Efficient Hierarchical Online-Autotuning

A Case Study on Polyhedral Accelerator Mapping

Philip Pfaffe
Department of Computer Science
Karlsruhe Institute of Technology
philip.pfaffe@kit.edu

Tobias Grosser
Department of Computer Science
ETH Zurich
tobias.grosser@inf.ethz.ch

Martin Tillmann
Department of Computer Science
Karlsruhe Institute of Technology
martin.tillmann@kit.edu

ABSTRACT

Identifying the (near) optimal program variants an optimizing and parallelizing compiler should generate is known to be difficult. Autotuning is the best solution to navigate the often high-dimensional space of possible options. However, to be practical an autotuner should (a) have high convergence speed and (b) be robust in face of varying inputs. Current techniques for offline tuning, where convergence speed is less important, provide solutions only for known inputs, whereas online tuning can be input sensitive but currently lacks in convergence speed. In this paper, we present hierarchical online-autotuning, a novel technique to exploit structure in the search space and the underlying tuning problem to increase convergence speed during online tuning. By modeling symmetries and redundancies in configurations and by exploiting domain knowledge to predict performance we reduce the search space size by orders of magnitudes. Combining our tuner with a polyhedral parallelizing compiler for GPUs, we show that the performance of a GEMM GPU kernel generated with default parameters is increased by 6× and that the convergence speed of the tuning process is increased by a factor of up to 1.7 compared to OpenTuner. With hierarchical tuning we make the deployment of always-on online-autotuning practical.

KEYWORDS

online-autotuning, performance optimization, GPGPU, polyhedral compilation

ACM Reference Format:
Philip Pfaffe, Tobias Grosser, and Martin Tillmann. 2019. Efficient Hierarchical Online-Autotuning: A Case Study on Polyhedral Accelerator Mapping. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3330345.3330377

1 INTRODUCTION

It is not the complexity of new hardware, but the generation of efficient code for today's increasingly complex hardware that is likely to become *the limiting factor* for further increases in computational performance. Today's continuously increasing demand for more compute power, together with a growing need for low power devices, has resulted in the near-ubiquitous deployment of heterogeneous hardware. From embedded devices to supercomputers, typical hardware deployments complement a general purpose host processor with a variety of specialized accelerators, most commonly GPUs, but also FPGA, DSPs, and others. Programming this hardware for high-performance does not only require knowledge of a range of specialized programming paradigms. It also requires the programmer to choose performance-optimal program implementations taken from a large set of choices with vastly different performance.

Even though analytical performance models exist and indeed excel in optimizing specific program properties, those reaching for near peak-performance usually fall back to automatic search space exploration techniques. While analytical performance models have been shown to be extremely successful for specific kernels and platforms such as BLAS on CPUs [29], more complex tuning problems that involve a variety of different kernels or tuning decisions on different levels of the code generation process are broadly assumed to require automatic tuning techniques. Hence, production DSL compilers such as Halide [37], Polymage [33], or Tensor Comprehensions [47] provide the ability to use manually specified transformations or rely on a simple analytical performance model, but for production runs or research evaluations autotuned codes are commonly used.

Applying autotuning to compilers raises a new major design question: Should tuning be applied offline (i.e., during compilation), or online (i.e, during application runtime)? The tuning scenario for offline methods is straightforward: the program is repeatedly compiled and measured on a set of predefined or randomly generated inputs, trying different compilation options in every iteration. For online-autotuning the scenario is more involved. Here, the autotuner is integrated into the program by the compiler and is always on, tuning continuously. To measure the to-be-tuned part the program flow has to execute it repeatably. The repetition does not require any structure and most programs of interest feature an expensive section that is called recursively, iteratively, or through event-based invocations. These repetitions form the tuning loop. Each iteration of the tuning loop is measured and communicated to the online-autotuner. Between iterations the online-autotuner chooses and sets new configurations. In an online context the input is not a predefined set of test inputs but the actual input of the deployed program. Continuous online-autotuning therefore optimizes the actual production use of the program. Optimized configurations for given inputs can be remembered for future occurrences of the same input, this circumvents the need to restart the online-autotuning process for already known states. Further

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330377

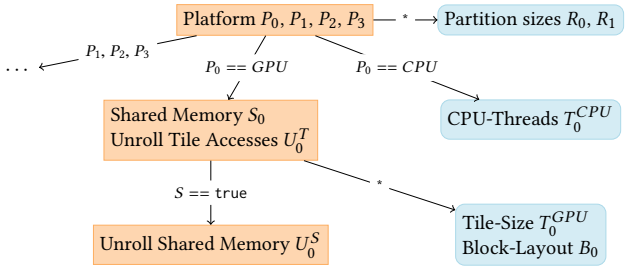


Figure 1: Search space graph for a partitioned 2D kernel offloaded to both GPU and CPU. Parameters in rectangular boxes are nominal, those in rounded boxes are non-nominal.

benefits of online-autotuning come from taking the current system state into account. Load levels, bandwidth limits or I/O are just some examples of runtime effects that cannot be observed during offline-autotuning but are inherently respected in an online context.

The major drawback of online tuning is that every decision, good or bad, is directly visible to the user of the tuned application. This means online tuners need to complete their search as quickly as possible, avoiding bad configurations as much as possible. To make deploying always-on online tuning feasible, we present a novel autotuner in this paper which improves upon the state of the art in terms of search time while still finding adequate results. We apply it to dynamically optimize automatic accelerator mapping. We model the space of possible program variants as a directed acyclic graph that expresses dependences between parameters. As an example, Figure 1 shows an excerpt of the search space graph for the accelerator mapping and tuning problem. Independent subspaces, which are the nodes of the graph, may be tuned by individual, potentially different search algorithms. Using analytical information about the search space, manually designed rules can further steer the search and additionally prune uninteresting parts of the search space without actually sampling them. By tuning *hierarchically*, we only need to explore a small subset of the actual search space. As part of a parallelizing compiler, both the analytical models and the dependences are derived without programmer interaction.

We evaluate our new online tuning approach by embedding it into an extension of Polly-ACC [20], a state-of-the-art parallelizing compiler for GPUs. Our extension enables cooperative execution on GPUs and CPUs, sharing work across platforms while optimizing the distribution and platform-specific parameters (such as tile sizes, thread blocks, shared memory, unrolling). Even though the search space we explore is large and complex, our experiments show that exploiting its structure while using analytical compiler heuristics to prune the search space yield performance improvements of 6× for the Polybench GEMM kernel, reducing the search time by a factor of 2.

The key contributions of this paper are:

- A method to exploit structure in the search space, allowing for hierarchical tuning of dependent subspaces using individual search algorithms, thus reducing the complexity of the space by orders of magnitude.
- A parallelizing compiler for cooperative multiplatform execution, embedding the online tuner into the program to optimize work distribution and platform-specific parameters.
- Pruning of the search space using rules automatically implemented by the compiler.

2 BACKGROUND

We summarize the background and formalization of both autotuning and polyhedral accelerator compilation.

2.1 Offline- and Online-Autotuning

For the task of autotuning, the categorization into offline and online methods is of little consequence, as any tuning method invented for either class can be applied to the other. The key difference is a shift of priorities. Offline tuning, as popularized by the ATLAS [54] library in 2001, entails searching for an optimal configuration ahead of time, e.g. during compilation or deployment. With online tuning, such as the tuner promoted by ActiveHarmony [44], optimization happens at application runtime. Every sample configuration and its runtime is thus observable by users of the system. Since every bad decision of the tuner impacts the overall performance, a desirable if not mandatory property of an online search is quick convergence with as few bad samples as possible. The fundamental difference to offline-autotuning is that the cost of exploration has to be amortized during the tuning process itself. Convergence in the context of tuning means for the tuner to eventually settle on a single configuration without exploring any further. This final configuration is not necessarily globally optimal. ActiveHarmony, for example, defaults to the Nelder-Mead [34] search algorithm instead of an exhaustive search, accepting that this might only produce a local optimum.

To formalize the autotuning process, we follow Pfaffe et al. [36] in their definition of tuning as the process of minimizing a *measurement function* $m : \mathcal{T} \to \mathbb{R}$. For a given *configuration* $C \in \mathcal{T}$, $m(C)$ is a measurement of the execution time of the parallelized program part using the parameter configuration C. The *tuning space* or *search space* is the space spanned by the *tuning parameters* $\tau_j \subseteq \mathbb{R}$:

$$\mathcal{T} = \tau_0 \times \cdots \times \tau_J \subseteq \mathbb{R}^J.$$

Pfaffe et al. further classify parameters into one of the four classes: Nominal, Ordinal, Interval, or Ratio parameters based on their respective scale. A typical nominal parameter, defined by an unordered collection of values, is controlling the choice of an algorithm or platform, whereas a ratio parameter would be the thread count in a parallel region. For simplicity, we only distinguish between nominal and non-nominal parameters in this work. The distinction between these classes is important because it affects the selection of search algorithms. Algorithms often cannot handle nominal parameters correctly, because they rely on metrics such as distance or the gradient of the measurement function, both of which have no meaning in a nominal parameter dimension.

2.2 Polyhedral Accelerator Compilation

Translating a sequential program into a program that runs efficiently on an accelerator system is a compilation problem where

```
  for (int t = 0; t <= T; t++)
    for (int i = 1; i < 1023; i++)
      for (int j = 1; j < 1023; j++)
        if (t % 2 == 0)
S:        A[i][j] += B[i-1][  j] + B[  i][j+1]
                   + B[  0][  0]
                   + B[  i][j-1] + B[i+1][  j];
        else
T:        B[i][j] += A[i-1][  j] + A[  i][j+1]
                   + A[  0][  0]
                   + A[  i][j-1] + A[i+1][  j];
```

Figure 2: A simple 2D stencil kernel.

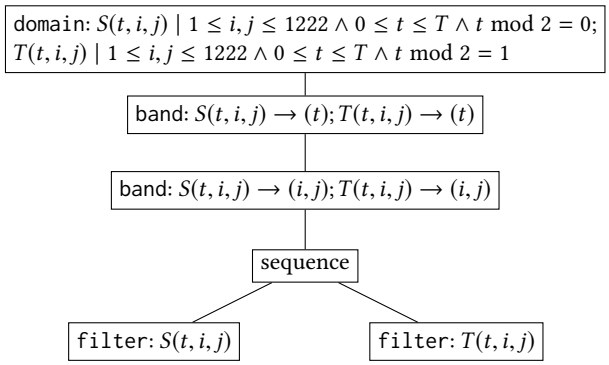


Figure 3: Schedule tree representation of the code in Figure 2.

polyhedral compilation techniques [18] have proven effective. PPCG [48] is a state-of-the-art automatic polyhedral mapper, which is used both in Facebook's domain specific deep learning compiler "Tensor Comprehension" [47] as well as in LLVM's Polly-ACC GPU compiler [20]. It's design is a good example for a polyhedral accelerator mapper. Starting from a DSL or (subsets of) imperative sequential C programs, an abstract model of the given computation is extracted [49]. This model captures three properties: 1) the *iteration space* describes the set of dynamic computations that are executed, 2) the *memory access relations* describe the data that is accessed for each dynamic computation, and 3) the *schedule* describes the relative execution time, as well as the hardware resource on which it is executed, for each dynamic computation. Optimizing transformations are expressed as transformations on either the memory access relations (2) or the schedule. After an optimal transformation has been found, an imperative AST [21] is regenerated.

At the core of polyhedral compilation are Presburger sets and relations. A Presburger set $S = \{\vec{i} \mid cons(\vec{i},\vec{p}), \vec{i} \in \mathbb{Z}^n, \vec{p} \in \mathbb{Z}^k\}$ is a $(n+k)$-dimensional vector space over $\mathbb{Z}$ with n variable dimensions and k designated parametric constant dimensions. The set of points contained in a set S is described by Presburger formulas over $\vec{i}$ and $\vec{p}$. Presburger formula are constructed from boolean operations ($\wedge, \vee, \implies, \not)$ over comparisons ($<, \le, >, \ge, =, \neq$) between quasi-affine expressions. An expression is quasi-affine if it is an integer constant, a variable, or a parametric constant. Sums and differences between quasi-affine expressions are also quasi-affine, products require at least one operand to be constant, and divisions and modulo operations require a constant divisor. A Presburger relation $R = \{\vec{i} \to \vec{j} \mid cons(\vec{i},\vec{j},\vec{p}), \vec{i} \in \mathbb{Z}^n, \vec{j} \in \mathbb{Z}^m \vec{p} \in \mathbb{Z}^k\}$ is defined accordingly. For readability, we allow tuples to carry "names".

Using the code in Figure 2 we illustrate how a simple C program is modeled with Presburger Sets. The iteration space (aka, its domain), is defined by $D = \{S(t,i,j) : 0 \le i,j \le 1222 \wedge 0 \le t \le T \wedge t \bmod 2 = 0; T(t,i,j) : 0 \le i,j \le 1222 \wedge 0 \le t \le T \wedge t \bmod 2 = 1\}$. The schedule $\theta = \{S(t,i,j) \to (t,i,j,0); T(t,i,j) \to (t,i,j,1)\}$ describes the execution order as specified in the original program. The relation $W = \{S(t,i,j) \to A(i,j); T(t,i,j) \to B(i,j)\}$ describes the set of memory locations written to, the relation $R = \{S(t,i,j) \to A(i',j') \mid i-1 \le i' \le i+1 \wedge j-1 \le j' \le j+1; T(t,i,j) \to B(i',j') \mid i-1 \le i' \le i+1 \wedge j-1 \le j' \le j+1\}$ describes the memory locations read. Many classical loop transformations such as interchange, tiling, fusion, and fission can be directly described as schedule transformations but, for more complex transformations such as GPU mapping, more structured schedule representation, e.g., schedule trees [50], have proven easier to work with.

A schedule tree at its root has a *domain node*, defining the iteration domain of the program. Below, there is a combination of the following nodes: 1) a *sequence node* defines the sequential execution of two sub trees, 2) a *filter node* limits the elements enumerated in sub tree to the elements specified in the provided filter, and 3) a *band node* defines a set of partial schedule dimensions.

Figure 3 illustrates the schedule tree corresponding to the C code in Figure 2. It consists of a domain node at the root We then see two band nodes. The first provides the schedule for the time loop, the second the schedule for the two space loops. Finally, there is a sequence node which sorts the two statements at the innermost level and filter nodes which ensure that each branch of the sequence node only executes the statement indicated in the filter node.

Polyhedral loop modeling techniques provide facilities that make accelerator mapping easy. To automatically generate GPU code, Polly-ACC [20] and the underlying polyhedral mapper PPCG [48] start from a schedule tree representation of the original program, and transform this tree gradually into a GPU-aware program. As a first step, a Pluto-style [9] scheduler is run on the schedule tree aiming to maximize the number of outer-parallel loop nests. Subsequently, band nodes that model parallel loop nests are mapped to both thread identifiers and block identifiers. These parametric constants are used to relate individual statement instances in the parallel loop round-robin to subsequent threads and blocks. In case there are more computations than threads or blocks, remaining computations are mapped to a thread and block id equal to the iteration id modulo the maximal thread or block count. After the actual computation has been mapped to the GPU, there are various additional optimizations that become possible. The memory access behavior can for example be improved by introducing shared or private memory. While GPU mapping is easy, many choices (e.g, tile sizes, use of shared memory, ...) need to be made to derive a good GPU mapping. Today, either simple heuristics or auto-tuning is used.

3 HIERARCHICAL ONLINE AUTOTUNING

In this section we will introduce our autotuning framework as well as our core contribution, hierarchical online-autotuning.

3.1 Search-Based Online-Tuning

We implemented the hierarchical tuning technique as part of our online autotuner `libtuning`. As online tuner, it is required to be low-overhead and to integrate easily with performance critical applications. It ships with a small array of search techniques: Nelder-Mead[34], ϵ-Greedy[36], Full-, and Random-Search. Nelder-Mead and ϵ-Greedy are, respectively, the defaults for non-nominal and nominal search spaces. The library supports arbitrary parameter types, and only distinguishes between nominal and non-nominal parameters. This is possible because of a strict abstraction between the application-facing parameter space and the search algorithm-facing search space. In this abstraction layer, we map the arbitrarily typed and usually bounded application parameters into a real-valued, unbounded space for the search to operate on. This not only simplifies applicability of the library, but also reduces numerical problems. The space mapping is controlled by the user. For the default numerical (i.e. integer and float valued) parameters, this mapping simply means rounding. If the parameter is bounded, the default mapping includes mirroring it at its boundaries, thus producing a periodic search space. Because of this feature search algorithms do not have to deal with boundary conditions. This approach preserves both local and global extrema, but makes the measurement function discontinuous at the boundaries.

The distinction between nominal and non-nominal parameters is a key feature in `libtuning`, and their handling is entirely transparent. The autotuner will construct separate search spaces for either class. This behavior is a major distinction from existing tuning approaches, which widely ignore the difference between the parameter classes. Few search algorithms are able to deal with nominal parameters correctly, the most prominent probably being genetic algorithms [36]. Nevertheless, in the existing literature, researchers apply general search based tuning techniques to this class, and still report success (e.g., OpenTuner[2], which Ansel et al. apply to GCC command line arguments).

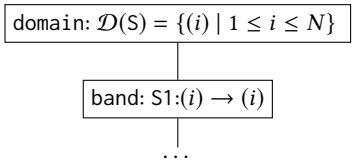
Separating the different search spaces of course does not imply independence, since parameters are generally interdependent. If they were not, tuning them separately with individual autotuners, thus reducing the dimensionality of the search space, would produce results more quickly. Hence, independently finding a partial configuration for both spaces and combining them into a total configuration is not equivalent to finding a total configuration on the global space. Restoring this equivalence requires reestablishing parameter dependences. We realize this by maintaining separate search states for every nominal parameter configuration. Assume, for instance, two parameters $\tau_N = \{0, 1, 2\}$ and $\tau_R = [1, 0] \subset \mathbb{R}$, where τ_N is nominal. To produce a global configuration, we first determine a partial configuration C_N for τ_N using an appropriate search algorithm. To obtain a partial configuration for τ_R, we instantiate the search algorithm of choice $|\tau_N| = 3$ times. Using C_N, we then pick the according instance and use it to produce a partial configuration C_R. This gives us a total configuration $C = (C_N, C_R)$ which correctly respects relationships between dependant parameters.

3.1.1 Parameter Dependences & Search Space Graph. While explicitly differentiating between nominal and non-nominal parameters allows us to reason about them more flexibly, it does not help reduce the dimensionality of the search space, and comes at the price of an increased memory footprint. However, since we now have purely nominal spaces, there is an interesting peculiarity of nominal parameters that frequently arises in practical applications: Often there are explicit relationships defined by the application, e.g., when tuning thread count and platform selection, the number of GPU threads is not needed if the GPU is unused.

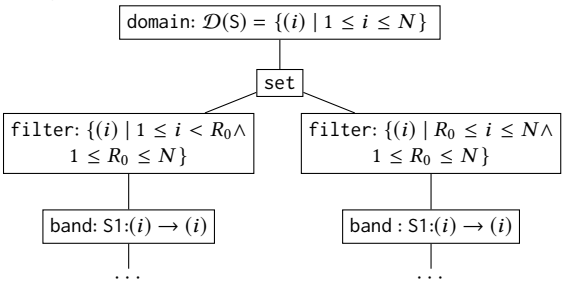
To exploit the dependencies, we decompose the global search space into a set of disjoint local spaces which we call the search space graph. The `libtuning`-interface allows application developers to specify dependency constraints, from which the autotuner automatically constructs the search space graph. To illustrate this idea we refer to Figure 1. The figure shows a subset of the decomposition of the search space for a partitioned 2D kernel, offloaded to CPU and GPU. Annotations on the edges represent dependencies, blue round-cornered boxes contain only nominal parameters, orange rectangular boxes only non-nominal parameters. Computing the decomposition is trivial and is implemented greedily. To find a configuration of the global search space the subspaces are tuned *hierarchically*, finding partial configurations by walking the graph from its root along all edges with satisfied constraints. Visited nodes are called the *active spaces*. Note that for presentation purposes the figure shows every non-nominal parameter only in exactly one box. In practice, non-nominal parameters are part of multiple subspaces in general, because when the dependence constraints for two such parameters are satisfied, they cannot be legally tuned separately. In the example, the parameters R_0, R_1 are thus present in every non-nominal subspace in practice. As a consequence, there is only a single active non-nominal space.

Because of similar reasons, configurations in dependent spaces cannot be selected independently from their parents. We handle this issue by lazily instantiating and retaining state of the search for the dependent space for every parent nominal configuration. This entails potentially consuming large amounts of memory, although the effect is moderate in our experience because of the reduction of dimensionalities: The nominal subspace of the graph in Figure 1 contains 7^4 configurations, instead of the 16^4 nominal configurations of the unconstrained global space. Nevertheless we expect that for applications in which there are no or too few exploitable dependencies the memory consumption may outweigh the benefits of our approach .

3.1.2 Runtime Feedback for Driving the Search. There is an additional opportunity to further decrease the numbers of configurations we need to sample, especially when we are applying tuning in the context of parallelizing compilers. In addition to the static information used to thin out the search space, we can use dynamic information at application runtime to identify uninteresting configurations without actually sampling them. As an example, consider autotuning offloading of a matrix multiplication kernel to a GPU, i.e. deciding whether or not the GPU should be used. If the input



(a) Original schedule: 1D domain with trivial coincident band.



(b) Partitioned schedule: Adds set of filters and a splitting point R_0.

Figure 4: Schedule partitioning.

matrix size is 4×4 elements, we can safely assume that choosing a GPU over a CPU is not optimal.

A configuration rejection is handled differently within nominal and non-nominal nodes. Since nominal search algorithms usually need to deal with an enumeration of all configurations, a rejection means simply skipping one. In a non-nominal search, skipping a single point is not that easy. Thus, handling may be implemented by the user, but we provide a non-stationary penalty function[24] by default.

3.2 Autotuning Polyhedral Parallelization

In this paper, we apply online-autotuning to Polly-ACC, which we extend for tunable heterogeneous partitioning. Fundamentally this means that we generate parallel code for both CPU and GPU, parametrically partitioning the original loop nest. Size and target platform for each partition can be determined dynamically at application runtime.

In the schedule tree we first identify kernel candidates for partitioning by finding leading coincident bands. To illustrate the partitioning process, consider the schedule tree example in Figure 4. Figure 4a shows a schedule tree for a static control part (SCoP) containing only one outer loop and a simple schedule mapping every iteration onto an individual timestamp. Assuming the corresponding leading band is coincident, we partition it according to the example in Figure 4b: Between the leading coincident band and its parent, we insert a `set` node, whose filter children will implement the partitioning. Although this is trivially generalizable to an arbitrary number of partitions, for simplicity's sake we only split the domain into two partitions. We introduce a new parameter R_0 to define the partition splitting point, subject to the same affine constraints as the dimension it is splitting. This parameter then becomes a new upper and lower bound in the higher and lower partition, respectively. The original `band` node is duplicated, and becomes a child to every new filter node. This process is repeated recursively for every leading coincident dimension that is to be partitioned. Consequently, if we partition d dimensions, we will end up inserting $2^d - 1$ set nodes and producing 2^d duplicates of the original band. The parameters R_i are tunable parameters.

To implement platform selection, we follow exactly the same path. For every `band` node we create, we insert a new `sequence` node as its immediate parent between the `band` node and the newly created set filter. In the two filters of the new `sequence` node, we introduce another new parameter P_i for platform selection, $0 \le P_i \le 1$, and insert the constraints $P_i = 0$ and $P_i = 1$ accordingly. Again duplicating the `band` node, we finally parallelize it for the GPU by applying the PPCG mapping algorithm to the band where $P_i = 1$. For its sibling, however, we insert a `mark` node, which we use during AST generation as the starting point of generating OpenMP code for CPU parallelization. GPU code generation uses the existing capabilities of Polly-ACC with one important extension. Previously, the compiler could safely assume that the GPU is the only entity reading and writing data while the kernel is running. Thus, Polly-ACC always copies arrays in their entirety, independent of how much of the array is actually accessed in the kernel. With multi platform partitioning, however, this becomes infeasible. Different partitions may access an array concurrently and write different parts of it. These writes must not be overwritten by data transfers. Hence, our solution analyzes accesses to an array precisely, producing a set of (strided) slices of accessed data. Intersecting these slices with a partition's schedule yields the exact regions of memory accessed by this partition, which we then copy individually between platforms.

3.2.1 Multiversioning for Platform Parameters. So far, we have extracted two classes of tunable parameters from the parallelized program. However, there are many additional tuning opportunities in this process. For instance, tuning the number of threads on a CPU partition is straightforward, since the OpenMP runtime library simply accepts this as a parameter. Additionally, we elicit several more parameters from the GPU mapping process. When calling into the PPCG mapper, multiple GPU platform parameters are exposed as configurable options, such as the maximum tile size, the block and grid layout, or which memory architectures to use. Naturally we wish to leverage the tunability of these parameters; however, it comes with a cost: Affineness is a necessary precondition of polyhedral modeling and code generation. While these platform parameters are configurable during mapping, they cannot be variable (and thus runtime-configurable), since that would make the model constraints non-affine. To circumvent this unfortunate issue we resort to multi-versioning. While we cannot change these parameters at application runtime, we nonetheless can generate a large amount of different configurations during compile time, and use autotuning to choose between configurations. In fact, from the perspective of the autotuner, there is no discernible difference between those two options. On the other hand, generating a few hundred or thousand different versions of the code enormously increases compile time and binary size. As a data-point, the size of the parallelized binary for the Polybench GEMM benchmark that we use in our evaluation is 9MiB.

In summary, the tuning parameters we generate are listed in Table 1, along with their respective semantics. The inter-parameter dependences induced by this choice are straightforward: The individual instances of the platform parameters for every partition,

Table 1: Tuning parameters for polyhedral parallelization.

Parameter	Semantics
$P_i \in \{GPU, CPU\}$	Pick the platform for partition i.
$R_d \in (0, 1) \subset \mathbb{R}$	Pick the sizes of partition at depth d.
$T_i^{CPU} \in [1, C]$	#OpenMP threads on the CPU (up to #cores C in the system).
$T_i^{GPU} = 32k, k \in [1, 8]$	#points/tile (#GPU-threads/block).
$S_i \in \{\texttt{true}, \texttt{false}\}$	Allow GPU shared memory.
$U_i^T \in \{\texttt{true}, \texttt{false}\}$	Unroll accesses to tile points.
$U_i^S \in \{\texttt{true}, \texttt{false}\}$	Unroll accesses when copying the tile into shared memory.
$B_i = 2^k, k \in [0, 5]$	Height of the 2D block (GPU grid size).

e.g. CPU-Threads or Tile-Sizes, naturally depend on the platform choice for this partition. Similarly, unrolling accesses to the shared memory only makes sense if shared memory is enabled. The search space graph for these parameters of a 2D kernel is partially shown in Figure 1, omitting for clarity the subgraphs for all but the first partition, which are structurally identical.

3.2.2 Runtime Feedback Rules. Beyond parameter dependences, we use the polyhedral representation of the partitions to derive runtime feedback rules to further aid the search. For a polyhedron, we first compute three metrics, $\mathcal{M}_{compute}, \mathcal{M}_{memory}, \mathcal{M}_{reuse}$, using Barvinok's algorithm for point counting in lattice polyhedra[5, 51]. The *compute volume* $\mathcal{M}_{compute}$ is simply the number of points in the polyhedron, which. corresponds to the number of dynamic statement instances. To compute the *memory footprint* of a single access $\mathcal{A}$, we count the points in the image of the access functions for the scheduled domain of $\mathcal{A}$. The metric $\mathcal{M}_{memory}$ is then the sum of all memory footprints for all accesses in the kernel. The $\mathcal{M}_{reuse}$ metric is computed similarly to the memory footprint, counting the number of accesses to the same element within an outer iteration. Note that cardinality of an integer set is *parametric*, in the sense that in general it is a polynomial expression of the tuning parameters and kernel inputs. To evaluate these metrics, we hence generate code to compute the expression using the actual parameter configurations and inputs at runtime.

We define three truth-valued rules, which, if evaluated to `true`, cause a configuration to be rejected:

Min-Compute A minimum compute to enable the GPU:

$$P_i = \texttt{GPU} \rightarrow \max_{R_k} \mathcal{M}_{compute_i} < \rho$$

Compute Intensity GPU kernels are compute bound

$$P_i = \texttt{GPU} \rightarrow \frac{\max_{R_k} \mathcal{M}_{compute_i}}{\max_{R_k} \mathcal{M}_{memory_i}} > \alpha$$

Min-Reuse Sufficient reuse for shared memory:

$$S_i \rightarrow \max_{R_k} \mathcal{M}_{reuse_i} \leq 1$$

Note that in all of these rules we use a max operator. Because of this, rules depend completely on nominal parameters, which allows the rejection of configurations early and within the nominal spaces. Rejecting a configuration in a nominal space is significantly simpler from a search algorithm perspective, and less error prone. Although fundamentally maximizing a polynomial over a bounded space is hard and expensive, we can greatly simplify the computation: because of how we introduce partitioning parameters and because everything is required to be affine in a polyhedral model, these parameters only ever appear as linear terms in the expression. Thus, we can substitute them for their lower and upper bounds, and maximize over the result at runtime.

4 EVALUATION

To evaluate our process, we apply polyhedral parallelization and runtime autotuning on `blas` and `kernels` linear algebra packages of the Polybench 4.2.1-beta benchmark suite. For all Benchmarks we use the predefined `EXTRA_LARGE` input sets. The performance results we present in this paper were obtained on a dual-socket 12-core Xeon machine with 2×SMT, clocked at 2.6GHz and equipped with an NVIDIA Tesla P100 GPU. To simplify this evaluation, we made small modifications to the original benchmarks. By changing Polybench's memory manager to allocate pinned memory on the host, we simplify the complexity of data movement. While this saves us from having to implement complex management of CUDA contexts per kernel partition for asynchronous data transfers, it also has an effect on benchmark results: placing buffers in pinned host memory enables DMA transfers between GPU and host. Additionally, we wrap Polybench's invocation of the kernel into a loop to create an online tuning loop. Lastly, we had to overcome a limitation of our current implementation of the parallelizer, which can only handle a single parallel kernel per SCoP. Hence we modified the structure of the kernel loops by fissioning multi-statement loops which after scheduling would produce a sequence of parallel bands. To break the new sequence of loops into seperate SCoPs we inserted memory fences as seperators.

In the following, we report overall speedup results for the Polybench benchmarks. To specifically assess the benefits of our hierarchical tuning approach for online tuning, we then present a case study of the `gemm` kernel and analyse the induced search space and the tuning behavior in detail. We compare hierarchical tuning against OpenTuner and classical Nelder-Mead search, and thus indirectly against ActiveHarmony which is based on it. With the exception of OpenTuner, all recorded time measurements always include all overhead of the autotuner, although profiling shows the overhead is generally negligible. We ensure kernel compilation overhead is not included in the measurements.

4.1 Overall Performance Results

In Figure 5 we compare the runtime of the polybench kernel after parallelization and tuning with our approach to the runtime using either Polly (i.e., `clang -O3 -polly`) or Polly-ACC[1]. Polly-ACC uses its default values $T^{GPU} = 32, S = U^T = U^S = \texttt{false}, B = 1$. Additionally, we also used OpenTuner to optimize our parallelized program. The plot shows speedups of our approach over three baselines, OpenTuner, Polly, and Polly-ACC and also includes a `clang -O3` baseline for `gemm`. To determine the kernel runtime for our approach, we parallelize the kernel and then tune it for 300 iterations, which is more than enough to guarantee that the tuner is converged. We repeat this run 20 times, reporting the median runtime of the last 20 iterations. The reference runtime is the median runtime computed from 20 kernel invocations for

[1]Both at svn revision r310059

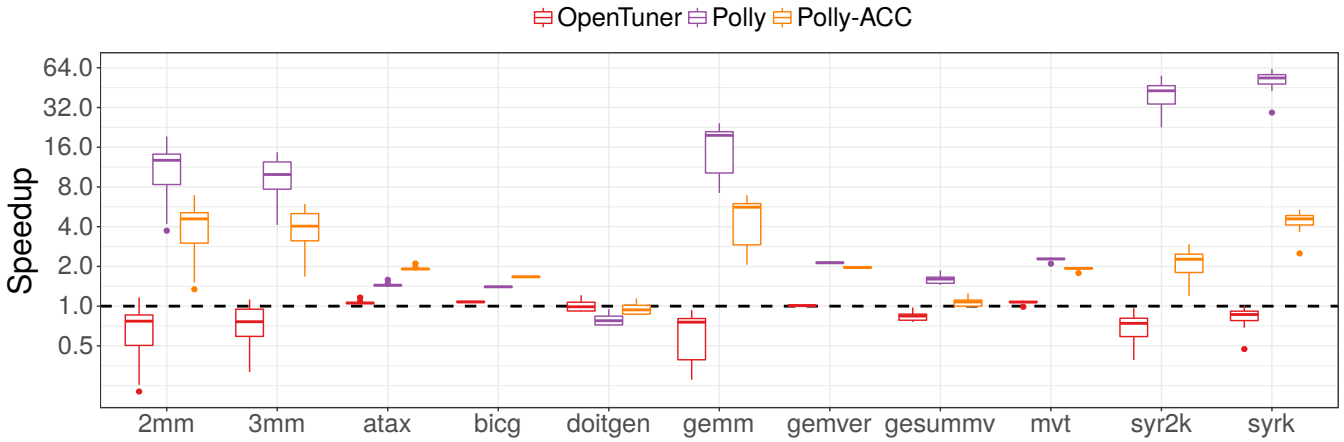


Figure 5: Speedups for linear algebra benchmarks over three baselines: OpenTuner, Polly, and Polly-ACC.

both Polly and Polly-ACC. Comparing to Polly-ACC, we achieve a speedup of up to 6× on the `gemm` kernel. On the other hand, for the benchmark `doitgen` we cause a minor slowdown of 6%. This is because this benchmark does not benefit much from GPU parallelization, and the additional overhead added by the multi-platform management kills possible performance gains. In all cases except `doitgen` our approach outperforms Polly by a large margin, up to 63× for the `syrk` kernel.

Comparing our approach against Opentuner, we see in most cases our tuning achieves on-par configurations. We compare our tuning results against the best configuration found by OpenTuner among 10 tuning repetitions for 100 iterations. For further details on how we use OpenTuner for tuning we refer to subsection 4.4. We achieve a geo-mean slowdown of 0.84 and our results are worst in the `3mm` and `syr2k` benchmarks. Interestingly, our approach also outperforms OpenTuner's configuration in a few cases, which indicates that we did not run OpenTuner for a long enough time. The limit of 100 iterations was selected because after that point almost all experiments using our approach made no further improvements.

Another interesting result is visible in `2mm`, `3mm` and `gemm`. Whereas we outperform both Polly and Polly-ACC, the tuned runtimes exhibit an enormous spread. Although this is not visible in the box plot, the configurations are not in fact spread normally around the mean but appear in a small number of clusters. This points to the existence of plateaus in the search space. Since the same can be observed using regular Nelder-Mead search, we are confident that our hierarchical tuning approach is not the cause of this behavior.

4.2 `gemm` Case Study: Precursory Exploration

As a precursory evaluation, we attempted to gain an understanding of the global search space. A full exploration of this space is infeasible because of its size, we resorted to subsampling this space by severely restricting parameter ranges. In addition to the ranges defined in Table 1 we set $U^S = U^T = 0$, $R_d \in \{0.25, 0.5, 0.75\}$, $B = 1$, $T^{GPU} \leq 512$, $T^{CPU} = 40$. Note that we also dropped the partition subscripts: All the per-partition parameters are shared across all partitions here. Exhaustively searching this space, we find that the optimal configuration with a runtime of 0.48s achieves a speedup of 1.5× over Polly-ACC, a speedup of over 7.8× over Polly and a speedup of over 22.4× over `clang` at the `O3` optimization level. This optimal configuration is however not truly unique; there are tens of configurations with a performance similar enough that the difference could be noise. We were unable to identify a pattern among these configurations.

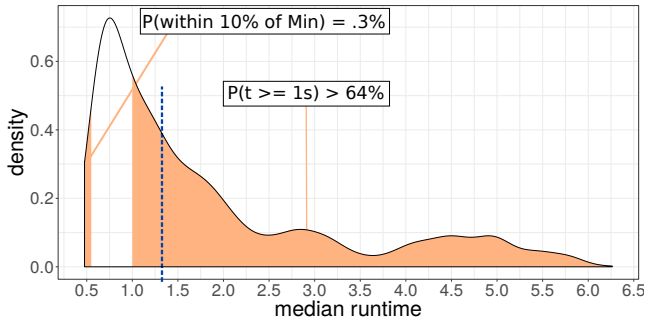
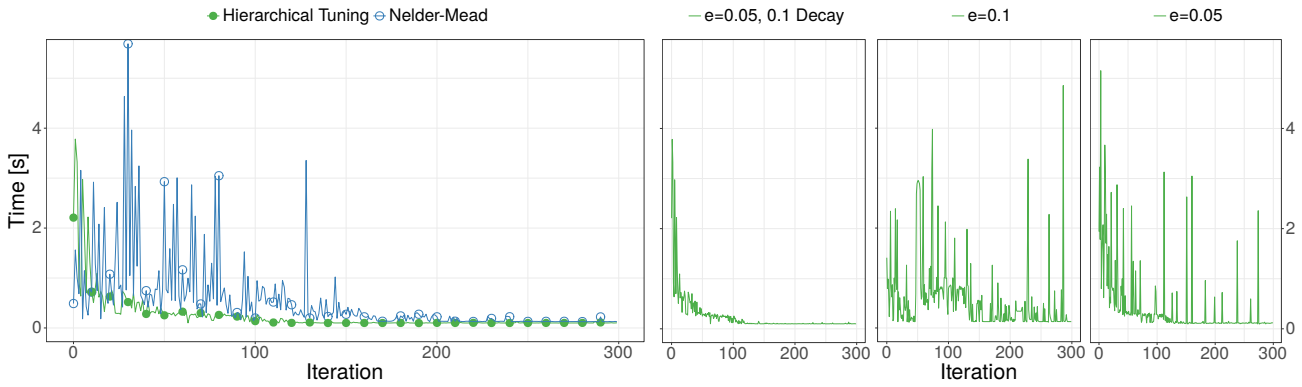


Figure 6: Runtime density of the reduced search space.

To further strengthen the claim that this search space is an interesting candidate for autotuning, we analyzed the runtime distribution. Figure 6 shows the runtime density. The median runtime at 1.3s is shown as a vertical dashed line. We further highlight two interesting regions in the plot. The leftmost solid-colored area shows the probability of guessing a configuration that is within 10% of the global optimum: If using random search, the probability of finding a configuration which is at most 10% worse than the optimum is merely 0.3%. On the other hand, the probability of being *at least* twice as slow as the optimum using random search is above 64%, as shown by the rightmost solid-colored area of the plot. As a consequence, we see that it is possible to achieve a noteworthy speedup over the default, but it is neither reasonably attainable by full nor random exploration.

4.3 `gemm` Case Study: Hierarchical Autotuning

We analyse the parallelizing and tuning of the `gemm` kernel on the full search space. We compare hierarchical autotuning with classical Nelder-Mead search on the full search space. In our approach,



(a) Tuning traces of Hierarchical Tuning and Nelder-Mead-Search. **(b) Sensitivity to choices of ϵ-Greedy parameters.**

Figure 7: Search traces of the autotuning process.

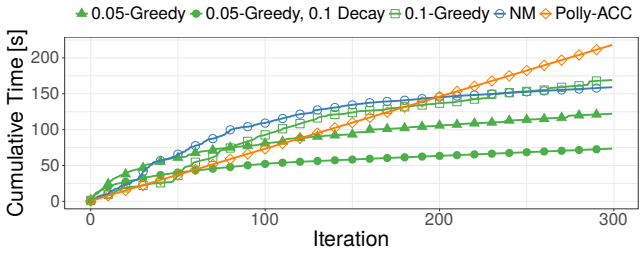


Figure 8: Amortization time of the search techniques.

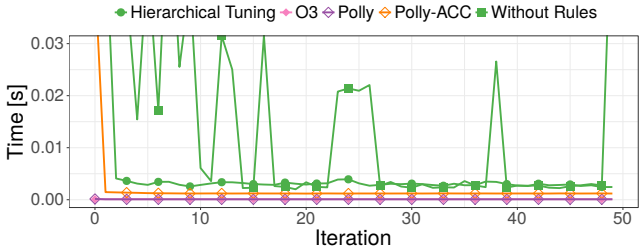


Figure 9: Tuning traces using small matrix size.

we use ϵ-Greedy search on the nominal search spaces, with a non-stationary $\epsilon = 5\%$ and an ϵ decay of 10%. I.e. , in every iteration, ϵ is reduced by 10%. In the non-nominal spaces we fall back to classical Nelder-Mead search. For the hyperparameters in the runtime feedback rules we set $\rho = 150000000, \alpha = 2$. The `gemm` case study was run on a Xeon E5-2680 v3 machine with a Tesla P100 GPU.

To attain a deeper understanding of the behavior of the autotuner, and especially the contribution of our novel search technique, we look at full traces of the autotuning process. Figure 7a compares a Nelder-Mead search on the full search space with a trace of our hierarchical tuning approach. While both searches find a configuration with similar performance, we can see that our approach converges much faster: Whereas Nelder-Mead converges at around 200 iterations, our search is completed at about 100 iterations. On top of that, our search is not only quicker to converge, but also samples much fewer bad configurations. In Figure 7b we show the effect of the choice of the parameters of the ϵ-Greedy search. It is easily visible that both the convergence rate as well as the chance of sampling bad configurations are highly sensitive to the values chosen for ϵ and the ϵ decay. The effect of this becomes even more apparent when considering amortization time. Amortization time is the time required for tuning to pay off, considering all configurations sampled over time. We compute it by integration over the traces in Figure 7. The results are shown in Figure 8, including Polly-ACC as reference. The plot demonstrate that our tuning approach begins to pay off after only 50 iterations. On the other hand, picking an unfortunate ϵ value leads to an even worse amortization than Nelder-Mead.

So far, the runtime feedback rules did not contribute to the results we have shown. Since the input matrices contain on the order of 4M elements, and the matrix multiplication algorithm performs $O(n^3)$ operations on $O(n^2)$ memory, neither of our rules apply. To evaluate the benefits provided by the rules, we look at the effect of smaller input sizes. Using 500 element matrices, we obtain traces as shown in Figure 9. With such small matrices, the kernel is obviously not a candidate for offloading, which we can see from the 25× slowdown compared to `clang -O3`. Even though Polly-ACC does perform offloading, it is still faster than our approach by a factor of two, due to much smaller overhead of launching only a single kernel instead of four. Although we are outperformed by the reference compilers by a large margin here, this is by no means an indication against our approach: Our compiler does not implement executing the original code or a single platform version. This can easily be implemented however, controlled via another tuning parameter. The key thing to note is that our autotuner converges within two iterations, because the runtime feedback rules immediately reject offloading anything to the GPU. Comparing this with the No Rules series, which is the same tuner, just not evaluating the runtime rules, the benefit is clearly visible. Nevertheless, this effect is not for free: the second iteration, which is off the chart here, is 20× more expensive than the converged one, because the autotuner spends tens of iterations until it finds a configuration that satisfies the rules.

4.4 Comparison against OpenTuner

Lastly, we compare the performance of our approach to the performance of OpenTuner. In Figure 5 we saw that our approach

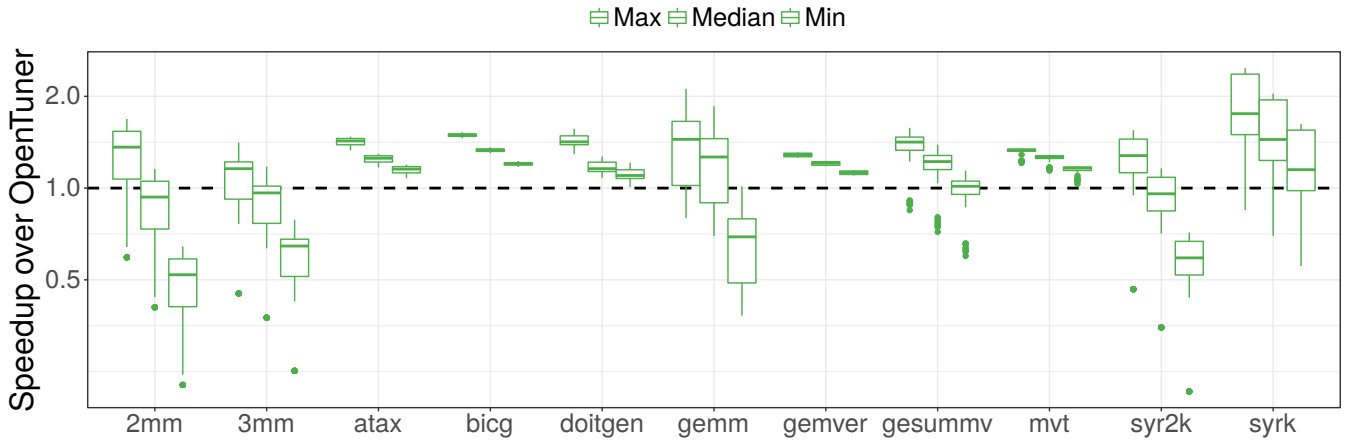


Figure 10: Comparing the time required to execute 100 tuning iterations between OpenTuner and Hierarchical Tuning. Speedup baselines are the maximum, median, and minimum times taken by OpenTuner, respectively.

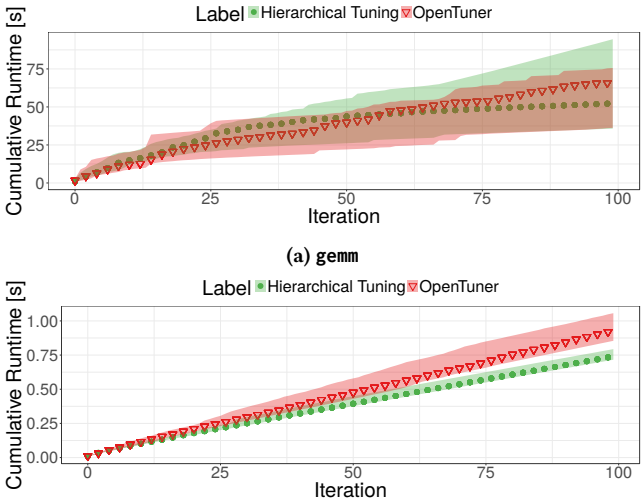


Figure 11: Execution timeline of the `gemm` and `atax` benchmarks for the first 100 iterations using both OpenTuner and Hierarchical Tuning. Dots are median times required to reach an iteration, the ribbon boundaries are the min and max.

can be on par with OpenTuner regarding the final configuration found. In the cases OpenTuner's best configuration outperforms Hierarchical Tuning we attribute this to the fact that we prioritize finding a configuration quickly. To show that we achieve faster cumulative runtimes through better amortization we compare the runtime behavior of the tuners in this section.

Because of its Python implementation, OpenTuner cannot be easily used as an online tuner operating within the tuning loop we inserted into the benchmarks. To obtain measurement samples for the tuner we thus allow loading parameter configurations from a file, which remains unchanged during the tuning loop. We reduce the trip count of the loop to 5. Then we use OpenTuner to create a configuration file containing the next parameter setting it wants to sample and then run the benchmark with this configuration. Note that this implies that unlike in our case, the time required to determine and set the new configuration is not included in the measurements. From the five acquired time measurements we discard the first two and give the average of the remaining three as feedback to the tuner. We run OpenTuner using its default search strategies for 100 iterations and repeat the tuning ten times.

In Figure 10 we compare the time required for OpenTuner and Hierarchical Tuning to perform the 100 tuning iterations. For OpenTuner this time is the sum of measurement samples and does not include OpenTuner's own runtime and the time spend starting the benchmark applications. The plot shows the speedups of our solution against three baselines, the Min, Median, and Max time required by OpenTuner. In most cases, we outperform OpenTuner. We achieve geometric mean speedups of 1.19 over the Median baseline and 1.70 over the Max baseline. Over the Min baseline we observe a geometric mean slowdown of 0.86. In the best case, which is `syrk`, we complete the 100 iterations in half of the time required by OpenTuner.

To better see the tuner behavior in the `gemm` and `atax` benchmarks, we show the full tuning timelines in Figure 11a and Figure 11b, respectively. Dots denote the median time required to reach an iteration, the ribbon boundaries are the min and max times. The plot shows dots only for even iteration numbers to improve readability. In the case of `gemm`, we see that the ribbons for both methods span over a wide range of runtimes. This indicates plateaus within the `gemm` search space. The median cumulative runtime of Hierarchical Tuning is lower than OpenTuner after about half of the iterations. For the `atax` benchmark we see that the two ribbons diverge. Most likely this is because OpenTuner does not discover a configuration that is as good as the one found by Hierarchical Tuning, as we saw in Figure 5.

5 RELATED WORK

Since autotuning became popular in the early 2000s with the ATLAS library [54], it has been applied in most branches of computing research and industry. Most autotuners are specifically tailored to

their respective application, and there are only a handful of general purpose tools available. Noteworthy is ActiveHarmony [44] and the more recent OpenTuner [2]. Being an online tuner, ActiveHarmony is the most closely related to our `libtuning`, the key difference being that ActiveHarmony is designed for distributed applications offering a central tuning server. Since OpenTuner is implemented in Python, it is rarely deployed in online scenarios. Its key feature is that instead of requiring users to choose a fitting search algorithm for their problem, OpenTuner makes this choice a tuning parameter. The OpenTuner article is also, to the best of our knowledge, the first to discuss nominal parameters. Pfaffe et al. [36] investigate tuning these parameters more deeply, but focus only on spaces containing a single such parameter. In a recent article, Rasch et al. introduce ATF [38], a directive-based general purpose tuning framework. Unlike previous approaches, ATF recognizes the existence of parameter interdependences, and allows users to express relatively arbitrary constraints. These constraints are in fact more powerful than the mechanism we introduce, but do not allow for the strict search space decomposition we use. Instead, Rasch et al. form the search space by enumerating all legal configurations. This method works for them because their search space is discrete. The search space of our application is large and real-valued, rendering Rasch et al.'s approach infeasible for us.

5.1 Tuning Compiler Optimizations

Finding good sequences or configurations of compiler transformations is difficult and parallelizing compilers are excellent beneficiaries of autotuning. This includes both empirical tuning as we apply it here as well as machine learning or model based techniques (e.g, [3, 7, 10, 52]). While there is a large body of work that automatically maps parallelizable codes to accelerators (e.g., [6, 15, 31, 35, 49]), most closely related to us is probably CHiLL. CHiLL [14] is a polyhedral compiler and loop optimizer. Originally, it uses a simple search strategy that systematically tries all configurations. In 2009 Tiwari et al. pair it with ActiveHarmony [45], also taking into account parameter constraints. They extend this towards an online tuning scenario [46], in which they evaluate multiple code variants in parallel on an HPC system. With CUDA-CHiLL [40], the system has also been expanded to generate parallel accelerator code. Similar to our approach, CUDA-CHiLL produces multiple code variants, and uses autotuning to navigate the resulting space. Additional noteworthy candidates that apply or at least enable autotuning are the ROSE [28] kernel outliner, which itself uses CHiLL and ActiveHarmony, the Cetus [16] source-to-source parallelizer, which generates OpenMP code and tunes it using "Combined Elimination", or Bones [35], which generates CUDA code.

There is also related work in the context of domain-specific languages (DSLs). Halide [37] by Ragan-Kelley et al., is a language for image processing pipelines. From high level algorithm and schedule descriptions, the Halide compiler generates parallel and accelerator code. Schedules can also be produced using autotuning, originally implemented using the PetaBricks tuner. PetaBricks [1] is itself a DSL aiming to optimize algorithmic choice. Most recently, the LIFT project [22, 43] positioned itself as a data-parallel intermediate representation for higher level DSLs. Using ATF and OpenTuner, LIFT generates high performance code for accelerators.

5.2 Polyhedral Performance Modeling

Polyhedral techniques are well-suited to deriving models for both tile size selection and cache miss models. Tile size selection with analytical models has been proposed in many variations [11, 23, 27, 29, 32, 39, 41, 42, 55]. While the earliest papers claimed "optimality" due to their analytical formulation, over time models became more elaborate, but often specialized for a specific problem. By integrating precise domain knowledge, Yotov et al. [55] and later Low et al. [29] show that analytical modeling can reach near-peak performance, if both algorithm and hardware are well understood. Model-driven empirical search [13, 17, 19, 25, 26, 26, 53] assumes that modeling hardware in all its complexity is too difficult. Hence, they use iterative search together with analytical models to reduce the size of the search space. Using a hierarchical model to reduce the cost of the search process itself, as we do, has not been explored yet. In the context of cache modeling, integer points counting [5] was predicted to allow for perfect cache miss models [12], but computing these models precisely was considered too expensive. Alternative models [8] based on reuse distance as a proxy for cache misses have been developed and approximations of the Barvinok algorithm [30] have been considered. Just recently an expensive, but complete cache model for affine programs has been presented [4]. Statistical methods based on machine learning have also been used for performance modeling with Yuki et al. [56] learning techniques to automatically derive tile size selection models.

6 CONCLUSION

In this paper we introduced hierarchical online-autotuning, a powerful technique to improve convergence in online tuning scenarios by exploiting structure and redundancies in the search space. We integrated hierarchical tuning with a novel polyhedral parallelization tool for heterogeneous systems. By parallelizing a kernel for sharing the work load across accelerators, this tool can exploit the heterogeneous system more effectively than a single-platform parallelizer. Using autotuning, we simultaneously optimize the distribution of work and various platform specific parameters. We evaluate our approach on the Polybench linear-algebra `blas` and `kernels` packages and are able to report substantial speedups over Polly and Polly-ACC. On the `gemm` kernel we achieve a speedup of 63× over Polly, and 6× over Polly-ACC, a state-of-the-art polyhedral parallelizer for GPUs. Additionally, our hierarchical tuning approach improves convergence of the parameter search by up to 1.7× over OpenTuner. These results show that it is possible to exploit multiple accelerators in a single system effectively. We further demonstrated that with hierarchical tuning deploying always-on online-autotuning is practical and promises to optimize programs for the actually used hardware and inputs.

ACKNOWLEDGMENTS

We thank Prof. Walter F. Tichy for his fruitful feedback. This work is partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project 299215159 , the Swiss National Science Foundation (Ambizione - PZ00P2_168016), ARM Holdings plc and Xilinx Inc. through Polly Labs, as well as a HiPEAC collaboration grant. Compute resources have been provided by CSCS.

REFERENCES

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 09)*. ACM, New York, NY, USA.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT 14)*. ACM, New York, NY, USA.

[3] Prasanna Balaprakash, Robert B. Gramacy, and Stefan M. Wild. 2013. Active-Learning-Based Surrogate Models for Empirical Performance Tuning. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*.

[4] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P. Sadayappan. 2017. Analytical Modeling of Cache Behavior for Affine Programs. *Proceedings of the ACM on Programming Languages* 2, Issue POPL, Article 32 (Dec. 2017).

[5] Alexander Barvinok. 2008. *Integer Points in Polyhedra*. European Mathematical Society.

[6] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. 2010. Automatic C-to-CUDA Code Generation for Affine Programs. In *International Conference on Compiler Construction*. Springer.

[7] James Bergstra, Nicolas Pinto, and David Cox. 2012. Machine Learning for Predictive Auto-Tuning with Boosted Regression Trees. In *Innovative Parallel Computing (InPar)*. IEEE, Washington, DC, USA.

[8] Kristof Beyls and Erik D'Hollander. 2001. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, Vol. 14.

[9] Uday Bondhugula, A Hartono, J Ramanujam, and P. Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*.

[10] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO 07)*. IEEE Computer Society, Washington, DC, USA.

[11] Jacqueline Chame and Sungdo Moon. 1999. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 13th international conference on Supercomputing*. ACM.

[12] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck. 2001. Exact analysis of the cache behavior of nested loops. *ACM SIGPLAN Notices* 36, Issue 5 (2001).

[13] Chun Chen, Jacqueline Chame, and Mary Hall. 2005. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization (CGO 05)*. IEEE.

[14] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A Framework for Composing High-Level Loop Transformations*. Technical Report.

[15] Marvin Damschen, Christian Plessl, Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, Sebastian Meisner, Achim Lösch, and others. 2015. Transparent Offloading of Computational Hotspots from Binary Code to Xeon Phi. *Proceedings of the 2015 Conference on Design, Automation and Test in Europe (DATE)* (2015).

[16] Chirag Dave and Rudolf Eigenmann. 2010. Automatically Tuning Parallel and Parallelized Programs. In *Languages and Compilers for Parallel Computing (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg.

[17] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R Clint Whaley, and Katherine Yelick. 2005. Self-adapting linear algebra algorithms and software. *Proc. IEEE* 93, Issue 2 (2005).

[18] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, Boston, MA.

[19] Basilio B Fraguela, Martın G Carmueja, and Diego Andrade. 2005. Optimal tile size selection guided by analytical models. *Procedings of Parallel Computing* 10 (2005).

[20] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM.

[21] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Transactions on Programming Languages and Systems* 37, Issue 4 (July 2015).

[22] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *International Symposium on Code Generation and Optimization (CGO 18)*. ACM, New York, NY, USA.

[23] Chung-hsing Hsu and Ulrich Kremer. 2004. A quantitative analysis of tile size selection algorithms. *The Journal of Supercomputing* 27, Issue 3 (2004).

[24] Jeffrey A. Joines and Christopher R. Houck. 1994. On the Use of Non-Stationary Penalty Functions to Solve Nonlinear Constrained Optimization Problems with GA's. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*.

[25] Peter MW Knijnenburg, Toru Kisuki, Kyle Gallivan, and Michael FP O'Boyle. 2004. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience* 16, Issues 2-3 (2004).

[26] Peter MW Knijnenburg, Toru Kisuki, and Michael FP O'Boyle. 2003. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing* 24, Issue 1 (2003).

[27] Monica D Lam, Edward E Rothberg, and Michael E Wolf. 1991. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, Vol. 19.

ACM.

[28] Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, and Thomas Panas. 2009. Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization. In *Languages and Compilers for Parallel Computing (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg.

[29] Tze Meng Low, Francisco D. Igual, Tyler M Smith, and Enrique S Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Transactions on Mathematical Software (TOMS)* 43, Issue 2 (2016).

[30] Benoit Meister and Sven Verdoolaege. 2008. Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In *Proceedings of 6th Workshop on Optimizations for DSP and Embedded Systems (ODES-6)*.

[31] Dmitry Mikushin, Nikolay Likhogrud, Eddy Z. Zhang, and Christopher Bergström. 2014. KernelGen – The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*.

[32] Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. 1998. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming* 26, Issue 6 (1998).

[33] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. 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, New York, NY, USA.

[34] J. A. Nelder and R. Mead. 1965. A Simplex Method for Function Minimization. *Comput. J.* 7, Issue 4 (Jan. 1965).

[35] Cedric Nugteren and Henk Corporaal. 2014. Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs. *ACM Transactions on Architecture and Code Optimization* 11, Issue 4 (Dec. 2014).

[36] Philip Pfaffe, Martin Tillmann, Sigmar Walter, and Walter F. Tichy. 2017. Online-Autotuning in the Presence of Algorithmic Choice. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

[37] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 13)*. ACM, New York, NY, USA.

[38] Ari Rasch, Michael Haidl, and Sergei Gorlatch. 2017. ATF: A Generic Auto-Tuning Framework. In *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)*.

[39] Gabriel Rivera and Chau-Wen Tseng. 1999. A comparison of compiler tiling algorithms. In *International Conference on Compiler Construction*. Springer.

[40] Gabe Rudy. 2010. *CUDA-CHiLL: A Programming Language Interface for GPGPU Optimizations and Code Generation.* The University of Utah.

[41] Vivek Sarkar and Nimrod Megiddo. 2000. An analytical model for loop tiling and its solution. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2000)*. IEEE.

[42] Robert Schreiber and Jack J Dongarra. 1990. *Automatic blocking of nested loops.* Technical Report.

[43] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *International Symposium on Code Generation and Optimization (CGO 17)*. IEEE, Washington, DC, USA.

[44] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC 02)*. IEEE Computer Society Press, Los Alamitos, CA, USA.

[45] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. 2009. A Scalable Auto-Tuning Framework for Compiler Optimization. In *2009 IEEE International Symposium on Parallel Distributed Processing*.

[46] Ananta Tiwari and Jeffrey K. Hollingsworth. 2011. Online Adaptive Code Generation and Tuning. In *2011 IEEE International Parallel Distributed Processing Symposium*.

[47] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[48] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, Issue 4 (2013).

[49] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT 12), Paris, France*.

[50] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule Trees. In *Fourth International Workshop on Polyhedral Compilation Techniques , Vienna, Austria (IMPACT 14)*.

[51] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48, Issue 1 (May 2007).

[52] Zheng Wang and Micheal F.P. O'Boyle. 2008. Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach. ACM, New York, NY, USA.

[53] R Clint Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society.

[54] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Comput.* 27, Issues 1–2 (Jan. 2001).

[55] Kamen Yotov, Xiaoming Li, Gang Ren, MJS Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. 2005. Is search really necessary to generate high-performance BLAS? *Proc. IEEE* 93, Issue 2 (2005).

[56] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E Eichenberger, and Kevin O'Brien. 2010. Automatic creation of tile size selection models. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM.