Dataset Sensitive Autotuning of Multi-Versioned Code based on Monotonic Properties *

Autotuning in Futhark

Philip Munksgaard[0000−0001−9499−199X], Svend Lund Breddam, Troels Henriksen[0000−0002−1195−9722], Fabian Cristian Gieseke[0000−0001−7093−5803], and Cosmin Oancea[0000−0001−5421−6876]

DIKU, University of Copenhagen, Copenhagen, Denmark
philip@munksgaard.me, svendbreddam@gmail.com, athas@sigkill.dk,
fabian.gieseke@di.ku.dk, cosmin.oancea@diku.dk

Abstract. Functional languages allow rewrite-rule systems that aggressively generate a multitude of semantically-equivalent but differently-optimized code versions. In the context of GPGPU execution, this paper addresses the important question of how to compose these code versions into a single program that (near-)optimally discriminates them across different datasets. Rather than aiming at a general autotuning framework reliant on stochastic search, we argue that in some cases, a more effective solution can be obtained by customizing the tuning strategy for the compiler transformation producing the code versions.

We present a simple and highly-composable strategy which requires that the (dynamic) program property used to discriminate between code versions conforms with a certain *monotonicity assumption*. Assuming the monotonicity assumption holds, our strategy guarantees that if an optimal solution exists it will be found. If an optimal solution doesn't exist, our strategy produces human tractable and deterministic results that provide insights into what went wrong and how it can be fixed.

We apply our tuning strategy to the incremental-flattening transformation supported by the publicly-available Futhark compiler and compare with a previous black-box tuning solution that uses the popular Open-Tuner library. We demonstrate the feasibility of our solution on a set of standard datasets of real-world applications and public benchmark suites, such as Rodinia and FinPar. We show that our approach shortens the tuning time by a factor of $6\times$ on average, and more importantly, in five out of eleven cases, it produces programs that are (as high as $10\times$) faster than the ones produced by the OpenTuner-based technique.

Keywords: Autotuning, GPGPU, Compilers, Nested Parallelism, Flattening, Performance

1   Introduction

Adapting the compilation technique to the dataset and hardware characteristics is an important research direction [8], especially in the functional context where rewrite-rule systems can, in principle, be used to aggressively generate a multitude of semantically-equivalent but differently-optimized versions of code [27].

The main target of this work is highly-parallel hardware, such as GPGPUs, which have been successfully used to accelerate a number of big-compute/data applications from various fields. Such systems are however notoriously difficult to program when the application exhibits nested parallelism—think imperfectly-nested parallel loops whose sizes are statically unknown/unpredictable.

Common parallel-programming wisdom says that, in principle, one should exploit enough levels of parallelism to fully utilize the hardware[1] and to efficiently sequentialize the parallelism in excess. However, even this simple strategy is difficult to implement when the parallel sizes vary significantly across (classes of) datasets: for example, one dataset may offer enough parallelism in the top parallel loop, while others require exploiting several levels of inner parallelism.

To make matters even more difficult, the common wisdom does not always hold: in several important cases [3, 14] it has been shown that even when the outer parallelism is large enough, exploiting inner levels of parallelism is more efficient, e.g., when the additional parallelism can be mapped to the threads of a Cuda block, and when the intermediate results fit in shared memory.[2] Finally, the best optimization strategy may not even be portable across different generations of the same type of hardware (GPU) from the same vendor [19].[3]

In essence, for many important applications, there is no silver-bullet optimization recipe producing one (statically-generated) code version resulting in optimal performance for all datasets and hardware of interest. A rich body of work has been aimed at solving this pervasive problem, for example by applying:

1. supervised offline training techniques to infer the best configuration of compiler flags that results in *best-average* performance across datasets [8, 13, 5];
2. various compile-time code-generation recipes for *stencil* applications, from which the best one is selected offline by stochastic methods and used online to compute same-shape stencils on larger arrays [24, 12, 15];
3. dynamic granularity-control analysis [2, 28] aimed at multicore execution, but which require runtime-system extensions that are infeasible on GPUs.

Such solutions (1-2) however, do not aim to cluster classes of datasets to the code version best suited for them, and thus to construct a single program that offers optimal performance for all datasets. Instead, a promising technique, dubbed

---

[1] Current GPU hardware require about a hundred thousands of concurrent threads to reach peak performance, and the number is still growing according to Moores law.

[2] In Cuda, shared memory refers to a small and fast memory that is used as a user-managed cache, and enables inter-thread communication within a block of threads.

[3] The LocVolCalib benchmark of FinPar suite [3], run on the **large** dataset, favors the common-wisdom approach on a Kepler GPU, but prefers exploiting inner levels of parallelism on a Turing GPU. Matters can only worsen across hardware vendors.
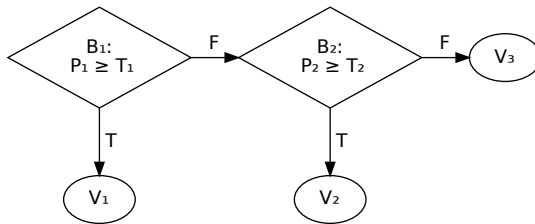
Fig. 1: The tuning tree of the paper's running examples. $V_{1...3}$ are code versions and $B_{1...2}$ are the predicates discriminating them. $P_{1...2}$ are the degree of parallelism exploited by $V_{1...2}$ and $T_{1...2}$ are the thresholds subject to autotuning.

incremental flattening [19], has been studied in the context of Futhark language [18, 11]: semantically-equivalent code versions are statically generated—by incrementally mapping increasing levels of application parallelism to hardware parallelism—and are combined into one program by guarding each of them with a predicate that compares the amount of exploited parallelism of that version with a externally defined threshold variable (integer), see fig. 1.

The amount of exploited parallelism is a dynamic program property (known at runtime), while the threshold values are found by offline tuning on a set of representative datasets. The proposed autotuner [19] uses a black-box approach that relies heavily on the stochastic heuristics of OpenTuner [4], but is impractical for application development and mainstream use, as demonstrated in section 4 on a number of standard datasets of real-world applications [14, 16] and public benchmarks from Rodinia [7] and FinPar [3] suites:

- even relatively "simple" programs, i.e., exhibiting a small number of thresholds, may result in unpredictable and suboptimal tuning times;
- the approach does not scale well, because the search space grows exponentially with the number of thresholds,[4] and thus an optimal result that perfectly discriminates between code versions may not be found, even if it exists.

1.1   Scope and Contributions of This Paper

Instead of aiming to implement a general flavor of autotuning (e.g., relying on stochastic search), this paper argues in favor of promoting a tighter compiler-autotuner codesign, by customizing the tuning technique to the code transformation producing the multi-versioned code. Our framework assumes that the multi-versioned program has the structure of a forest[5] of tuning trees, such as the one depicted in fig. 1, namely that code versions $V_{1...3}$ are placed inside branches $B_{1...2}$, whose conditions compare a dynamic program property (value) $P_{1...2}$ against *one* freshly introduced unknown/threshold variable $T_{1...2}$. Our framework finds an optimal integral value for each threshold as long as the dynamic properties (P_i) conform with a *monotonic assumption*, namely:

---

[4] A program may consist of several computational kernels, and each such kernel may produce multi-versioned code, hence the number of thresholds can grow large.

[5] Each tuning tree corresponds to a computational kernel of the original program.

If for a certain dynamic program value P_i, the corresponding code version V_i is found to be faster/slower than any/a combination of versions belonging to the subtree at the right of B_i, then it will remain faster/slower for any dynamic program value greater/less than P_i.

If the dynamic program value refers to the utilized parallelism, the trivial intuition for monotonicity is that if a code version parallelizes only the outermost loop, and has been found optimal for some loop count, then increasing the count of that parallel loop should not require exploiting the parallelism of inner loops. One can similarly reason for (combined) properties referring to load balancing, thread divergence, or locality of reference. Conversely, our technique is not suitable for tuning tile sizes, for example.

These limitations enable the design of a simple, but highly-composable and practically-effective tuning strategy. We present the intuition by using the tuning tree of fig. 1 and the simplifying assumption that the dynamic property values P_i do not change during the execution of a dataset. Then autotuning should select (exactly) one code version per (training) dataset. The rationale is as follows:

For a fixed dataset d, we can always find an instantiation of threshold values that exercises V_3 and then V_2, and we can measure their runtime. We can now combine V_2 and V_3 into an optimal subprogram on d, named V_2', by assigning T_2 *the maximal interval* that selects the fastest of the two, i.e., $[0, P_2]$ if V_2 is faster and $[P_2 + 1, \infty]$ otherwise—please notice that maximizing the interval is sound under the monotonic assumption. We continue in a recursive fashion to compute the interval of T_1 that discriminates between V_1 and subprogram V_2'.

Once we have (independently) computed partial solutions for each training dataset, we compute a global solution for each threshold by intersecting the maximal intervals across datasets. It is easy to see, by definition of intersection and maximal intervals, that (i) the resulted intervals are maximal, (ii) if nonempty, then any instantiation of the resulting intervals optimally satisfies each dataset, and (iii) conversely, if empty, then no solution exists that optimally satisfies all datasets—we use the term "near-optimal" to accommodate the empty case. Furthermore, this rationale naturally extends to the general case in which the values of P_i might vary during the execution of a dataset (see section 3.5).

In this case, the maximal interval of T_i that perfectly discriminates versions V_i and V_{i+1} is found by binary searching the set of m_i distinct values taken by P_i. This requires $O(log_2\ m_i)$ program runs, instead of $O(1)$ in the simple case.

In comparison with solutions reliant on stochastic search, our technique:

– processes each dataset independently and composably between code versions, thus requiring a predictable and small number of program runs;
– produces a guaranteed optimal solution that perfectly discriminates the training datasets if the resulting intervals are non-empty;[6]
– produces human tractable, deterministic[7] results, which, if sub-optimal, provide insight into what went wrong (empty intervals) and how it can be fixed.

---

[6] Of course, the accuracy of classifying new (test) datasets depends on whether the training datasets capture the sweet points—this is the user's responsibility.

[7] It produces the same result modulo variances in execution time.

For example, one can consider only the maximal set of datasets that produces non-empty intervals, or one can possibly instruct the compiler to generate the code versions in a different order or even redundantly, see section 3.4.

The information used by our autotuner requires minimal and trivial compiler modifications that add profiling printouts to the resulting code (details in section 3.1), hence our framework can be easily ported to other compilers employing similar multi-versioned analysis.

We demonstrate the benefits of our approach by applying it to Futhark's incremental flattening analysis and evaluating a number of (i) real-world applications [14, 16] from the remote-sensing and financial domains and (ii) benchmarks from standard suites, such as Rodinia [7] and Finpar [3, 20]. In comparison with the OpenTuner-based implementation, our method reduces the tuning time by a factor as high as $22.6\times$ and on average $6.4\times$, and in 5 out of the 11 cases it finds better thresholds that speed-up program execution by as high as $10\times$.

2 Background

This section provides a brief overview of the Cuda and Futhark features necessary to understand this paper.

2.1 Brief Overview of Cuda

Cuda [1] is a programming model for Nvidia GPGPUs. Writing a Cuda program requires the user to explicitly (de-)allocate space on the GPU device, and to copy the computation input and result between the host (CPU) and device (GPU) memory spaces. The GPU-executed code is written as a Cuda kernel and executed by all threads. The parallel iteration space is divided into a grid of blocks of threads, where the grid and block can have up to three dimensions. The threads in a block can be synchronized by means of barriers, and they can communicate by using a small amount of fast/scratchpad memory, called shared memory. The shared memory is intended as a user-managed cache, since it has much smaller latency—one-to-two orders of magnitude—than the global memory. The global memory is accessible to all threads, but in principle no synchronization is possible across threads in different blocks—other than terminating the kernel, which has full-barrier semantics.

2.2 Incremental Flattening

Futhark [18, 11] uses a conventional functional syntax. Futhark programs are written as a potentially-nested composition of second-order array combinators (SOACs) that have inherently-parallel semantics—such as map, reduce, scan, scatter, generalized histograms [17]—and loops that are always executed sequentially. Loops have the semantics of a tail recursive function, and they explicitly declare the variables that are variant throughout the execution of the loop.

```
1  let mapscan1 [m][n] (xss: [m][n]i32) : [m][n]i32 =
2      map2 (\(row: [n]i32) (i: i32) ->
3              loop (row: [n]i32) for _ in 0 ..< 64 do
4                let row' = map (+ i) row
5                in scan (+) 0 row'
6            )
7            xss (0 ..< m)
```

Fig. 2: Futhark program with size-invariant parallelism.

Fig. 2 shows the contrived but illustrative `mapscan1` function that is used as a running example in this paper. The function takes as input a $m \times n$ matrix and produces a $m \times n$ matrix as result (line 1). The function body maps each `row` of the input matrix and each row number `i` with a lambda function (line 2) that consists of a loop that iterates 64 times (line 3). The loop-variant variable `row` is initialized with the row `i` of the `xss` matrix, and the result of the loop-body expression will provide the input for the next iteration. The loop body adds `i` to each element of `row` (line 4), and computes all prefix sums of the result (line 5).

One can observe that `mapscan1` has two levels of imperfectly-nested parallelism: the outer `map` at line 2 and the inner `map-scan` composition at lines 4-5, but the Cuda model essentially supports only flat parallelism. The application parallelism is mapped to the hardware by the incremental-flattening analysis [19], which builds on Blelloch's transformation [6][8] but is applied incrementally:

V_1: a first code version is produced by utilizing only the parallelism of the outer `map` of size m, and sequentializing the inner `map-scan` composition.

V_2: a second code version (that uses $m \times n$ parallelism) is produced that maps the outer `map` parallelism on the Cuda grid, sets the Cuda block size to the size n of inner parallelism, and performs the inner `map-scan` composition in parallel by each Cuda block of threads. The intermediate arrays `row` and `row'` are maintained and reused from Cudas fast shared memory.

V_3: the flattening procedure is (recursively) applied, for example by interchanging the outer `map` inside the loop, and by distributing it across the inner `map` and `scan` expressions. The arrays will be maintained in global memory. In principle, if the nested-parallel depth is greater than 2, then the recursive application will produce many more code versions.

Unfortunately, when flattening `mapscan1`, we don't statically know what the different degrees of parallelism will be, because they depend on the input data. If the input matrix is very tall, we might prefer to use the outer parallelism and sequentialize the inner, and vice versa if the matrix is wide. Essentially, each of the three generated code versions $V_{1..3}$ might be the best one for a class of datasets. As mentioned earlier, Futhark will generate all three code versions and arrange them in a tuning tree as shown in fig. 1, where the dynamic program property refers to the degree of parallelism utilized by a certain code version, e.g., $P_1 = m$ and $P_2 = n$ (or $P_2 = m \cdot n$).

---

[8] Blelloch's flattening also work in the presence of divide-and-conquer recursion, but Futhark does not support recursive functions.

3　Autotuning Framework

3.1　Tuning Forests, Program Instrumentation

While the introduction has presented the intuition in terms of the tuning tree of fig. 1, the structure used by the tuner is essentially a tuning forest, because:

1. a program may consist of multiple computational kernels, each of them potentially generating multi-version code, and
2. the recursive step of incremental flattening may split the original computation by means of (outer)-map fission into multiple kernels, each of them potentially generating multiple code versions.

Other than the high-level structure that discriminates between code versions—i.e., the branches $B_{1..2}$—the tuning-forest representation is completely oblivious to the control flow in which various code versions are (arbitrarily) nested in. The only manner in which this control flow is (indirectly) observable by and relevant to the tuning framework is by the fact that a dynamic property P_i may take multiple values during the execution of one dataset, e.g., if a code version is executed inside a loop then its degree of parallelism may also be loop variant. Our approach requires (minimal) compiler instrumentation, added to determine:

1. the structure of the tuning forest: this is static information documenting the control dependencies between thresholds: in fig. 1, T_2 depends on T_1 because the code versions V_2 and V_3, are only executed when $P_1 \geq T_1$ fails.
2. dynamic information corresponding to the dynamic property (degree of parallelism) of each executed kernel instance, and the *overall* running time of the application. Importantly, we do not require the ability to perform fine-grained profiling of program fragments.

3.2　Autotuning Overview

The key insight of the tuning algorithm is that one can perform the tuning independently for each dataset, and that the result of tuning each threshold is a maximal interval. Furthermore, the threshold interval can be found by performing a bottom-up traversal of the tuning forest, where each step tunes one threshold (also individually). Finally, a globally-optimal solution can be found by intersecting the locally-optimal intervals across all datasets and then selecting any value in the resulting interval (as long as the training datasets are representative). This is sound and guarantees that a near-optimal solution will be found (if one exists) as long as the dynamic program property used as driver for autotuning conforms with the following monotonic assumption:

If for a certain dynamic program value P_i, a code version V_i is found to be faster/slower than any/a combination of versions belonging to the subtree at the right of B_i, then it will remain faster/slower for any dynamic program value greater/less than P_i.

The driver of the tuning algorithm is implemented by the function TUNE-PROGRAM, presented in fig. 3, which takes as arguments a program p and a

```
 1: function TUNEPROGRAM(p, ds)
 2:      ▷ p is the program being run
 3:      ▷ ds are the training datasets
 4:      ▷ t̄ = t₁, . . . , tₙ are p's thresholds in the order they appear in the tuning graph
 5:      r_{t_i} ← (0, ∞) for each program threshold t_i, 1 ≤ i ≤ n
 6:      for d in ds do
 7:          t_i ← ∞, ∀1 ≤ i ≤ n
 8:          bestRun ← run p on d with values (t₁, . . . , tₙ)
 9:          for i in n . . . 1 do
10:              ((lb_i, ub_i), bestRun) ← TUNETHRESHOLD(p, d, t̄, i, bestRun)
11:              t_i ← (lb_i + ub_i)/2
12:              r_{t_i} ← r_{t_i} ∩ (lb_i, ub_i)
13:          end for
14:      end for
15:      return (r_{t₁}, . . . , r_{tₙ})
16: end function
```

Fig. 3: Algorithm for tuning a program across a set of training datasets. For a given dataset, the near-optimal interval for each threshold is (individually) determined during a bottom-up traversal of the tuning tree (forest), where the previously determined thresholds values are used for subsequent runs. The partial results are aggregated across all datasets by taking the intersection of the corresponding intervals.

set of training datasets ds and produces a globally-optimal interval r_{t_i} for each threshold $t_i, 1 \leq i \leq n$. The outer loop starting on line 6 iterates over the available datasets. For each dataset, all thresholds are first set to infinity (line 7), forcing the bottom-most code version to run, eg. V_3 in fig. 1. Running that code version and timing it (line 8) provides a baseline for further tuning. The loop on lines 9–13 tunes each threshold in bottom-up order. After finding the optimal threshold interval for each threshold (line 10), the threshold is set to an arbitrary value in the locally optimal interval (line 11) and finally the interval is intersected with the globally optimal interval found so far (line 12).

3.3 Tuning Size-Invariant Thresholds on a Single Dataset

When tuning a single threshold, we need to distinguish between *size-variant* and *size-invariant* branches. If during the execution of the given program on *a single dataset*, we call a particular branch B_i size-invariant if, whenever it is encountered in the tuning-graph, the corresponding dynamic program value, P_i, is constant. If P_i can change during a single execution, we call the branch size-variant. As an example, it is clearly the case that `mapscan1` of fig. 2 is size-invariant, because the parallel sizes do not change during execution, hence neither does the degree of parallelism of each code version.

Because the degree of parallelism never changes, it stands to reason that for a given branch we should always perform the same choice: Either use the guarded

```
 1: function TUNETHRESHOLDINVAR(p, d, t̄, i, bestRun)
 2:     ePar ← EXPLOITEDPAR(p, d, t_i)
 3:     t_i ← 0
 4:     newRun ← run p on d with threshold values t̄
 5:     if new < best + ε then
 6:         bestRun ← min(newRun, bestRun)
 7:         lb_i ← 0, ub_i ← ePar
 8:     else
 9:         lb_i ← ePar + 1, ub_i ← ∞
10:     end if
11:     return ((lb_i, ub_i), bestRun)
12: end function
```

Fig. 4: Tuning algorithm for a size-invariant threshold. EXPLOITEDPAR(p, d, t) is the constant amount of parallelism of the code version guarded by threshold t on dataset d.

(a) Running the program on V_3 by setting all thresholds to ∞.

(b) Targetting V_2 by setting $T_2 = 50$.

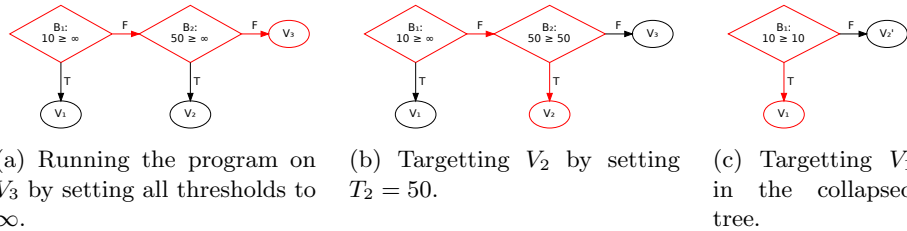(c) Targetting V_1 in the collapsed tree.

Fig. 5: Tuning the bottom-most threshold of the tuning-graph on a single dataset.

code version or progress further down the tree. Therefore, in order to find the optimal threshold value for the given input, we have to time the guarded code version, compare it to the best run time found further down the tree, and pick a threshold value that chooses the fastest of the two.

Figure 4 shows the pseudocode of a version of TUNETHRESHOLD for tuning a single size-invariant threshold on a given dataset by doing exactly that. The arguments correspond to the arguments given to TUNETHRESHOLD in fig. 3. The idea is simple: Whenever TUNETHRESHOLDINVAR is called on a threshold T_i, all the thresholds further down the tree (T_j where $j > i$) have already been tuned, and the best run time that has been encountered so far is `bestRun`. Therefore, we need to run the program once using the code version guarded by T_i (done on line 4 of fig. 4) to determine if it is faster than any of the previously tested code versions. If it is, the optimal threshold interval for T_i is the one that always chooses V_i, namely the interval from 0 to P_i (lines 6–7). Otherwise the interval from $P_i + 1$ to ∞ is optimal (line 8). As stated in the introduction, taking the maximal interval is sound under the monotonic assumption.

Figure 5 shows an example of how the size-invariant tuning works. In fig. 5a all thresholds are set to ∞, forcing V_3 to run. That allows us to find the baseline performance and get the dynamic program values $P_1 = 10$ and $P_2 = 50$. Then,

(a) Swapping B_1 and B_2.
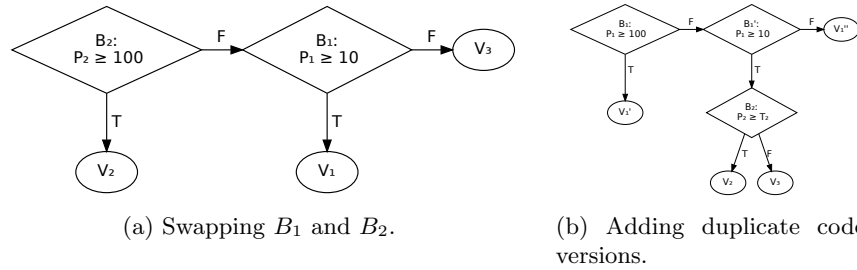
(b) Adding duplicate code versions.

Fig. 6: Alternative versions of the tuning graph, enabling different constraints.

in fig. 5b, We use the knowledge of P_2 to force V_2 to run. The change in overall run time of the program represents the difference between running code versions V_3 and V_2. After choosing an optimal threshold, we can think of the bottom part of the tree as one collapsed node, and continue our tuning by moving up the tree to run V_1, as seen in figure fig. 5c

3.4 Monotonicity Assumption

The monotonicity assumption, outlined in section 1.1, is what ultimately makes our tuning method work, and it is therefore also the primary restriction for our method. In essence, we assume that for any branch B_i, the performance of the guarded code version as a function of P_i, is monotonically increasing compared with any of the code versions further down the tree. In terms of fig. 5, if V_1 is found to outperform any of the other versions when $P_1 = 10$, then V_1 will keep outperforming the other code versions for larger values of P_1.

The implication of the monotonicity assumption is that there is at most one cross-over point for each branch. The interval found using the method described above precisely models this behavior.

This simplifying assumption relies on the compiler choosing meaningful measures to distinguish between code versions. In other words, for a given branch B_i guarding V_i, the dynamic program value P_i should be a measure of how "good" V_i is, compared to the code versions further down the tuning tree. That, in turn, puts restrictions on what P_i should measure. In the context of incremental flattening, each P_i measures the degree of parallelism of the guarded code version, and thus the monotonicity assumption should hold according to the common wisdom of optimizing the amount of parallelism.

The monotonicity assumption is closely related to the structure of the tuning forest. The tuning forest built by incremental flattening, and tuned by our technique, does not allow for more complex ways to discriminate between code versions. For instance, in fig. 5, it is not possible to specify that V_1 should be preferred when $P_1 \leq 10$ or $P_1 \geq 100$, or that V_1 should be preferred when $P_1 \geq 10$ unless $P_2 \geq 100$. However, in principle, one can still model such casses by instructing the compiler to generate the code versions in a different order, or

```
1  let mapscan2 [k] (ns: []i32) (xs: [k]i32) : [k]i32 =
2      loop xs for n in ns do
3        let m = k / n
4        let xss': [m][n]i32 = unflatten m n xs
5        let xss =
6          map2(\(row: [n]i32) (i: i32) ->
7                   loop (row: [n]i32) for _ in 0 ..< 64 do
8                     let row' = map (+i) row
9                     in scan (+) 0 row'
10               )
11               xss' (0 ..< m)
12      in flatten_to k xss
```

Fig. 7: Futhark program with size-variant parallelism.

even to duplicate some code versions in the tuning forest. For instance fig. 6a shows a reordered version of fig. 1, which enables us to model the first restriction while still conforming with the monotonicity assumption. Similarly, the second restriction can be modeled by adding duplicate code versions, as in fig. 6b, where V_1' and V_1'' are obtained from handicapping V_1 in the case when $P_1 < 10$ and $P_1 > 100$, respectively. Such transformations hint that the monotonicity restriction can be relaxed to a piece-wise monotonic one.

While our tuning technique is primarily aimed at incremental-flattening analysis, it should work in other contexts, as long as the modeled (dynamic) program property conforms with the monotonicity assumption.

3.5 Tuning Size-Variant Thresholds

In section 3.3, we assumed that the degrees of parallelism exhibited by the different branches were constant during a single execution of the program. However, that is not always the case.

For instance, the mapscan2 function shown in fig. 7 is *size-variant*. Again, we're not interested in the specific computation, but rather in the structure which serves to illustrate the difference between size-invariant and size-variant programs. The core algorithm is similar to mapscan1, but with a loop added around it. In each iteration of the outer loop, the input is transformed into a differently shaped matrix,[9] which is then mapped over. If fig. 1 is the tuning tree for this function, P_1 and P_2 would take on different values during the course of a single execution, because the degrees of outer and inner parallelism (as determined by the size of the matrix) change.

It follows that when tuning a single threshold on a single dataset, it is no longer the case that the guard predicate should always be either true or false. For instance, if a given dynamic program value P_i takes on the values 10, 50,

---

[9] The unflatten function transforms an array into a matrix of the given dimensions. flatten transforms a matrix into an array.

```
1: function TUNEVARTHRESHOLD(p, d, t̄, i, bestRun)
2:      ePar' = ePar_1, ..., ePar_m ← EXPLOITEDPAR(p, d, t_i)
3:              ePar' is a sorted sequence of unique values
4:      ePar ← 0, ePar', ∞
5:      low  ← 0,      r_low  ← run p on d with t_i set to 0
6:      high ← m + 1, r_high ← bestRun
7:      (bestRun, bestInd) ← minInd(r_low, low, r_high, high)
8:      while low < high do
9:          mid ← ⌊(low + high)/2⌋
10:         r_mid ← run p on d with t_i set to ePar_mid
11:         if r_high was faster than r_mid then
12:             low ← mid + 1
13:         else
14:             if r_low was faster than r_mid then
15:                 high ← mid − 1
16:             else
17:                 r_grd ← run p on d with t_i set to ePar_{mid+1}
18:                 if r_mid was faster than r_grd then
19:                     update bestRun, bestInd
20:                     high ← mid − 1
21:                 else
22:                     update bestRun, bestInd
23:                     low ← mid + 2
24:                 end if
25:             end if
26:         end if
27:     end while
28:     (lb_i, ub_i) ← expand bestInd to left and right
29:                 within a given runtime variance
30:     return ((lb_i, ub_i), bestRun)
31: end function
```

Fig. 8: Tuning algorithm for a size-variant threshold. $\overline{ePar'}$ is a sorted sequence of unique values denoting the amount of parallelism of the code version guarded by threshold t_i, encountered during the execution of dataset d.

and 100 during a single execution, it might be optimal to run V_i when P_i is 100, but otherwise choose the best code version further down the tree. However, according to the monotonicity assumption, there will still be a single cross-over point for size-variant thresholds, so it is still possible to find an optimal interval for a single dataset. The question is, how do we do that efficiently.

The answer relies on the insight that only the exhibited dynamic program values and ∞ are relevant to try as threshold values, as these are the only values that accurately discriminate between different distributions of code versions. In the example from above, there are four possible ways to distribute the loop iterations: Setting T_i to 10 will always choose V_i, setting T_i to 50 will choose V_i except when P_i is 10, and so on. Any other value, like 45, will not result in changes in what code versions are being run. Therefore, we only have to try those particular values.

Furthermore, the monotonicity assumption implies that there is a gradient in the direction of the optimal solution, so we can actually perform a binary search in the space of possible threshold values, by trying two neighboring threshold candidates and determining the gradient in order to reduce the search space.

Figure 8 shows an alternate version of TuneThreshold which is used to tune size-variant thresholds using this binary tuning technique. Using this function, we can tune one size-variant thresholds on a single dataset in $O(\log n)$ runs, where n is the number of different degrees of parallelism exhibited.

We conclude with a formal argument of why the use of gradient is sound under the monotonic assumption. We denote by V_i a code version that corresponds to a size-variant threshold T_i whose dynamic program property takes n distinct increasingly-sorted values $P_i^{1\cdots n}$ during the execution on a fixed dataset d. We denote by V'_{i+1} the near-optimal subprogram to the right of the branch. Assume we have run the program with $T_i \leftarrow P_i^j$ and with $T_i \leftarrow P_i^{j+1}$ and that the first run is faster. The only difference between the two runs is that the first run uses V_i for dynamic property value P_i^j while the second run uses V'_{i+1} for P_i^j; the other uses of code versions V_i and V'_{i+1} are the same between the two runs.

The first run being faster thus means that V_i is faster than V'_{i+1} for the dynamic value P_i^j, and by the monotonic assumption, it follows that it will remain faster for any value higher than P_i^j, which means that we should continue the binary search to the left of P_i^j (lines 19-20 in fig. 8). Conversely, following a similar logic, if the second run is faster, then we should continue the binary search to the right of P_i^j (lines 22-23 in fig. 8).

4   Experimental Validation

This section evaluates the tuning time of our technique as well as the performance of the tuned programs (i.e., the accuracy of tuning), by comparing with results obtained using the old OpenTuner-based black-box tuner. All benchmarks are tuned and run on a GeForce RTX 2080Ti GPU, though we have observed similar results on an older GTX780Ti.

We use a set of publicly available, non-trivial benchmarks and datasets. For each benchmark, we base our analysis on two datasets, chosen to exhibit different degrees of parallelism and to prefer different code versions. The benchmarks, datasets and the number of thresholds are shown in table 1. Heston and BFAST are real-world applications: Heston is a calibration program for the Hybrid Stochastic Local Volatility / Hull-White model [16], for which we use datasets from the `futhark-benchmarks` repository [10]. BFAST [14] is used to detect landscape changes, such as deforestation, in satellite time series data and is widely used by the remote sensing community. We use the peru and africa datasets from the `futhark-kdd19` repository [11].

---

[10] https://github.com/diku-dk/futhark-benchmarks
[11] https://github.com/diku-dk/futhark-kdd19

Benchmark	D1	D2	# Thrs.	Opent.	Our	Speedup
Heston	1062 quotes	10000 quotes	9	3798s	168s	22.59x
BFAST	peru	africa	16	1127s	206s	5.47x
LocVolCalib	medium	large	2	101s	21s	4.83x
OptionPricing	small	large	1	31s	6s	5.40x
LUD	$\mathcal{M}^{256 \times 256}$	$\mathcal{M}^{2048 \times 2048}$	9	611s	430s	1.42x
Backprop	2^{14}	2^{20}	1	30s	8s	3.65x
LavaMD	$\mathcal{M}^{10^3 \times 50}$	$\mathcal{M}^{3^3 \times 50}$	4	104s	28s	3.67x
NW	$\mathcal{M}^{2048 \times 2048}$	$\mathcal{M}^{1024 \times 1024}$	6	222s	29s	7.62x
NN	1×855280	4096×128	3	125s	36s	3.48x
SRAD	$1 \times \mathcal{M}^{502 \times 458}$	$1024 \times \mathcal{M}^{16 \times 16}$	4	148s	28s	5.31x
Pathfinder	$1 \times \mathcal{M}^{100 \times 10^5}$	$391 \times \mathcal{M}^{100 \times 256}$	1	66s	10s	6.81x

Table 1: Tuning-time speedup between the OpenTuner implementation and our autotuner on a number of benchmarks on GeForce RTX 2080 Ti. There are two datasets for each benchmark, D1 and D2, with dataset sizes given in their respective columns. The LUD benchmark is size-variant, the rest are size-invariant.

LocVolCalib (local volatilty calibration) and OptionPricing are implementations of real-world financial computations from FinPar [3, 20], for which we use datasets from the `finpar` repository [12].

LUD, Backprop, LavaMD, NW, NN, SRAD and Pathfinder are Futhark implementations of benchmarks from the Rodinia benchmark suite [7]. Some Rodinia benchmarks, like Backprop, only has one default dataset (layer length equal to 2^{16}). In those cases we've created datasets that span the Rodinia inputs— e.g., layer length 2^{14} and 2^{20} for Backprop—otherwise we have used the Rodinia datasets directly. The NW, SRAD and Pathfinder benchmarks implement batched versions of their respective algorithms, so the outer number is the number of matrix inputs ($\mathcal{M}$ denotes matrix). For instance, SRAD solves one instance of an image of size 502×458 for D1, and 1024 different images of sizes 16×16 for D2, while NN solves one nearest-neighbor problem for one query and 855280 reference points for D1, and 4096 problems each having 128 reference points.

We wish to investigate the impact of our tuning method on tuning time and run time using the tuned thresholds. Because the OpenTuner based tuner is inherently random, and benchmarking GPU programs is suspectible to run-time fluctuations, we base our analysis on three separate autotuning and benchmarking passes. For each pass, we first benchmark all programs untuned by running them 500 times with each dataset, then we tune the programs using the OpenTuner-tool and benchmark all programs using the found thresholds (500 runs), and finally we tune using our autotuner and benchmark again (500 runs). We'll pick the best tuning times for both OpenTuner and our autotuner, but it should be noted that the OpenTuner-tool has a significantly larger variance in tuning time on some benchmarks, like LUD (between 366s and 881s). To measure run time performance we first find the fastest out of the 500 runs in each

---

[12] https://github.com/HIPERFIT/finpar

pass. Then, for OpenTuner, we will show both the best and worst of those three passes, while for our autotuner we will only show the worst, because the variance is significantly smaller (and our tuning strategy is deterministic otherwise). For OpenTuner, it is also important to point out that, because it was the only tool available to tune thresholds before creating the new autotuner, it has been highly optimized, and will, among other things, use memoization techniques to minimize the number of runs, i.e., it avoids running the same combination of code versions twice.

Table 1 shows the datasets used for each benchmark, the number of tuning thresholds[13] and the average tuning times using OpenTuner and our autotuner, as well as the speedup in tuning time. Overall, we see a significant reduction in tuning time, from 1.4x for LUD to 22.6x for Heston. Without those two outliers, the average speedup is 5.1x. In general, we see that more tuning parameters result in longer tuning times, but other factors also play in, such as the time it takes to run the benchmark on a single dataset and the number of different degrees of parallelism for each particular threshold. The LUD benchmark has the least improvement in tuning time: It has size-variant parallelism, so our autotuner has to perform more test runs to find the right thresholds. We'll see that OpenTuner sometimes finds bad threshold values for LUD, so the relatively small difference in tuning time should not necessarily be seen as a boon for OpenTuner.
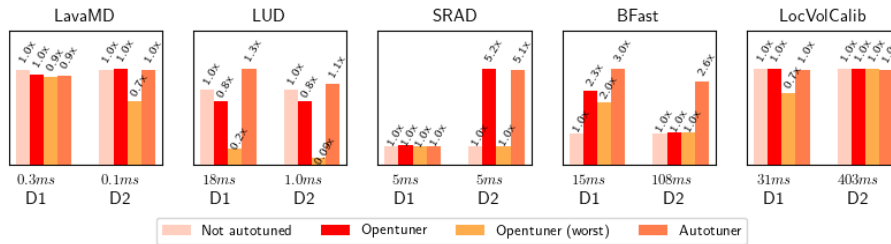

Fig. 9: Application run time speedup. The baseline is untuned performance. Higher is better.

Figure 9 shows the performance of five of the benchmarks described above: LavaMD, LUD, SRAD, BFAST and LocVolCalib. The rest of the benchmarks have similar performance characteristics when tuned using OpenTuner and our autotuner, primarily because of recent improvements in the default thresholds and heuristics used in the Futhark compiler. The benchmarks shown in fig. 9 are interesting because the different tuning methods result in programs whose performance differ significantly.

LUD is an implementation of LU matrix diagonalization with size-variant parallelism, as mentioned above. Running this program efficiently is a matter of using intra-group parallelism as long as the inner parallelism fits inside a Cuda

---

[13] The number of code-versions is equal to the number of tuning thresholds plus one.

block, which is also what the untuned version does. Our autotuner correctly finds tuning parameters that encode this behavior while OpenTuner fails to do so. In fact, it sometimes produces extremely degenerate results, due to the randomness inherent in the technique.

In the SRAD benchmark, OpenTuner will sometimes find the correct threshold values the datasets, but not always, as shown in the second dataset. A similar story can be told for LocVolCalib and LavaMD, where the OpenTuner tool sometimes find bad threshold values.

BFAST, which also relies on intra-group parallelism and is highly sensitive to tuning parameter variations, receives a significant performance boost from accurate tuning. However, the OpenTuner tool cannot even handle the largest dataset for BFAST (africa) because it causes our GPU to run out of memory, with no suitable fallback strategy, which is why we see no improvement in the second dataset at all compared to the untuned version. Our autotuner can correctly identify which threshold is causing the device to run out of memory and correctly tune to avoid it.

Interestingly, one can observe that benchmarks which have many thresholds, but not a big difference in tuning time, such as LUD and BFAST, are also the ones on which OpenTuner results in the worst program execution time. OpenTuner is not able to accurately discriminate between the different code versions, and seems to get stuck in local minimas because it terminates before the time-out is reached.

Finally, we should emphasize that, in contrast to the OpenTuner-based tool, the tuning time of our autotuner is deterministic. This means that you can reason about how many datasets you want to tune on, without having to fear tuning for unexpectedly long time. For instance, one might use the savings in tuning time to increase the set of training datasets, so as to improve the likelihood of hitting the threshold sweet spots, thus improving the prediction for new datasets.

5   Related Work

The study of autotuning solutions has been motivated by two observations: The first is that, in general, there might not exist one optimization recipe that results in best performance across all datasets, i.e., "one size does not fit all". The second is that not all performance optimizations are portable across different hardware combinations. Related work is aligned along three directions:

The first direction is to infer the best configuration of compilation flags that results in the *best average performance* across a set of training datasets on a given hardware setup. Solutions typically apply machine learning techniques, for example by relying on supervised off-line training [13], and promising results have been reported for both multi-core [8] and many-core [5] systems. For example, such techniques have successfully inferred (i) the compilation flags of the `-O3` GCC option, and improved on it when the set of programs is restricted, and (ii) near-optimal tile sizes used in GPU code generation of linear algebra operations that outperformed finely-tuned libraries, such as cuBLAS.

The second direction has been to promote a compiler design reliant on auto-tuning that separates concerns: The compiler maintains a thesaurus of legal code transformations that might improve performance, and the autotuner is responsi-ble for selecting the combination of transformations that maximize performance for *a given dataset run on some given hardware*. For example, Lift [27, 15] and SPIRAL [12], exploit the rich rewrite-rule systems of functional languages in this way. Similarly, Halide [24] applies stochastic methods to find the best fusion schedule of image-processing pipelines, corresponding to various combinations of tiling, sliding window and work replication transformations. The per-dataset tuning is feasible in cases such as stencil computations, because the important tuning parameter is the stencil's shape, and the performance is likely portable on larger arrays.

The third research direction is to provide a general black-box autotuning framework such as OpenTuner [4], which uses a repertoire of stochastic search strategies, such as hill-climbing and simulated annealing, and also provides the means for the user to define custom search strategies. ATF [25] similarly fol-lows this research direction and provides a generic framework that supports annotation-driven autotuning of programs written in any language. ATF simpli-fies the programming interface, allows the specification of constraints between tuning parameters and optimizes the process of search-space generation, but it only supports tuning a single dataset at a time. However, like OpenTuner, ATF does not use any knowledge of the program structure or of the compilation technique that is being used.

Such strategies can work well when every point in the space provides new information to guide the tuning. Unfortunately, our results indicate that the threshold parameter space of compilation schemes such as incremental flatten-ing [19] is too sparse for such black-box startegies to be effective in practice: (i) in several cases, near-optimal configurations are not (reliably) found even when enough time is given for the search to finish naturally, and (ii) typically the tuning times are too large (and unpredictable), which makes it infeasible to use it during application development stages.

The main high-level difference of our approach, compared to these other approaches, is that it integrates the multi-versioned compilation with a rela-tively cheap and one-time autotuning process that results in one executable that automatically selects the most efficient combination of code versions for *any* dataset.[14] In comparison, the first direction selects the compilation strategy that is best on average for the training datasets, and the second direction needs to repeat the autotuned compilation whenever the stencil shape changes.

At a very high level, our method has some relation to software product lines (SPLs), where techniques have been explored to, for instance, generate a multi-tude of code versions and statically determine the energy usage of each [9].

---

[14] This way of combining static and dynamic analysis by means of lightweight pred-icates is reminiscent of techniques used for automatic parallelization of sequential loops [22, 26].

Finally, another related research direction has been the study of various run-time systems aimed at dynamically optimizing program execution on the target dataset, for example by dynamically adjusting the granularity at which parallelism is exploited for multicore execution [2, 28] and by speculatively executing in parallel loops with statically unknown dependencies [10, 21, 23].

6   Conclusion

We have presented a general technique for tuning thresholds in multi-versioned programs. By taking advantage of the knowledge of the tuning-forest, we can efficiently target each code version in turn, thereby finding the (near-)optimal threshold parameters using only the necessary number of runs. For size-invariant branches, we only require a single test-run, whereas we perform a binary search across the set of unique threshold values for size-variant branches. Having tuned thresholds for each dataset individually, we combine the partial-tuning results at the end, in order to find threshold parameters that optimally distinguish between the code versions in question. We have shown substantial improvement in tuning time and tuned-execution run-time compared to the previous OpenTuner-based tuning technique. Furthermore, we remark that a significant amount of effort has been devoted to downgrade the incremental-flattening analysis by pruning at compile time the number of generated code versions, precisely because the OpenTuner-based autotuning was unreliable and slow.

In comparison with more complex stochastic-search strategies, our framework proposes a custom solution that trades off generality—each predicate introduces one unknown threshold, and thus there might not exist a set of threshold values that optimally implements a top-level strategy of combining code versions—for an efficient solution that significantly reduces the number of program runs. Finally, our strategy promotes human reasoning and understanding of results, by providing sanity-assumptions, limitations and guarantees:

– The central assumption is that the dynamic values that appear in the tuning predicates satisfy a notion of monotonic behavior, namely if version V_i is optimal for a certain P_i then it remains optimal for any dynamic value greater than P_i.
– The principal guarantee is that if a (near-)optimal set of threshold values exists then it will be found. If it does not exists then necessarily the intersection of threshold intervals across datasets is empty, and a reasonable approximation is derived by considering the maximal number of datasets that result in a non-empty intersection. Alternatively, user-defined attributes may in principle change the order in which code-versions are generated, which may enable the existence of an optimal configuration.

References

1. https://docs.nvidia.com/cuda/

2. Acar, U.A., Aksenov, V., Charguéraud, A., Rainey, M.: Provably and practically efficient granularity control. pp. 214–228. PPoPP '19 (2019). https://doi.org/10.1145/3293883.3295725

3. Andreetta, C., Bégot, V., Berthold, J., Elsman, M., Henglein, F., Henriksen, T., Nordfang, M.B., Oancea, C.E.: Finpar: A parallel financial benchmark **13**(2) (Jun 2016). https://doi.org/10.1145/2898354

4. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.M., Amarasinghe, S.: Opentuner: An extensible framework for program autotuning. In: International Conference on Parallel Architectures and Compilation Techniques (2014), http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf

5. Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A.F., Ketema, J., et al.: Pencil: a platform-neutral compute intermediate language for accelerator programming. In: 2015 PACT. pp. 138–149 (2015)

6. Blelloch, G.E., Hardwick, J.C., Sipelstein, J., Zagha, M., Chatterjee, S.: Implementation of a Portable Nested Data-Parallel Language. Journal of parallel and distributed computing **21**(1), 4–14 (1994)

7. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. pp. 44–54 (10 2009). https://doi.org/10.1109/IISWC.2009.5306797

8. Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., Wu, C.: Evaluating iterative optimization across 1000 datasets. pp. 448–459. PLDI '10. https://doi.org/10.1145/1806596.1806647

9. Couto, M., Borba, P., Cunha, J., Fernandes, J.P., Pereira, R., Saraiva, J.: Products go green: Worst-case energy consumption in software product lines. In: Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, Sevilla, Spain, September 25-29, 2017. pp. 84–93. https://doi.org/10.1145/3106195.3106214

10. Dang, F., Yu, H., Rauchwerger, L.: The R-LRPD test: speculative parallelization of partially parallel loops. In: Proceedings 16th International Parallel and Distributed Processing Symposium. pp. 20–29 (2002). https://doi.org/10.1109/IPDPS.2002.1015493

11. Elsman, M., Henriksen, T., Annenkov, D., Oancea, C.E.: Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large **2**(ICFP), 97:1–97:30 (2018)

12. Franchetti, F., Low, T.M., Doutreligne, M., Veras, R.M., Spampinato, D.G., Johnson, J.R., Meshkati, J., Hoe, J.C., Moura, J.M.F.: Spiral: Extreme performance portability. Proceedings of the IEEE **106**, 1935–1968 (2018)

13. Fursin, G., Kashnikov, Y., Memon, A.W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C.K.I., O'Boyle, M.: Milepost gcc: Machine learning enabled self-tuning compiler. International Journal of Parallel Programming pp. 296–327 (2011)

14. Gieseke, F., Rosca, S., Henriksen, T., Verbesselt, J., Oancea, C.E.: Massively-parallel change detection for satellite time series data with missing values. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE). pp. 385–396 (2020). https://doi.org/10.1109/ICDE48307.2020.00040

15. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. pp. 100–112. ACM (2018). https://doi.org/10.1145/3168824
16. Henriksen, T., Elsman, M., Oancea, C.E.: Modular acceleration: Tricky cases of functional high-performance computing. In: Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing. p. 1021. FHPC 2018 (2018). https://doi.org/10.1145/3264738.3264740
17. Henriksen, T., Hellfritzsch, S., Sadayappan, P., Oancea, C.: Compiling generalized histograms for gpu. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '20, IEEE Press (2020)
18. Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F., Oancea, C.E.: Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. pp. 556–571. PLDI 2017 (2017). https://doi.org/10.1145/3062341.3062354
19. Henriksen, T., Thorøe, F., Elsman, M., Oancea, C.: Incremental flattening for nested data parallelism. pp. 53–67. PPoPP '19. https://doi.org/10.1145/3293883.3295707
20. Oancea, C.E., Andreetta, C., Berthold, J., Frisch, A., Henglein, F.: Financial software on gpus: Between haskell and fortran. In: Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing. p. 6172. FHPC '12 (2012). https://doi.org/10.1145/2364474.2364484
21. Oancea, C.E., Mycroft, A.: Set-congruence dynamic analysis for thread-level speculation (TLS). In: Amaral, J.N. (ed.) Languages and Compilers for Parallel Computing. pp. 156–171. Springer Berlin Heidelberg (2008)
22. Oancea, C.E., Rauchwerger, L.: A hybrid approach to proving memory reference monotonicity. In: Rajopadhye, S., Mills Strout, M. (eds.) Languages and Compilers for Parallel Computing. pp. 61–75. Springer Berlin Heidelberg (2013)
23. Oancea, C.E., Selby, J.W.A., Giesbrecht, M., Watt, S.M.: Distributed Models of Thread-Level Speculation. In: Procs. of Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05). pp. 920–927 (2005)
24. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. pp. 519–530. PLDI '13, ACM (2013). https://doi.org/10.1145/2491956.2462176
25. Rasch, A., Gorlatch, S.: Atf: A generic directive-based auto-tuning framework. Concurrency and Computation: Practice and Experience **31**(5), e4423 (2019)
26. Rus, S., Hoeflinger, J., Rauchwerger, L.: Hybrid Analysis: Static & Dynamic Memory Reference Analysis. Int. Journal of Par. Prog **31(3)**, 251–283 (2003). https://doi.org/10.1023/A:1024597010150
27. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. pp. 205–217. ICFP 2015. https://doi.org/10.1145/2784731.2784754
28. Thoman, P., Jordan, H., Fahringer, T.: Compiler multiversioning for automatic task granularity control. Concurr. Comput. : Pract. Exper. **26**(14), 2367–2385 (2014). https://doi.org/10.1002/cpe.3302