).

[Ash+18] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. "A Survey on Compiler Autotuning Using Machine Learning". In: *ACM Computing Surveys* 51.5 (Sept. 2018), pp. 1–42. ISSN: 0360-0300 (cit. on p. 15).

[ATN09] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures". In: *Euro-Par 2009 – Parallel Processing Workshops*. Ed. by HX. Lin et al. Lecture Notes in Computer Science vol. 6043. Springer, Berlin, Heidelberg, 2009, pp. 56–65. ISBN: 978-3-642-14121-8 (cit. on p. 55).

[Aug+09] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Euro-Par 2009 – Parallel*

Processing. Ed. by H. Sips, D. Epema, and HX. Lin. Vol. 5704. Lecture Notes in Computer Science vol. 5704. Springer, Berlin, Heidelberg, 2009. ISBN: 978-3-642-03869-3 (cit. on p. 54).

[Bai16] Alexander Baier. “Automatic Loop Partitioning for Heterogeneous Systems”. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT) – IPD Tichy, 2016 (cit. on p. 114).

[Bai95] Leemon Baird. “Residual Algorithms: Reinforcement Learning with Function Approximation”. In: *Proceedings of the Twelfth International Conference on Machine Learning.* Morgan Kaufmann, 1995, pp. 30–37. ISBN: 978-1558603776 (cit. on p. 19).

[Ban88] Utpal K. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988. ISBN: 978-0-89838-289-1 (cit. on p. 25).

[Bao+16] Wenlei Bao, Changwan Hong, Sudheer Chunduri, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. “Static and Dynamic Frequency Scaling on Multicore CPUs”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.4 (Dec. 2016), 51:1–51:26. ISSN: 1544-3566 (cit. on pp. 10, 56).

[Bar08] Alexander Barvinok. *Integer Points in Polyhedra.* European Mathematical Society, 2008. ISBN: 978-3-03719-052-4 (cit. on p. 28).

[Bas+08a] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. “A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs”. In: *Proceedings of the 22Nd Annual International Conference on Supercomputing.* ICS ’08. ACM, 2008, pp. 225–234. ISBN: 978-1-60558-158-3 (cit. on p. 73).

[Bas+08b] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. “Automatic Data Movement and Computation Mapping for Multi-Level Parallel Architectures with Explicitly Managed Memories”. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP ’08. ACM, 2008, pp. 1–10. ISBN: 978-1-59593-795-7 (cit. on p. 73).

[Bas04] Cedric Bastoul. “Code Generation in the Polyhedral Model Is Easier Than You Think”. In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’04. IEEE Computer Society, 2004, pp. 7–16. ISBN: 978-0-7695-2229-6 (cit. on p. 63).

[BE92] William Blume and Rudolf Eigenmann. “Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs”. In: *IEEE Transactions on Parallel and Distributed Systems* 3.6 (Nov. 1992), pp. 643–656. ISSN: 1045-9219 (cit. on p. 58).

[BGW13] Prasanna Balaprakash, Robert B. Gramacy, and Stefan M. Wild. “Active-Learning-Based Surrogate Models for Empirical Performance Tuning”. In: *Proceedings of the 2013 IEEE International Conference on Cluster Computing*. CLUSTER ’13. IEEE, 2013, pp. 1–8. ISBN: 978-1-4799-0898-1 (cit. on pp. 10, 55).

[Bil+97] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. “Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology”. In: *Proceedings of the 11th International Conference on Supercomputing*. ICS ’97. ACM, 1997, pp. 340–347. ISBN: 978-0-89791-902-9 (cit. on p. 72).

[Blu+96] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Jaejin Lawrence Thomas Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. “Parallel Programming with Polaris”. In: *Computer* 29.12 (Dec. 1996), pp. 78–82. ISSN: 0018-9162 (cit. on p. 58).

[Böh15] Lukas Böhm. “Compile-Time Code Parallelization for Self-Adapting Acceleration on Intel Xeon Phi”. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT) – IPD Tichy, 2015 (cit. on pp. 116, 118, 120, 121).

[Böh18] Lukas Böhm. “Pattern-Based Heterogeneous Parallelization”. Master’s Thesis. Karlsruhe Institute of Technology (KIT) – IPD Tichy, 2018 (cit. on pp. 114, 115).

[Bon+08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. “A Practical Automatic Polyhedral Parallelizer and Locality Optimizer”. In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. ACM, 2008, pp. 101–113. ISBN: 978-1-59593-860-2 (cit. on pp. 29, 63).

[BPC12] James Bergstra, Nicolas Pinto, and David Cox. “Machine Learning for Predictive Auto-Tuning with Boosted Regression Trees”. In: *Proceedings of the 2012 Conference on Innovative Parallel Computing*. InPar ’12. IEEE, 2012, pp. 1–9. ISBN: 978-1-4673-2633-9 (cit. on p. 51).

[BR15] Tania Banerjee and Sanjay Ranka. “A Genetic Algorithm Based Autotuning Approach for Performance and Energy Optimization”. In: *Proceedings of the Sixth International Green and Sustainable Computing Conference*. IGSC ’15. IEEE, 2015, pp. 1–8. ISBN: 978-1-5090-0172-9 (cit. on p. 74).

[Bre95] Eric A. Brewer. “High-Level Optimization via Automated Statistical Modeling”. In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’95. ACM, 1995, pp. 80–91. ISBN: 978-0-89791-700-1 (cit. on p. 49).

[Bro+16] Kevin J. Brown, HyoukJoong Lee, Tiark Romp, Aarvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. “Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns”. In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO ’16. ACM, 2016, pp. 194–205. ISBN: 978-1-4503-3778-6 (cit. on p. 67).

[BRS10] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. “Automatic C-to-CUDA Code Generation for Affine Programs”. In: *International Conference on Compiler Construction*. CC ’10. Ed. by Rajiv Gupta. Lecture Notes in Computer Science vol. 6011. Springer, Berlin, Heidelberg, 2010, pp. 244–263. ISBN: 978-3-642-11969-9 (cit. on pp. 62, 73).

[BWZ94] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. “Chains of Recurrences - a Method to Expedite the Evaluation of Closed-Form Functions”. In: *Proceedings of the International Symposium on Sym-*

bolic and Algebraic Computing. ISSAC '94. ACM, 1994, pp. 242–249. ISBN: 0-89791-638-7 (cit. on p. 22).

[Cav+06] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin, and Olivier Temam. "Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs". In: *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '06. ACM, 2006, pp. 24–34. ISBN: 978-1-59593-543-4 (cit. on pp. 69, 70).

[Cav+07] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. "Rapidly Selecting Good Compiler Optimizations Using Performance Counters". In: *Proceedings of the 2007 International Symposium on Code Generation and Optimization*. CGO '07. IEEE, 2007, pp. 185–197. ISBN: 978-0-7695-2764-2 (cit. on p. 70).

[CCH08] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A Framework for Composing High-Level Loop Transformations*. Technical Report. University of Southern California, 2008 (cit. on p. 72).

[Cha12] Kuo-Hao Chang. "Stochastic Nelder–Mead Simplex Method – A New Globally Convergent Direct Search Method for Simulation Optimization". In: *European Journal of Operational Research* 220.3 (Aug. 1, 2012), pp. 684–694. ISSN: 0377-2217 (cit. on p. 93).

[CSB11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures". In: *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium*. IPDPS '11. IEEE, 2011, pp. 676–687. ISBN: 978-1-61284-372-8 (cit. on p. 77).

[Dam+15] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. "Transparent Offloading of Computational Hotspots from Binary Code to Xeon Phi". In: *Proceedings of the 2015 Conference on Design, Automation and Test in Europe*. DATE '15. IEEE, 2015. ISBN: 978-3-9815-3705-5 (cit. on pp. 63, 118).

[DE10] Chirag Dave and Rudolf Eigenmann. “Automatically Tuning Parallel and Parallelized Programs”. In: *Languages and Compilers for Parallel Computing*. Ed. by Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li. Lecture Notes in Computer Science vol. 5898. Springer, Berlin, Heidelberg, 2010, pp. 126–139. ISBN: 978-3-642-13373-2 (cit. on p. 73).

[DGH17] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. “Optimistic Loop Optimization”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO’17. IEEE, 2017, pp. 292–304. ISBN: 978-1-5090-4931-8 (cit. on p. 28).

[Dia+10] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. “Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems”. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’10. ACM, 2010, pp. 353–364. ISBN: 978-1-4503-0178-7 (cit. on p. 65).

[Din+15] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O’Reilly, and Saman Amarasinghe. “Autotuning Algorithmic Choice for Input Sensitivity”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. ACM, 2015, pp. 379–390. ISBN: 978-1-4503-3468-6 (cit. on p. 76).

[DK06] Isaac Dooley and Laxmikant Kale. “Quantifying the Interference Caused by Subnormal Floating-Point Values”. In: *Proceedings of the Workshop on Operating System Interference in High Performance Applications*. OSHIPA ’06. 2006 (cit. on p. 130).

[Eng00] Robert A. Van Engelen. *Symbolic Evaluation of Chains of Recurrences for Loop Optimization*. Technical Report. Florida State University, 2000 (cit. on p. 22).

[Fea91] Paul Feautrier. “Dataflow Analysis of Array and Scalar References”. In: *International Journal of Parallel Programming* 20.1 (Feb. 1991), pp. 23–53. ISSN: 1573-7640 (cit. on p. 28).

[Fer+10] Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. "Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL". In: *International Workshop on Languages and Compilers for Parallel Computing*. Ed. by K. Cooper, J. Mellor-Crummey, and V. Sarkar. Lecture Notes in Computer Science vol. 6548. Springer, Berlin, Heidelberg, 2010, pp. 215–229. ISBN: 978-3-642-19594-5 (cit. on p. 66).

[FL11] Paul Feautrier and Christian Lengauer. "Polyhedron Model". In: *Encyclopedia of Parallel Computing*. Springer, Boston, MA, 2011, pp. 1581–1592. ISBN: 978-0-387-09766-4 (cit. on p. 26).

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (1987), pp. 319–349 (cit. on pp. 24, 25).

[Fri+13] Daniel Fried, Zhen Li, Ali Jannesari, and Felix Wolf. "Predicting Parallelization of Sequential Programs Using Supervised Learning". In: *Proceedings of the 12th International Conference on Machine Learning and Applications*. ICMLA '13. IEEE, 2013, pp. 72–77. ISBN: 978-0-7695-5144-9 (cit. on p. 70).

[Fur+11] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. "Milepost GCC: Machine Learning Enabled Self-Tuning Compiler". In: *International Journal of Parallel Programming* 39.3 (Jan. 2011), pp. 296–327. ISSN: 0885-7458 (cit. on p. 70).

[GAC12] Serge Guelton, Mehdi Amini, and Béatrice Creusillet. "Beyond Do Loops: Data Transfer Generation with Convex Array Regions". In: *International Workshop on Languages and Compilers for Parallel Computing*. LCPC '12. Ed. by H. Kasahara and K. Kimura. Lecture Notes in Computer Science vol. 7760. Springer, Berlin, Heidelberg, 2012, pp. 249–263. ISBN: 978-3-642-37657-3 (cit. on p. 59).

[GGL12] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. “Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”. In: *Parallel Processing Letters* 22.04 (Dec. 2012). ISSN: 0129-6264 (cit. on p. 63).

[GH16] Tobias Grosser and Torsten Hoefler. “Polly-ACC Transparent Compilation to Heterogeneous Hardware”. In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS ’16. ACM, 2016, pp. 1–13. ISBN: 978-1-4503-4361-9 (cit. on pp. 63, 111).

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. “Practical Dependence Testing”. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI ’91. ACM, 1991, pp. 15–29. ISBN: 0-89791-428-7 (cit. on pp. 24, 25, 115).

[Gro+13] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. “Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles”. In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. GPGPU-6. ACM, 2013, pp. 24–31. ISBN: 978-1-4503-2017-7 (cit. on p. 63).

[Gro+14] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. “Hybrid Hexagonal/Classical Tiling for GPUs”. In: *Proceedings of the 2014 International Symposium on Code Generation and Optimization*. CGO’14. ACM, 2014, pp. 66–75. ISBN: 978-1-4503-2670-4 (cit. on p. 63).

[Gum+10] Jayanth Gummaraju, Ben Sander, Laurent Morichetti, Benedict R. Gaster, Micheal Houston, and Bixia Zheng. “Twin Peaks: A Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors”. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT’10. IEEE, 2010, pp. 205–215. ISBN: 978-1-4503-0178-7 (cit. on p. 65).

[GVC15] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. “Polyhedral AST Generation Is More Than Scanning Polyhedra”. In: *ACM Trans-*

actions on Programming Languages and Systems (TOPLAS) 37.4 (July 15, 2015), pp. 1–50. ISSN: 0164-0925 (cit. on p. 30).

[Hag+18] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. “High Performance Stencil Code Generation with Lift”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO ’18. ACM, 2018, pp. 100–112. ISBN: 978-1-4503-5617-6 (cit. on p. 78).

[Hal+05] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. “Interprocedural Parallelization Analysis in SUIF”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27.4 (July 2005), pp. 662–731. ISSN: 0164-0925 (cit. on p. 59).

[Hal+96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. “Maximizing Multiprocessor Performance with the SUIF Compiler”. In: *Computer* 29.12 (Dec. 1996), pp. 84–89. ISSN: 0018-9162 (cit. on p. 59).

[Ham+09] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. “Profiling Java Programs for Parallelism”. In: *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*. IWMSE ’09. IEEE, 2009, pp. 49–55. ISBN: 978-1-4244-3718-4 (cit. on p. 59).

[Her+19] Killian Herveau, Philip Pfaffe, Martin Tillmann, Walter F. Tichy, and Carsten Dachsbacher. “Hybrid Online Autotuning for Parallel Ray Tracing”. In: *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Hank Childs and Steffen Frey. The Eurographics Association, 2019. ISBN: 978-3-03868-079-6 (cit. on pp. 81, 83).

[Hin01] Michael Hind. “Pointer Analysis: Haven’t We Solved This Problem Yet?” In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’01. ACM, 2001, pp. 54–61. ISBN: 1-58113-413-4 (cit. on p. 24).

[Hor97] Susan Horwitz. “Precise Flow-Insensitive May-Alias Analysis Is NP-Hard”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.1 (Jan. 1997), pp. 1–6. ISSN: 0164-0925 (cit. on p. 24).

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann, 2011. ISBN: 978-0-12-383872-8 (cit. on pp. 23, 24).

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. “Semantical Interprocedural Parallelization: An Overview of the PIPS Project”. In: *Proceedings of the 5th International Conference on Supercomputing.* ICS ’91. ACM, 1991, pp. 244–251. ISBN: 978-0-89791-434-5 (cit. on p. 58).

[Jim12] Alexandra Jimborean. “Adapting the Polytope Model for Dynamic and Speculative Parallelization”. Dissertation. Université de Strasbourg, 2012 (cit. on p. 62).

[Joh+04] Troy A. Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff. “Experiences in Using Cetus for Source-to-Source Transformations”. In: *Languages and Compilers for High Performance Computing.* LCPC ’04. Ed. by Rudolf Eigenmann, Z. Li, and Samuel P. Midkiff. Lecture Notes in Computer Science vol. 3602. Springer, Berlin, Heidelberg, 2004, pp. 1–14. ISBN: 978-3-540-28009-5 (cit. on p. 73).

[Jor+12] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. “A Multi-Objective Auto-Tuning Framework for Parallel Codes”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* SC ’12. IEEE, 2012, pp. 1–12. ISBN: 978-1-4673-0804-5 (cit. on pp. 10, 73).

[KA97] R. Matthew Kretchmar and Charles W. Anderson. “Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning”. In: *Proceedings of the International Conference on Neural Networks.* ICNN ’97. IEEE, 1997, pp. 834–837. ISBN: 978-0-7803-4122-7 (cit. on p. 104).

[KBK11] Mario Kicherer, Rainer Buchty, and Wolfgang Karl. “Cost-Aware Function Migration in Heterogeneous Systems”. In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. HiPEAC ’11. ACM, 2011, p. 137. ISBN: 978-1-4503-0241-8 (cit. on p. 55).

[Kic+12] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. “Seamlessly Portable Applications: Managing the Diversity of Modern Heterogeneous Systems”. In: *ACM Transactions on Architecture and Code Optimization (TACO) - Special Issue on High-Performance Embedded Architectures and Compilers* 8.4 (Jan. 2012), 42:1–42:20. ISSN: 1544-3566 (cit. on p. 55).

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. “Reinforcement Learning: A Survey”. In: *Journal of Artificial Intelligence Research* 4.1 (May 1996), pp. 237–285. ISSN: 1076-9757 (cit. on p. 17).

[Klö14] Andreas Klöckner. “Loo.Py: Transformation-Based Code Generation for GPUs and CPUs”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY ’14. ACM, 2014, 82:82–82:87. ISBN: 978-1-4503-2937-8 (cit. on p. 63).

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. “The Organization of Computations for Uniform Recurrence Equations”. In: *Journal of the ACM* 14.3 (July 1967), pp. 563–590. ISSN: 0004-5411 (cit. on p. 62).

[Kop18] Timo Kopf. “Adaptives Online-Tuning für kontinuerliche Zustandsräume”. Master’s Thesis. Karlsruhe Institute of Technology (KIT) – IPD Tichy, 2018. 96 pp. (cit. on p. 82).

[KP11] Thomas Karcher and Victor Pankratius. “Run-Time Automatic Performance Tuning for Multicore Applications”. In: *Euro-Par 2011 – Parallel Processing*. Ed. by Esmmanuel Jeannot, Raymond Namyst, and Jean Roman. Lecture Notes in Computer Science vol. 6852. Springer, Berlin, Heidelberg, 2011, pp. 3–14. ISBN: 978-3-642-23399-9 (cit. on p. 51).

[Kru14a] Michael Kruse. “Introducing Molly: Distributed Memory Parallelization with LLVM”. In: *CoRR* abs/1409.2088 (2014). arXiv: 1409.2088 `[cs]` (cit. on p. 63).

[Kru14b] Michael Kruse. “Lattice QCD Optimization and Polytopic Representations of Distributed Memory”. Dissertation. Université Paris Sud - Paris XI, 2014 (cit. on p. 63).

[LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization.* CGO ’04. IEEE, 2004. ISBN: 0-7695-2102-9 (cit. on pp. 6, 20).

[Lam74] Leslie Lamport. “The Parallel Execution of DO Loops”. In: *Communications of the ACM* 17.2 (Feb. 1974), pp. 83–93. ISSN: 0001-0782 (cit. on p. 62).

[LHK09] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. “Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping”. In: *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture.* MICRO 42. ACM, 2009, pp. 45–55. ISBN: 978-1-60558-798-1 (cit. on pp. 75, 130).

[Lia+09] Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, and Thomas Panas. “Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization”. In: *Languages and Compilers for Parallel Computing.* LCPC ’09. Ed. by Guang R. Gao, John Cavazos, Xiaoming Li, and Lori L. Pollock. Lecture Notes in Computer Science vol. 5898. Springer, Berlin, Heidelberg, 2009, pp. 308–322. ISBN: 978-3-642-13373-2 (cit. on p. 73).

[Lin+08] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. “Merge: A Programming Model for Heterogeneous Multi-Core Systems”. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS XIII. ACM, 2008, pp. 287–296. ISBN: 978-1-59593-958-6 (cit. on p. 65).

[LJW13] Zhen Li, Ali Jannesari, and Felix Wolf. “Discovery of Potential Parallelism in Sequential Programs”. In: *Proceedings of the 2013 42nd International Conference on Parallel Processing*. ICPP ’13. IEEE, 2013, pp. 1004–1013. ISBN: 978-0-7695-5117-3 (cit. on p. 59).

[LR91] William Landi and Barbara G. Ryder. “Pointer-Induced Aliasing: A Problem Classification”. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’91. ACM, 1991, pp. 93–103. ISBN: 978-0-89791-419-2 (cit. on p. 24).

[LRG12] Changmin Lee, Won W. Ro, and Jean-Luc Gaudiot. “Cooperative Heterogeneous Computing for Parallel Processing on CPU/GPU Hybrids”. In: *Proceedings of the 2012 16th Workshop on Interaction between Compilers and Computer Architectures*. INTERACT ’12. IEEE, 2012, pp. 33–40. ISBN: 978-1-4673-2613-1 (cit. on pp. 77, 130).

[Mae+10] Hamid Reza Maei, Csaba Szepesvári, Shalabh Bhatnagar, and Richard S. Sutton. “Toward Off-Policy Learning Control with Function Approximation”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Omnipress, 2010, pp. 719–726. ISBN: 978-1-60558-907-7 (cit. on p. 19).

[Mar13] Daniel E. Marthaler. “An Overview of Mathematical Methods for Numerical Optimization”. In: *Numerical Methods for Metamaterial Design*. Ed. by Kenneth Diest. Topics in Applied Physics vol. 127. Springer, Dordrecht, 2013, pp. 31–53. ISBN: 978-94-007-6664-8 (cit. on p. 15).

[MBC79] M. D. McKay, R. J. Beckman, and W. J. Conover. “A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code”. In: *Technometrics* 21.2 (1979), pp. 239–245. ISSN: 0040-1706. JSTOR: 1268522 (cit. on pp. 93, 152).

[Mik+14] Dmitry Mikushin, Nikolay Likhogrud, Eddy Z. Zhang, and Christopher Bergström. “KernelGen – The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs”. In: *Proceedings of the 2014 IEEE International Parallel and Distributed Processing Symposium Workshops*. IPDPSW ’14. IEEE, 2014, pp. 1011–1020. ISBN: 978-1-4799-4116-2 (cit. on p. 63).

[MMT15] Korbinian Molitorisz, Tobias Müller, and Walter F. Tichy. “Patty: A Pattern-Based Parallelization Tool for the Multicore Age”. In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM ’15. ACM, 2015, pp. 153–163. ISBN: 978-1-4503-3404-4 (cit. on pp. 67, 74).

[Mor+03] Anna Morajko, Oleg Morajko, Josep Jorba, Tomàs Margalef, and Emilio Luque. “Dynamic Performance Tuning of Distributed Programming Libraries”. In: *Computational Science — ICCS 2003*. ICCS 2003. Ed. by Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Yuriy E. Gorbachev, Jack J. Dongarra, and Albert Y. Zomaya. Lecture Notes in Computer Science vol. 2660. Springer, Berlin, Heidelberg, 2003, pp. 191–200. ISBN: 978-3-540-40197-1 (cit. on p. 54).

[Mor+04] Anna Morajko, Oleg Morajko, Tomàs Margalef, and Emilio Luque. “MATE: Dynamic Performance Tuning Environment”. In: *Euro-Par 2004 – Parallel Processing*. Ed. by Marco Danelutto, Marco Vanneschi, and Domenico Laforenza. Lecture Notes in Computer Science vol. 3149. Springer, Berlin, Heidelberg, 2004, pp. 98–107. ISBN: 978-3-540-22924-7 (cit. on p. 54).

[MRR12] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming*. Morgan Kaufmann, 2012. ISBN: 978-0-12-415993-8 (cit. on p. 23).

[Mur+14] Saurav Muralidharan, Manu Shantharam, Mary Hall, Micheal Garland, and Bryan Catanzaro. “Nitro: A Framework for Adaptive Code Variant Tuning”. In: *Proceedings of the 2014 IEEE International Parallel and Distributed Processing Symposium*. IPDPS ’14. IEEE, 2014, pp. 501–512. ISBN: 978-1-4799-3799-8 (cit. on p. 55).

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. “PolyMage: Automatic Optimization for Image Processing Pipelines”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’15. ACM, 2015, pp. 429–443. ISBN: 978-1-4503-2835-7 (cit. on p. 78).

[NC14] Cedric Nugteren and Henk Corporaal. “Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (Dec. 2014), pp. 1–25. ISSN: 1544-3566 (cit. on p. 67).

[NM65] J. A. Nelder and R. Mead. “A Simplex Method for Function Minimization”. In: *The Computer Journal* 7.4 (Jan. 1965). ISSN: 0010-4620 (cit. on pp. 15, 91).

[OPT09] Frank Otto, Victor Pankratius, and Walter F. Tichy. “High-Level Multicore Programming with Xjava”. In: *Proceedings of the 2009 31st International Conference on Software Engineering - Companion Volume.* ICSE ’09. IEEE, 2009, pp. 319–322. ISBN: 978-1-4244-3495-4 (cit. on p. 76).

[Ott+10] Frank Otto, Christoph A. Schaefer, Matthias Dempe, and Walter F. Tichy. “A Language-Based Tuning Mechanism for Task and Pipeline Parallelism”. In: *Euro-Par 2010 – Parallel Processing.* Ed. by Pasqua D’Ambra, Mario Guarracino, and Domenico Talia. Lecture Notes in Computer Science vol. 6272. Springer, Berlin, Heidelberg, 2010, pp. 328–340. ISBN: 978-3-642-15291-7 (cit. on p. 76).

[Pac+12] Maciej Pacula, Jason Ansel, Saman Amarasinghe, and Una-May O’Reilly. “Hyperparameter Tuning in Bandit-Based Adaptive Operator Selection”. In: *Applications of Evolutionary Computation.* Ed. by Cecilia Chio et al. Lecture Notes in Computer Science vol. 7248. Malaga, Spain: Springer, Berlin, Heidelberg, 2012. ISBN: 978-3-642-29178-4 (cit. on p. 52).

[Par+13] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P. Sadayappan. “Predictive Modeling in a Polyhedral Optimization Space”. In: *International Journal of Parallel Programming* 41.5 (Feb. 2013), pp. 704–750. ISSN: 0885-7458, 1573-7640 (cit. on p. 70).

[Par94] Jeong-Soo Park. “Optimal Latin-Hypercube Designs for Computer Experiments”. In: *Journal of Statistical Planning and Inference* 39.1 (Apr. 1994), pp. 95–111. ISSN: 0378-3758 (cit. on pp. 93, 152).

[PE06] Zhelong Pan and Rudolf Eigenmann. “Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning”. In: *Proceedings of the 2006 International Symposium on Code Generation and Optimization.* CGO ’06. IEEE, 2006. ISBN: 0-7695-2499-0 (cit. on p. 73).

[Pfa+17] Philip Pfaffe, Martin Tillmann, Sigmar Walter, and Walter F. Tichy. “Online-Autotuning in the Presence of Algorithmic Choice”. In: *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops.* IPDPSW ’17. IEEE, 2017, pp. 1379–1388. ISBN: 978-1-5386-3408-0 (cit. on pp. 13, 14, 94, 95).

[PGT19] Philip Pfaffe, Tobias Grosser, and Martin Tillmann. “Efficient Hierarchical Online-Autotuning: A Case Study on Polyhedral Accelerator Mapping”. In: *Proceedings of the ACM International Conference on Supercomputing.* ICS ’19. ACM, 2019, pp. 354–366. ISBN: 978-1-4503-6079-1 (cit. on pp. 82, 98, 107, 138, 143, 147–149).

[PW94] William Pugh and David Wonnacott. “Static Analysis of Upper and Lower Bounds on Dependences and Parallelism”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.4 (July 1994), pp. 1248–1278. ISSN: 0164-0925 (cit. on p. 27).

[Rag+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI ’13. ACM, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6 (cit. on p. 77).

[RDP14] Benjamin Ranft, Oliver Denninger, and Philip Pfaffe. “A Stream Processing Framework for On-Line Optimization of Performance and Energy Efficiency on Heterogeneous Systems”. In: *Proceedings of the 2014 IEEE International Parallel and Distributed Processing Symposium Workshops.* IPDPSW ’14. IEEE, 2014. ISBN: 978-1-4799-4116-2 (cit. on pp. 77, 130).

[Rep96] Thomas Reps. “On the Sequential Nature of Interprocedural Program-Analysis Problems”. In: *Acta Informatica* 33.8 (Nov. 1996), pp. 739–757. ISSN: 1432-0525 (cit. on p. 26).

[RHG17] Ari Rasch, Michael Haidl, and Sergei Gorlatch. “ATF: A Generic Auto-Tuning Framework”. In: *Proceedings of the 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems*. HPCC/SmartCity/DSS’17. IEEE, 2017, pp. 64–71. ISBN: 978-1-5386-2588-0 (cit. on p. 52).

[Rud+10] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. “A Programming Language Interface to Describe Transformations and Code Generation”. In: *Languages and Compilers for Parallel Computing*. Languages and Compilers for Parallel Computing. Ed. by Keith Cooper, John Mellor-Crummey, and Vivek Sarkar. Lecture Notes in Computer Science vol. 6548. Springer, Berlin, Heidelberg, 2010, pp. 136–150. ISBN: 978-3-642-19594-5 (cit. on p. 72).

[RVD07] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. “Function Level Parallelism Driven by Data Dependencies”. In: *ACM SIGARCH Computer Architecture News* 35.1 (Mar. 2007), pp. 55–62. ISSN: 0163-5964 (cit. on p. 67).

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 1st edition. MIT Press, 1998. ISBN: 0-262-19398-1 (cit. on pp. 17, 18).

[Sch09] Christoph A. Schaefer. “Reducing Search Space of Auto-Tuners Using Parallel Patterns”. In: *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*. IWMSE ’09. IEEE, 2009, pp. 17–24. ISBN: 978-1-4244-3718-4 (cit. on p. 51).

[Sch17] Bernhard Scheirle. “Musterbasierte Analyse zur Detektion von Parallelisierungsmöglichkeiten”. Master’s Thesis. Karlsruhe Institute of Technology (KIT) – IPD Tichy, 2017 (cit. on p. 114).

[SHH62] W. Spendley, G. R. Hext, and F. R. Himsworth. “Sequential Application of Simplex Designs in Optimisation and Evolutionary Operation”. In: *Technometrics* 4.4 (Nov. 1962), pp. 441–461. ISSN: 0040-1706 (cit. on p. 15).

[SKK17] Lukas Sommer, Jens Korinth, and Andreas Koch. “OpenMP Device Offloading to FPGA Accelerators”. In: *Proceedings of the 2017 IEEE 28th International Conference on Application-Specific Systems, Architectures and Processors*. ASAP ’17. IEEE, 2017, pp. 201–205. ISBN: 978-1-5090-4825-0 (cit. on p. 66).

[SMS08] Richard S. Sutton, Hamid R. Maei, and Csaba Szepesvári. “A Convergent O(n) Temporal-Difference Algorithm for Off-Policy Learning with Linear Function Approximation”. In: *Advances in Neural Information Processing Systems 21*. Ed. by D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou. Curran Associates, Inc., 2008, pp. 1609–1616. ISBN: 978-1605609492 (cit. on p. 20).

[SOB14] Zehra Sura, Kevin OBrien, and Jose Brunheroto. “Using Multiple Threads to Accelerate Single Thread Performance”. In: *Proceedings of the 2014 IEEE International Parallel and Distributed Processing Symposium*. IPDPS ’14. IEEE, 2014, pp. 985–994. ISBN: 978-1-4799-3799-8 (cit. on p. 60).

[SP78] Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. Technical Report. New York University, Courant Institute of Mathematical Sciences, ComputerScience Department, 1978 (cit. on p. 26).

[SPT09] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. “Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications”. In: *Euro-Par 2009 – Parallel Processing*. Ed. by Henk Sips, Dick Epema, and Hai-Xiang Lin. Lecture Notes in Computer Science vol. 5704. Springer, Berlin, Heidelberg, 2009, pp. 9–20. ISBN: 978-3-642-03868-6 (cit. on p. 50).

[SPT10] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. “Engineering Parallel Applications with Tunable Architectures”. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Soft-*

ware Engineering - Volume 1. ICSE '10. New York, NY, USA: ACM, 2010, pp. 405–414. ISBN: 978-1-60558-719-6 (cit. on p. 76).

[SRD17] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. “LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization.* CGO '17. IEEE, 2017, pp. 74–85. ISBN: 978-1-5090-4931-8 (cit. on p. 78).

[Ste46] Stanley Smith Stevens. “On the Theory of Scales of Measurement”. In: *Science* 103.2684 (1946), pp. 677–680. ISSN: 0036-8075 (cit. on p. 13).

[Str+12] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. “Sambamba: A Runtime System for Online Adaptive Parallelization”. In: *Compiler Construction.* Ed. by Michael O'Boyle. Lecture Notes in Computer Science vol. 7210. Berlin, Heidelberg: Springer, Berlin, Heidelberg, 2012, pp. 240–243. ISBN: 978-3-642-28651-3 (cit. on p. 60).

[Str+13] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. “Sambamba: Runtime Adaptive Parallel Execution”. In: *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems.* ADAPT '13. New York, NY, USA: ACM, 2013, pp. 1–6. ISBN: 978-1-4503-2022-1 (cit. on p. 60).

[Str+15] Kevin Streit, Johannes Doerfert, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. “Generalized Task Parallelism”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.1 (Apr. 2015), 8:1–8:25. ISSN: 1544-3566 (cit. on p. 60).

[Sus92] Alan Sussman. “Model-Driven Mapping Onto Distributed Memory Parallel Computers”. In: *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing.* Supercomputing '92. IEEE, 1992, pp. 818–829. ISBN: 0-8186-2630-5 (cit. on p. 69).

[Sut88] Richard S. Sutton. “Learning to Predict by the Methods of Temporal Differences”. In: *Machine Learning* 3.1 (Aug. 1988), pp. 9–44. ISSN: 1573-0565 (cit. on p. 18).

[ŢCH02] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. “Active Harmony: Towards Automated Performance Tuning”. In: *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing.* SC ’02. IEEE, 2002, pp. 1–11. ISBN: 0-7695-1524-X (cit. on p. 50).

[Tes95] Gerald Tesauro. “Temporal Difference Learning and TD-Gammon”. In: *Communications of the ACM* 38.3 (Mar. 1995), pp. 58–68. ISSN: 0001-0782 (cit. on p. 18).

[TF10] Georgios Tournavitis and Björn Franke. “Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information”. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques.* PACT ’10. ACM, 2010, pp. 377–388. ISBN: 978-1-4503-0178-7 (cit. on p. 67).

[TH11] Ananta Tiwari and Jeffrey K. Hollingsworth. “Online Adaptive Code Generation and Tuning”. In: *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium.* IPDPS ’11. IEEE, 2011, pp. 879–892. ISBN: 978-1-61284-372-8 (cit. on p. 72).

[Til+14] Martin Tillmann, Thomas Karcher, Carsten Dachsbacher, and Walter F. Tichy. “Application-Independent Autotuning for GPUs.” In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE).* Ed. by Michael Bader, Arndt Bode, Hans-Joachim Bungartz, Michael Gerndt, Gerhard R. Joubert, and Frans J. Peters. Advances in Parallel Computing 25. IOS Press, 2014, pp. 626–635. ISBN: 978-1-61499-380-3 (cit. on p. 52).

[Tiw+09] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. “A Scalable Auto-Tuning Framework for Compiler Optimization”. In: *Proceedings of the 2009 IEEE International Parallel Distributed Processing Symposium.* IPDPS ’09. IEEE, 2009, pp. 1–12. ISBN: 978-1-4244-3751-1 (cit. on p. 72).

[Tou+09] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O’Boyle. “Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI ’09. ACM, 2009, pp. 177–187. ISBN: 978-1-60558-392-1 (cit. on p. 70).

[Vas+18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. “Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions”. In: *CoRR* abs/1802.04730 (2018). arXiv: 1802.04730 `[cs]` (cit. on p. 78).

[VD00] Richard Vuduc and James W. Demmel. “Code Generators for Automatic Tuning of Numerical Kernels: Experiences with FFTW Position Paper”. In: *Semantics, Applications, and Implementation of Program Generation*. SAIG ’00. Ed. by Walid Taha. Lecture Notes in Computer Science vol. 1924. Springer, Berlin, Heidelberg, 2000, pp. 190–211. ISBN: 978-3-540-41054-6 (cit. on p. 50).

[VDB01] Richard Vuduc, James W. Demmel, and Jeff Bilmes. “Statistical Models for Automatic Performance Tuning”. In: *Computational Science — ICCS 2001*. Ed. by Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C. J. Kenneth Tan. Lecture Notes in Computer Science vol. 2073. Berlin, Heidelberg: Springer, Berlin, Heidelberg, 2001, pp. 117–126. ISBN: 978-3-540-42232-7 (cit. on p. 50).

[VE00] Micheal J. Voss and Rudolf Eigenmann. “ADAPT: Automated De-Coupled Adaptive Program Transformation”. In: *Proceedings of the 2000 International Conference on Parallel Processing*. ICPP’00. IEEE, 2000, pp. 163–170. ISBN: 0-7695-0768-9 (cit. on p. 75).

[VE01] Michael J. Voss and Rudolf Eigemann. “High-Level Adaptive Program Optimization with ADAPT”. In: *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. PPoPP ’01. ACM, 2001, pp. 93–102. ISBN: 1-58113-346-4 (cit. on p. 75).

[Ver+13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. “Polyhedral Parallel Code Generation for CUDA”. In: *ACM Transactions on Architecture and Code Optimization (TACO) - Special Issue on High-Performance Embedded Architectures and Compilers* 9.4 (Jan. 2013), 54:1–54:23. ISSN: 1544-3566 (cit. on pp. 26, 27, 29, 62, 108).

[Ver10] Sven Verdoolaege. “Isl: An Integer Set Library for the Polyhedral Model”. In: *Mathematical Software – ICMS 2010*. Ed. by Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama. Lecture Notes in Computer Science vol. 6326. Springer, Berlin, Heidelberg, 2010, pp. 299–302. ISBN: 978-3-642-15582-6 (cit. on p. 30).

[Ver16] Sven Verdoolaege. *Presburger Formulas and Polyhedral Compilation*. 2016. URL: https://lirias.kuleuven.be/bitstream/123456789/523109/3/polycomp-tutorial-v0.02.pdf (visited on 01/11/2020) (cit. on p. 27).

[Vud03] Richard Wilson Vuduc. “Automatic Performance Tuning of Sparse Matrix Kernels”. Dissertation. University of California, Berkeley, 2003 (cit. on p. 50).

[Wan+08] Cheng Wang, Youfeng Wu, Edson Borin, Shiliang Hu, Wei Liu, Tin-fook Ngai, and Jesse Fang. “New Slicing Algorithms for Parallelizing Single-Threaded Programs”. In: *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*. PESPMA ’08. 2008 (cit. on p. 59).

[Wat89] Christopher Watkins. “Learning From Delayed Rewards”. Dissertation. King’s College, 1989 (cit. on p. 19).

[WB95] Greg Welch and Gary Bishop. *An Introduction to the Kalman Filter*. Technical Report. University of North Carolina at Chapel Hill, 1995 (cit. on p. 78).

[WD92] Christopher J. C. H. Watkins and Peter Dayan. “Q-Learning”. In: *Machine Learning* 8.3-4 (1992), pp. 279–292. ISSN: 1573-0565 (cit. on p. 18).

[WD98] R. C. Whaley and J. J. Dongarra. “Automatically Tuned Linear Algebra Software”. In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. SC ’98. IEEE, 1998, pp. 38–38. ISBN: 0-89791-984-X (cit. on pp. 9, 49).

[Wen16] André Wengert. “Adaptives Auto-Tuning”. Master’s Thesis. Karlsruhe Institute of Technology (KIT) – IPD Tichy, 2016 (cit. on p. 82).

[WO09] Zheng Wang and Micheal F.P. O'Boyle. "Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach". In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '09. ACM, 2009, pp. 75–84. ISBN: 978-1-60558-397-6 (cit. on p. 70).

[Wol82] Michael Joseph Wolfe. "Optimizing Supercompilers for Supercomputers". Dissertation. University of Illinois at Urbana-Champaign, 1982 (cit. on p. 25).

[YSD05] Haihang You, Keith Seymour, and Jack Dongarra. *An Effective Empirical Search Method for Automatic Software Tuning*. Technical Report. University of Tennessee, Innovative Computing Laboratory, 2005 (cit. on pp. 16, 50).

[Zha+13] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. "Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. ACM, 2013, pp. 435–446. ISBN: 978-1-4503-2014-6 (cit. on p. 22).

[ZR08] Xin Zheng and Radu Rugina. "Demand-Driven Alias Analysis for C". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. ACM, 2008, pp. 197–208. ISBN: 978-1-59593-689-9 (cit. on p. 22).