Probabilistic Auto-Tuning for Architectures with Complex Constraints

Benjamin Ylvisaker
GrammaTech, Inc.*
benjaminy@grammatech.com

Scott Hauck
University of Washington
hauck@uw.edu

ABSTRACT

It is hard to optimize applications for coprocessor accelerator architectures, like FPGAs and GPUs, because application parameters must be tuned carefully to the size of the target architecture. Moreover, some combinations of parameters simply do not work, because they lead to overuse of a constrained resource. Applying auto-tuning—the use of search algorithms and empirical feedback to optimize programs—is an attractive solution, but tuning in the presence of unpredictable failures is not addressed well by existing auto-tuning methods.

This paper describes a new auto-tuning method that is based on probabilistic predictions of multiple program features (run time, memory consumption, etc.). During configuration selection, these predictions are combined to balance the preference for trying configurations that are likely to be high quality against the preference for trying configurations that are likely to satisfy all constraints. In our experiments, our new auto-tuning method performed substantially better than the simpler approach of treating all failed configurations as if they succeed with a "very low" quality. In many cases, the simpler strategy required more than twice as many trials to reach the same quality level in our experiments.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors

General Terms

Optimization, probabilistic

Keywords

Auto-tuning, accelerator architectures

1. INTRODUCTION

[0]Benjamin was at the University of Washington when he did the work described in this paper.

Tuning is the process of adapting a program to a particular target architecture or class of target architectures. Automatic and semi-automatic tuning has been a topic of interest in the high performance computing community for many years, and tuning for embedded systems and even general purpose processors has been growing in importance recently. Cohen, et al.[5], argue that one technology trend driving interest in auto-tuning is the widening gap between true peak performance and what is typically achieved by conventional compiler optimization. The central problem is that architectures are so complex that for most intents and purposes it is impossible to accurately model the effects of optimizations or configuration changes on the performance of a program.

This paper focuses on parallel coprocessor accelerators, like field programmable gate arrays (FPGAs) and general purpose graphics processing units (GPGPUS),[1] for which tuning is extremely important for achieving high performance. Accelerators have many explicitly managed resources, like distributed memories, non-uniform local networks and memory interfaces, that applications must use well to achieve good performance. Adjusting algorithm-level parameters, like loop blocking factors, is an important part of this tuning process and it is hard and tedious to tune by hand. Nevertheless, manual tuning is still common.

Figure 1 gives an intuitive sense for the complex optimization spaces that all auto-tuning methods have to contend with. The plateaus, sharp cliffs and chaotic behavior make simple search strategies like hill climbing ineffective.

Accelerators present an additional challenge for automatic tuning: they have relatively poor architectural support for graceful performance degradation. If a particular configuration of a program needs to use more local data or instruction memory than is available, the program will simply fail to compile or run properly. Thus, tuning for accelerators combines searching for high values of a quality function (e.g. performance) with satisfying a number of resource constraints. Conventional approaches to auto-tuning focus on just the quality function. Quality-only methods can be adapted by giving a default "very bad" score to configurations that violate some constraint. However, our results provide evidence that this is not an effective strategy.

This paper describes a new auto-tuning method that is designed to address the constraint satisfaction problem. The primary novel feature of our search algorithm is that it uses

[1]Our experiments are geared to FPGA-like architectures, though we believe the auto-tuning methods described in this paper are applicable to a wider class of architectures.

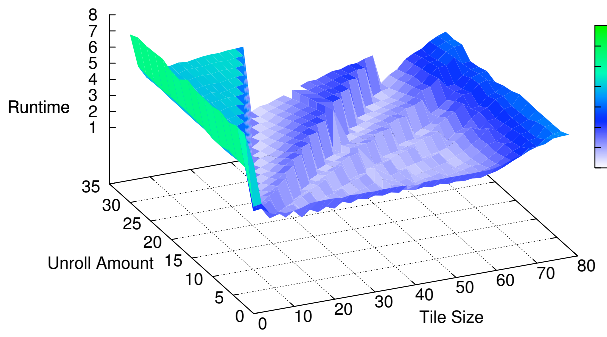

Figure 1: A tuning space for a matrix multiplication kernel. The run time function is not generally smooth, which makes tuning a challenge. In particular, there are flat "plateaus", sharp "cliffs" and multiple "troughs" with near-optimal values. (Graphic borrowed from [18].)

probabilistic predictions of several program features, then combines these predictions to calculate an overall score for untested configurations. We assume that the programmer (or some high-level program generator) has declared a number of *tuning knobs*, for which the auto-tuner will discover good values. We sometimes call tuning knobs *parameters*, and a complete set of tuning knob values for an application is a *configuration*. A number programming language-level interfaces to auto-tuning systems have been proposed recently [1, 2, 11, 16, 24, 25, 27].

Failures (constraint violations) make auto-tuning more challenging because it is no longer sufficient to optimize a single quality function. It is *possible* to define the quality of all failing configurations to be "very low". However, there are two important weaknesses to this simple approach to failures:

- If a large portion of all possible configurations fail, it can be hard for a tuning search to find any regions of the space where there are successes, because all failures are equally bad.
- The highest quality configurations are often very close to failing configurations, because it is usually best to use up all the available resources without oversubscribing them. Thus it is likely that smart auto-tuning algorithms for accelerators will spend a lot of time exploring the border between successful and failing configurations. Understanding *why* some configurations fail can help the search choose better sequences of configurations to test.

Using a probabilistic framework for tuning has some additional benefits that are not necessarily limited to accelerators. We will discuss these throughout the paper. There is a great deal of prior related work, both in tuning application-level parameters, as well as tuning compilers and architectures. We discuss the relationships between this paper and prior work in Section 9.

2. OVERVIEW OF THE TUNING KNOBS METHOD

There are two basic ingredients required to use the tuning knob system:

```
void fir1(int *I, int *O, int *C, int N, int NC)
{
  for (j = 0; j < N; j++) {
    O[j] = 0;
    for (k = 0; k < NC; k++) {
      O[j] += I[j+k] * C[k];
} } }
```

Figure 2: A simple generic FIR filter.

- A real-valued optimization formula written by the programmer, with program features as the variables and simple arithmetic like addition and multiplication.
- A set of Boolean-valued constraint formulas, some of which are written by the programmer (e.g., energy consumed less than some application-defined limit) and some of which are provided as part of the system implementation (e.g., memory usage less than capacity of the target architecture).

The tuning process involves iteratively selecting and testing configurations until some stopping criterion is met. The search algorithm has to make predictions about which untested point is most likely to both have a good value for the optimization formula and satisfy all the constraint formulas. To our knowledge, this paper describes the first application-level auto-tuning method that uses probabilities and probability distributions to represent predictions about the values of program features, the likelihood of meeting constraints, and the likelihood of having a "good" value for the optimization formula. Casting the problem in probabilistic terms is useful because we can use rich statistical math to combine many competing factors.

3. AN EXAMPLE APPLICATION

To see how tuning knob are used in the applications we experimented with, consider the finite impulse response (FIR) filter in Figure 2. This is a simple sequential implementation of the algorithm, which assigns to each output location the sum of values in a window of the input array, scaled by the values in a coefficient array.

There is abundant potential parallelism in this application. All $\texttt{N} \times \texttt{NC}$ multiplications are completely independent, and the $\texttt{N} \times \texttt{NC}$ additions are `N` independent reductions of size `NC`.

Adapting a FIR filter for high-performance execution on an accelerator requires making a number of implementation choices. The inner loop should be parallelized, but complete parallelization is unrealistic, if `NC` is large. We assume that `NC` is large enough that the coefficient array will have to be broken up into banks and distributed around the local memories of the accelerator. Different accelerators have memory structures that support different access patterns. Thus we assume that the loop is partially parallelized, controlled by a tuning knob that we will call "Banks".

For the purpose of the tuning method presented in this paper, it is not important whether these high-level structural changes are performed by a human or a domain-specific program generator. For our experiments we wrote code with explicit tuning knobs by hand. A more detailed development of the example can be found in [25].

In this small example, the optimization formula is simply the run time, which should be minimized. The constraints are all architecture-specific, and would be provided by the compiler. The constraints are described in Section 8.

4. PROBABILISTIC AUTO-TUNING

In this section, we describe the most important features of our auto-tuning method in a top-down style. There are many implementation details in the system, and many of them have a number of reasonable alternatives. We will explicitly point out what parts are essential to the functioning of the system and what parts could be replaced without fundamentally altering it.

In theory, at each step in the search process the algorithm is trying to find the untested configuration (or *candidate*) c that maximizes the following joint probability formula:

$$\begin{gathered}\mathrm{P}(c \text{ is the highest quality}^{2} \text{ configuration} \\ \cap\ c \text{ satisfies all constraints})\end{gathered}$$

Analyzing the interdependence of the quality and constraint features of a tuning space is hard given the relatively small amount of training data that auto-tuners typically have to work with. There certainly are such interdependences, but we found in our experimentation that it works reasonably well to assume that the two factors are completely independent. With this independence assumption we can factor the joint probability into the product of two simpler probabilities.

$$\begin{gathered}P(c \text{ is the highest quality configuration})\times \\ P(c \text{ satisfies all constraints})\end{gathered}$$

A successful tuning algorithm must maintain a balance between focusing the search in the areas that seem most promising versus testing a more evenly distributed set of configurations to get a good sampling of the space. We discovered a nice technical trick that helps maintain this balance: instead of predicting the probability that a candidate is the very highest quality, we predict the probability that the quality of a candidate is higher than an adjustable target quality (q_t). Selection of the target quality is addressed in Section 7.1; the quality of the best configuration tested so far is a good starting point.

$$P(\text{quality}(c) > q_t)\times P(c \text{ satisfies all constraints})$$

Next we consider how to compute the joint probability that a configuration satisfies all constraints. Ideally, the system would be able to model the correlation between different constraints and use them to predict the joint probability of satisfying all constraints. Unfortunately, the small number of configurations that auto-tuning systems generally test provide very little training data for these kinds of sophisticated statistical analyses. However, there are often strong correlations between different constraints, since most of them relate to overuse of some resource, and a knob that correlates with one kind of resource consumption often correlates with consumption of other kinds of resources. In our experiments we found that using the minimum probability of success across all constraints worked well. This is an optimistic simplification; if we assume instead that all constraints are completely independent, using the product of the individual probabilities would be appropriate.

$$\begin{gathered}P(\text{quality}(c) > q_t)\times \\ \text{Min}_{N\in\text{constraints}}\big(P(c \text{ satisfies constraint } N)\big)\end{gathered}$$

At each step in the tuning process, the system attempts to find the untested configuration that maximizes this formula. This formula is complex enough that it is not clear that there is an efficient way to solve precisely for the maximum. Instead our tuning algorithm chooses a pseudorandom set of candidates, evaluates the formula on each one, and tests the one with the highest probability.

To evaluate this formula, we need probabilistic predictions for the program features that determine quality and constraint satisfaction. We call raw features like the run time or energy consumption of the program *sensors*. Sensors can be combined with arithmetic operations to make *derived features*, like run time-energy product.

For each candidate point and each sensor, the tuning knob search algorithm produces a predicted value distribution for that sensor at the given point. We use normal (Gaussian) distributions, because as long as we assume that the features are independent, we can combine normal distributions with arithmetic operations to produce predictions for the derived features that are also normal. Derived features are discussed in more detail in Section 6.

Aside. In our experiments, we made the simplifying assumption that a particular configuration will always have the same values for its features (run time, memory use, ...) if it is compiled and tested multiple times. In other words, we assume that the system we are tuning behaves deterministically. This is clearly a major simplification, and accommodating random system variation is an interesting and important direction for future work. It is possible that probabilistic predictions, as implemented in the tuning knob search, will be a useful framework for addressing the random variation problem as well.

4.1 Hard to predict constraints

One of the trickiest issues left to deal with is deciding what the constraint formula should be for some failure modes. The easy failures are those for which some program feature can be used to predict failure or success, and it is possible to get a value for that feature for both failed and successful tests. For example, the programmer can impose the constraint that the program cannot use more than a certain amount of energy. Every test can report its energy use, and these reported values can be used to predict the energy use of untested configurations.

The harder failures are those for which the most logical feature for predicting the failure does not have a defined value at all for failing tests. For example, consider an application where some tuning knobs have an impact on dynamic memory allocation, and a non-trivial range of configurations require more memory than is available in the system. It is possible to record the amount of memory used as a program feature, but for those configurations that run out of memory it is not easy to get the value we really want, which is *how much memory would this configuration have used if it had succeeded.*

Another constraint of the nastier variety is compile time. Full compilation for FPGAs and other accelerators can take hours or even days, and it can be especially high when the resource requirements of the application are very close to

[2]For simplicity of presentation we assume that the optimization formula specifies that high values are good.

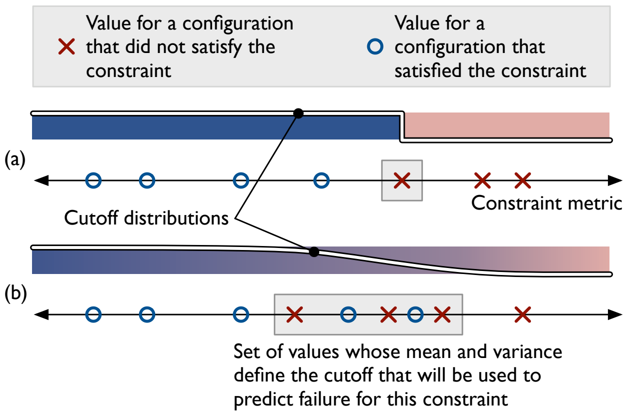
Value for a configuration that did not satisfy the constraint
Value for a configuration that satisfied the constraint
(a)
Cutoff distributions
Constraint metric
(b)
Set of values whose mean and variance define the cutoff that will be used to predict failure for this constraint

Figure 3: Two examples of setting the cutoff value for some constraint. The portion of a candidate's predicted value that is less than the cutoff determines what its predicted probability of passing this constraint will be. Values of this metric for tested configurations are represented as red x's and blue o's. Tested configurations that cannot be said to have passed or failed this constraint (because they failed for some other reason) are not represented here at all. Note that if there is some overlap in the constraint metric between cases that passed and cases that failed, the cutoff will be a distribution, not a scalar value; this is fine: the result of comparing two normal distributions with a greater-than or less-than operator can still be interpreted as a simple probability.

the limits of the architecture. For this reason it is common to impose a time limit on compilation. Like the memory usage example, failing configurations do not tell us how much of the relevant resource (compile time) would have been required to make the application work.

To compute predictions for the probability of satisfying the harder constraints, we use proxy constraints that the programmer or compiler writer believes correlate reasonably well with the "real" constraint, but for which it is possible to measure a value in both successful and failed cases. An example of a proxy constraint from our experiments is the size of the intermediate representation (IR) of a kernel as a proxy for hitting the time limit during compilation. This is not a perfect proxy in the sense that some configurations that succeed will be larger than some that cause a time limit failure. This imperfection raises the question of what the IR size limit should be for predicting a time limit failure.

To set the limit for constraint f with proxy metric p, we examine all tested points. If a configuration failed constraint f, its value for metric p is recorded as a failed value. If a configuration succeeded, or even got far enough to prove that it will not fail constraint f, its value for metric p is recorded as a successful value. Configurations that failed in some other way that does not determine whether they would have passed f or not are not recorded.

The failed and successful values are sorted, as indicated in Figure 3. The cutoff region is considered to be everything from the lowest failed value up to the lowest failed value that is higher than the highest successful value. In the special case of no overlap between successful and failed values, the cutoff region is a single value. The cutoff is then computed as the mean and variance of the values in the cutoff region. Since the system is already using normal distributions to model the predictions for all real values, it is completely natural to compare this distribution with the IR size prediction distribution to compute the probability of hitting the compiler time limit.

There are many other strategies that could be used to predict the probability of a candidate satisfying all constraints. For example, classification methods like support vector machines (SVMs[17]) or neural networks could prove effective. The classical versions of these methods produce absolute predictions instead of probabilistic predictions, but they have been extended to produce probabilistic predictions in a variety of ways. Also, the intermingling of successful and failing configurations (as opposed to a clean separation between the classes) is a challenge for some conventional classification methods.

5. PROBABILISTIC REGRESSION ANALYSIS

At the heart of our tuning knob search method is probabilistic regression analysis. Regression analysis is the prediction of values of a real-valued function at unknown points given a set of known points. Probabilistic regression analysis produces a predicted distribution, instead of a single value. Classical regression analysis is a very well-studied problem. Probabilistic regression analysis has received less attention, but there are a number of existing approaches in the applied statistics and machine learning literature. One of the most actively studied methods in recent years is Gaussian processes (GPs[15]).

Probabilistic regression analysis has been used to solve a number of problems (e.g., choosing sites for mineral extraction), but we are not aware of any auto-tuning algorithms that use it. The regression analysis needed for auto-tuning is somewhat different from the conventional approaches. Most existing approaches to probabilistic regression require a *prior distribution*, which is an assumption about the shape of the function before any training data have been observed. These assumptions are usually based on some formal or informal model of the system being measured. Auto-tuners generally have a priori no way to know the shape of the function for some program feature.

We must, however, make some assumptions about the characteristics of the underlying functions. Without any assumptions, it is impossible to make predictions; any value would be equally likely. We designed our own relatively simple probabilistic regression method based on the assumption that local linear averaging and linear projections from local slope estimates are good guides for predicting values of untested configurations. The effect of these assumptions is that our regression analysis produces more accurate predictions for features that can be modeled well by piecewise linear functions.

To keep it as clear as possible, the initial description of the complete tuning knob search method uses simplistic implementations for some subcomponents. More sophisticated alternatives are described in Section 7.

Throughout this section we use one-dimensional visualizations to illustrate the mathematical concepts. The math itself generalizes to an arbitrary number of dimensions.

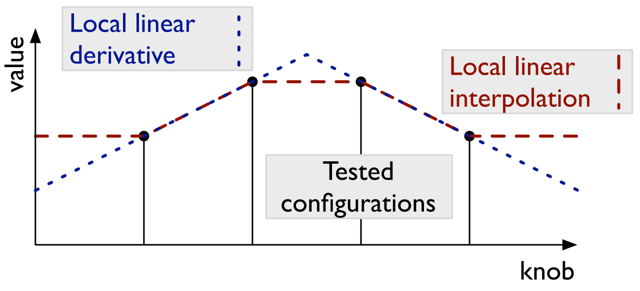

Figure 4: A comparison of local linear averaging and derivative projection. Averaging is "safe" in the sense that it never produces predictions outside the range of observed values. However, plain averaging does not follow the trends in the data, and so produces predictions that do not seem intuitively right when the tested values seem to be "pointing" in some direction.

5.1 Averaging tested values

The first step in calculating a candidate's distribution is a local linear averaging. For some candidate point $\vec{p}$, $\text{interp}_{\vec{p}}$ is the weighted average[3] of the values of the points that are *neighbors* of $\vec{p}$, where the weight for neighbor $\vec{n}$ is the inverse of the *distance* between $\vec{p}$ and $\vec{n}$. This model has two components that require definition: distance and neighbors.

5.1.1 Distance

The distance between two points is composed of individual distances between two settings on each knob, and a method for combining those individual distances. For continuous and discrete range knobs, we initially assume that the distance between two settings is just the absolute difference between their numerical values.

We combine the individual distances by summing them (*i.e.*, we use the Manhattan distance). Euclidean distance can be used as well; in our experiments, the impact of the difference between Manhattan and Euclidean distances on final search effectiveness was small.

5.1.2 Neighbors

There are many reasonable methods for deciding which points should be considered neighbors of a given point. For the initial description, we will assume that a point's neighbors in some set are the k nearest points in that set, where we choose k to be 2 times the number of tuning knobs (*i.e.* dimensions) in the application. The intuition for this k value is that if the tested configurations are evenly distributed, most points will have one neighbor in each direction. This definition of neighbors performed reasonably well in preliminary experiments, but has some weaknesses. In Section 7.2 we give a more sophisticated alternative.

5.2 Derivative-based projection

Averaging is important for predicting the value of a function, but it does not take trends in the training data into account at all, as illustrated in Figure 4. In order to take trends in the data into account, we add a derivative-based projection component to the regression analysis. In a sense, projection is actually serving two roles: (1) it helps make the predictions follow our assumption that functions are piecewise linear; (2) it helps identify the regions where there is a lot of uncertainty about the value of the function.

[3]Any weighted averaging method works (arithmetic, geometric, etc.); we used the arithmetic mean.

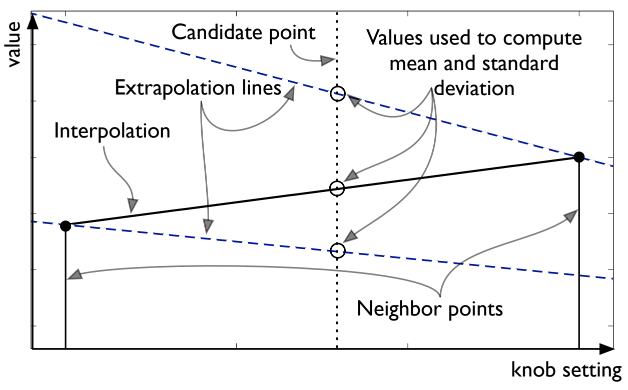

Figure 5: The basic ingredients that go into the probabilistic regression analysis used in the tuning knob search. The distribution for a given candidate configuration is the weighted mean and standard deviation of the averaged value between neighboring tested points (black line) and projected values from the slope at the neighbors (dashed blue lines).

For each candidate point $\vec{c}$ we produce a separate projection from each of the neighbors of $\vec{c}$. We do this by estimating the derivative in the direction of $\vec{c}$ at each neighbor $\vec{n}$. The derivative estimate is made by using the averaging model to estimate the value of the point ϵ distance from $\vec{n}$ in the opposite direction from $\vec{c}$.

$$\vec{d} = \vec{n} + \frac{\epsilon}{dist(\vec{c}, \vec{n})}(\vec{n} - \vec{c})$$

We use the averaging model to calculate a value for $\vec{d}$, which gives us a predicted derivative at $\vec{n}$.

$$\text{derivative at } \vec{n} \text{ towards } \vec{c} = \frac{\text{value}(\vec{n}) - \text{interp}(\vec{d})}{\epsilon}$$

Finally to get the value for $\vec{c}$ predicted from $\vec{n}$ we project the derivative back at $\vec{c}$.

$$\text{extrap}(\vec{c}, \vec{n}) = \text{value}(\vec{n}) + dist(\vec{c}, \vec{n}) \times \text{derivative}(\vec{n}, \vec{c})$$

A useful property of this projection method is that it takes into account the values of points farther from $\vec{c}$ than its immediate neighborhood; in a sense expanding the set of local points that influence the prediction.

Figure 5 illustrates averaging between two tested points and derivative projection from the same points. Three different values are generated for the candidate configuration (the dotted vertical line); the mean and variance of these values become the predicted distribution for this candidate for whatever feature we are currently working with.

5.3 Predicted distribution

The final predicted distribution for each candidate point $\vec{c}$ is the weighted mean and variance of its distance-weighted average, and projected values from all its neighbors to compute its predicted value distribution. The selection of the

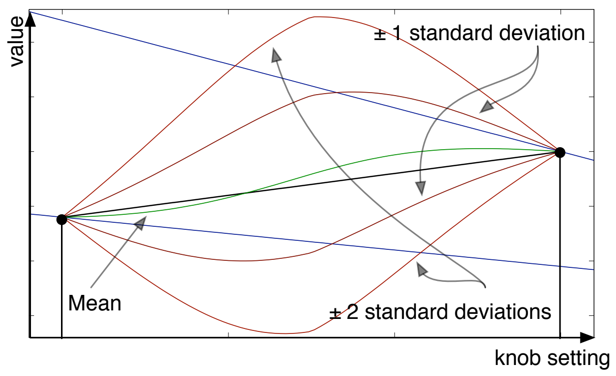

Figure 6: An illustration of the distributions that produced by the regression method presented here. Notice that the variance is much higher farther away from tested points. Also the slopes at the neighbors (blue lines) "pull" the mean line up or down. Finally, the "best" configuration to test next depends on the relative importance given to high mean quality versus large variance.

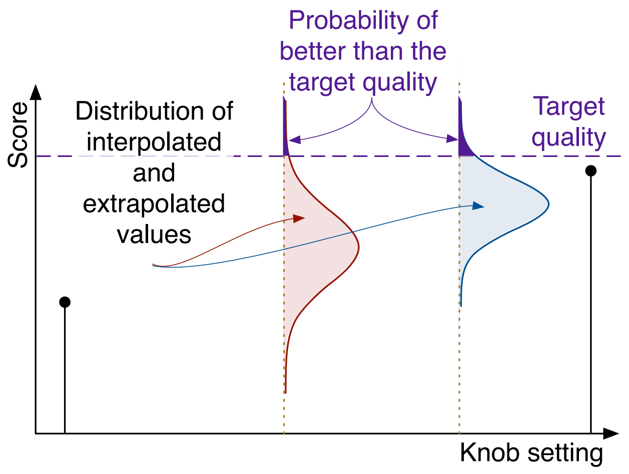
Probability of better than the target quality
Target quality
Score
Distribution of interpolated and extrapolated values
Knob setting

Figure 7: An illustration of randomly selected candidate configurations (dotted vertical lines), quality predictions for those configurations (normal distributions), the current target (dashed line), and the probability of a candidate being better than the target (dark portions of the distributions).

weights is important, and we use a "gravitational" model, where the weight on each projected value is the square of the inverse distance from the neighbor that we used to calculate that projection ($w_{\vec{n}} = \frac{1}{dist(\vec{p},\vec{n})^2}$). The weight on the averaged value is equal to the sum of the weights on all the projected values. In other words, the averaging is as important as all of the projections combined. So the complete set of weights and values used to compute the predicted distribution is:

$$\left\{\left(\left(\sum_{n\in\text{neighbors}} w_n\right), \text{interp}(\vec{c})\right)\right\} \cup \\ \left\{\left(w_n, \text{extrap}(\vec{c},\vec{n})\right) \middle| n \in \text{neighbors}\right\}$$

Observe that the variance of this set will be large when the values projected from each of the neighbors are different from each other and/or the averaged value.

Figure 6 shows what the predicted distributions would look like for the whole range of candidates between two tested points.

5.4 Target quality

Initially we assume that the target is simply the quality of the best configuration found so far; the high-water mark. Figure 7 illustrates how candidates' quality predictions are compared with the target. In this example, the highest quality tested point is not shown.

6. DERIVED FEATURES

An important part of the tuning knob search method is that predictions for sensors (features for which raw data is collected during configuration testing) can be combined in a variety of ways to make derived feature predictions. A very simple example of a derived feature is the product of program run time and energy consumption. It is possible to compute a run time-energy product value for each tested point, and then run the regression analysis directly on those values. However, by running the regression analysis on the constituent functions (run time, energy), then combining the predicted distributions mathematically we can sometimes make significantly more accurate predictions.

A slightly more complex example derived feature is an application with two sequenced loops nested within an outer loop. We can measure the run time of the whole loop nest as a single sensor, or we can measure the run time of the inner loops separately and combine them into a derived feature for the run time of the whole loop nest. If the adjustment of the tuning knobs in this application trade off run time of the two inner loops in some non-trivial way, it is more likely that we will get good predictions for the individual inner loops. The individual effects of the knobs on the inner loops are conflated together in the run time of the whole loop nest, which makes prediction harder.

An example of a derived feature that is used in our experiments is the proxy metric for compiler time limit violations. The proxy metric combines a number of measures of intermediate representation size and complexity; each measure is predicted in isolation, then the predictions are combined using derived features.

Each mathematical operator that we would like to use to build derived features needs to be defined for basic values (usually simple) and distributions of values. The simplest operators are addition and subtraction. The sum of two normal distributions (for example the predicted run times for the two inner loops in our example above) is a new normal distribution whose mean is the sum of the means of the input distributions and whose standard deviation is the sum of the input standard deviations. This definition assumes that the input distributions are independent, which is a simplifying assumption we make for all derived features.

Multiplication and division are also supported operators for derived features. Unfortunately, multiplying and dividing normal distributions does not result in distributions that are precisely normal. However, as long as the input distributions are relatively far from zero, the output distribution can be closely approximated with a normal distribution. We use

```
1  Basic tuning knob search
2      Initialize $T$, the set of results, to empty
3      Select a config. at random; test it; add results to $T$
4      while (termination criteria not met)
5          Compute quality target, $q_t$
6          Do pre-computation for regression analysis
               (e.g. build neighborhood)
7          Repeat N times (100 in our experiments)
8              Select an untested config. $c$ at random
9              Perform regression analysis at point $c$ for
                   all sensors
10             Evaluate derived features at point $c$
11             pSuccess $\leftarrow$ 1
12             $\forall f \in$ failure modes
13                 pSuccess $\leftarrow \min\big($pSuccess,
                       $P($feature linked to $f)\big)$
14             score$(c) = P($quality$(c) > q_t) \times$ pSuccess
15         Test best candidate, add results to $T$
```

Figure 8: The complete basic tuning knob search algorithm.

such an approximation, and it is currently left to the user (either the programmer or the compiler writer) to decide when this might be a problem. For details of all supported derived feature operations, see [25].

The complete basic tuning knob algorithm is shown in Figure 8.

7. ENHANCEMENTS

As described so far, the search method worked reasonably well in our preliminary experiments, but we found a number of ways to improve its overall performance and robustness. The main quantitative result in the evaluation section will show the large difference between our search method with all the refinements versus using the approach that treats all failing configurations as having a very low quality. Compared to the large difference between sophisticated and simplistic failure handling, the refinements in the following sections have a relatively small impact.

Due to space constraints, we cannot fully cover all of the topics explored in [25]. In particular, the following items we only mention briefly.

- **Constraint scaling**. We developed a method for dynamically adjusting constraint violation predication probabilities based on observed failure rates.
- **Termination criteria**. In our experiments all searches tested 50 configurations.
- **Concurrent testing**. It is relatively easy to accommodate concurrent testing in our tuning framework.
- **Boundary conditions**. The edges of the configuration space are treated specially to avoid boundary effects.
- **Distance scaling**. It is useful to scale the relative distance of different parameters according to how big of an impact they have on program features.

7.1 Target quality

Given a predicted quality distribution for several candidates, it is not immediately obvious which is the best to test next. This is true even if we ignore the issue of failures entirely. Some candidates have a higher predicted mean and smaller variance, whereas some have a larger variance and lower mean. This is a question of how much "risk" the search algorithm should take, and there is no simple best answer. The strategy we use is to compute the probability that each candidate's quality is greater than some target, which is dynamically adjusted as the search progresses.

The simplest method for choosing the target that worked well in our experiments is using the maximum quality over all the successful tested configurations. There is no reason that the target has to be exactly this high-water mark, though, and adjusting the target is a very effective way of controlling how evenly distributed the set of tested points is. The evenness of the distribution of tested points is an interesting metric because the ideal distribution of tested points is neither perfectly evenly distributed nor too narrowly focused. Distributions that are too even waste many searches in regions of the configuration space that are unlikely to contain good points. Distributions that are too uneven run a high risk of missing good points by completely ignoring entire regions of the space.

To keep the set of tested points somewhat evenly distributed the target is adjusted up (higher than the high-water mark) when the distribution is getting too even and adjusted down when the distribution is getting too uneven. Higher targets tend to favor candidates that have larger variance, which are usually points that are farther from any tested point, and testing points in "empty space" tends to even out the distribution.

There are many ways to measure the evenness of the distribution of a set of points, including [8] and [12]. See [25] for the details of our target quality adjustment heuristic.

7.2 Neighborhoods

When the distribution of a set of points is fairly uneven, the simple k-nearest and radius δ hypersphere definitions of neighbor pairs (illustrated in Figure 9) do not capture some important connections. In particular, points that are relatively far apart but do not have any points in between them might not be considered neighbors, because there are enough close points in other directions. To get a better neighbor connection graph, we use a method similar to the elliptical Gabriel graph described in [13]. Two points are considered neighbors as long as there does not exist a third point in a "region of influence" between them. For details, see [25].

8. EVALUATION

Comparing the tuning knob search against existing autotuning approaches is problematic because the main motivation for developing a new search method was handling hard-to-predict failures, and we are not aware of any other autotuning methods that address this issue. As evidence that our failure handling is effective, we compare the complete tuning knob algorithm against the same algorithm, but with the constraint/failure prediction mechanisms turned off. Points that fail are simply given the quality value of the lowest quality configuration found so far.[4]

As a basic verification that the tuning knob search algorithm selects a good sequence of configurations to test, we

[4]We also tried making the score for failing points lower than the lowest found so far. The results were not significantly different.

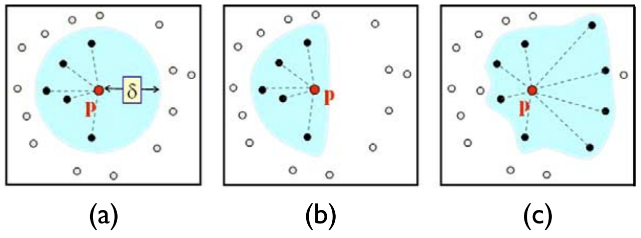

Figure 9: The mathematically simple methods for defining neighbor graphs can produce unsatisfactory results. (a) illustrates the radius δ hypersphere method; (b) illustrates the k-nearest method (with k=5). In both cases the set of neighbors is highly skewed. (c) shows a more desirable neighbor graph. Graphic borrowed from [13].

also compare against pseudorandom configuration selection. Pseudorandom searching is a common baseline in the autotuning literature. Other baselines that are used in some published results include hill-climbing style searches and simulated annealing-style searches. These kinds of algorithms could be combined with trivial failure handling, but we chose not to perform these comparisons because they would not shed light on the central issue of the importance of handling failures intelligently.

We performed our experiments in the context of a research architecture and toolflow developed at the University of Washington, called Mosaic [9, 20]. Mosaic architectures are FPGA-like, but with coarse-grained interconnect and ALUs. We generated four concrete architectures by varying two parameters: number of clusters and number of memories per cluster. All architectures had 4 ALUs per cluster. The smaller architectures had 16 clusters, or 64 ALUs, and the larger architectures had 64 clusters, or 256 ALUs. The low memory architectures had one data memory per cluster, and the high memory architectures had two. All architectures were assumed to have 16 instruction slots per ALU.[5]

8.1 The applications

To give a sense for the shapes of the tuning spaces, Figure 10 shows plots of all the performance and failure data gathered during all of our experimentation for one application (the FIR filter). Data for the other applications can be found in [25]. These plots include data from experiments with many different search methods, so the distribution of tested configurations is not meaningful. There is one plot for each architecture. The meaning of the symbols in the plots are given in the following table.

Symbol	Meaning
Black dot	Not tested
Green star	Max instruction memory failure
Empty blue square	Data memory or I/O failure
Red X	Compiler timeout failure
Filled square	Color indicates normalized quality

[5]The amount of instruction memory is small compared to conventional processors, but fairly typical of architectures in the time-multiplexed FPGA family.

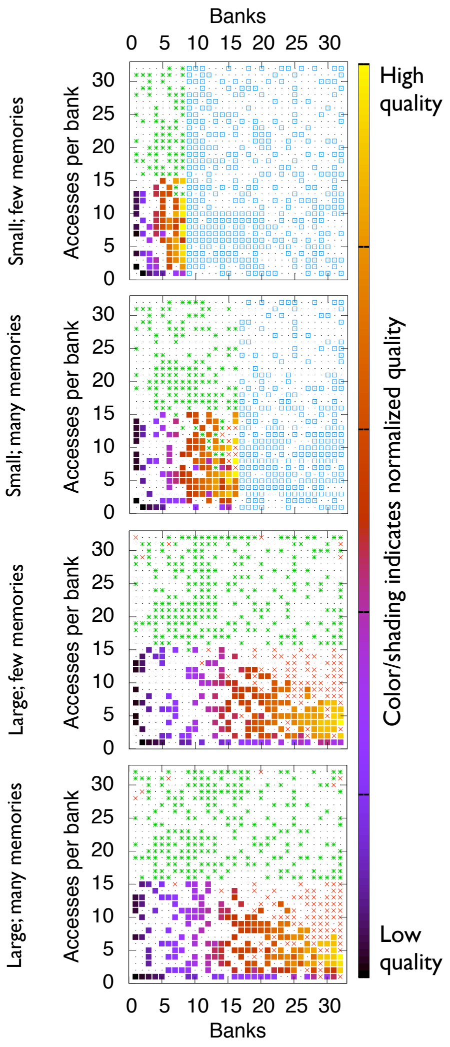

Figure 10: Normalized performance and failure modes for the FIR filter. Orange/1 is the highest performance setting; Black/0 is the lowest performance setting. Small/Large and Few/Many refer to the architecture variants that we experimented with.

The finite impulse response (FIR) filter application, which was discussed in detail earlier, has a knob that controls the number of banks into which the coefficient and input buffer arrays are broken. The more banks, the more distributed memories that can be used in parallel. The second knob controls the number of accesses to each bank per iteration. Performance should increase with increasing values of both knobs, because more parallel arithmetic operations are available.

As you can see in Figure 10, both data memory and instruction memory failures are common; more so in the smaller architectures. The number of memories clearly limits the banks knob, which results in the large regions of failures on the right side of the plots for the smaller architectures. The number of parallel accesses to an array is limited by the instruction memory of the architecture, which creates the large regions of failure toward the top of the plots. The larger architectures show an interesting effect, when both knobs are turned up relatively high the compiler begins to hit the time limit because the application is just too large to compile. This creates the jagged diagonal line of failures.

The dense matrix multiplication implementation has one level of blocking in both dimensions in the SUMMA style[19]. SUMMA-style matrix multiplication involves reading stripes of the input matrices in order to compute the results for small rectangular blocks of the output matrix, one at a time. The tuning knobs control the size of the blocking in each dimension.

The Smith-Waterman (S-W) implementation follows the common strategy of parallelizing a vertical swath of some width. One of the tuning knobs controls the width of the swath and the other controls the number of individual columns that share a single array for the table lookup that is used to compare individual letters.

The 2D convolution implementation has one knob that controls the number of pixels it attempts to compute in parallel, and another that controls the width of the row buffer (assuming that all of the rows do not fit in memory simultaneously). Of the applications and configurations that we have tested, the convolution setup had one of the most challenging shapes.

For all the applications, the highest quality configurations are directly adjacent to, and sometimes surrounded by, configurations that fail. This supports the assertion that in order to have any hope of finding the highest quality configurations a tuning method for accelerators needs an approach to failure handling that is at least somewhat sophisticated. For example, a search that had a strong preference for testing points far away from failures would clearly not perform particularly well.

8.2 Results

The experimental validation of our tuning strategy involved performing tuning searches for each of the application/architecture combinations with a particular candidate selection method. The three main methods compared were the full tuning knob search, a purely random search and the tuning knob search with the trivial approach to failures. In all cases the termination criterion was 50 tested configurations. All the search methods have some randomness, so we ran each experiment with 11 different initial seeds; the reported results are averages and ranges across all initial seeds.

Figure 11 shows the summary of the search performance results across all applications and architectures. To produce this visualization, the data for each application/architecture combination are first normalized to the highest quality achieved for that combination across all experiments. For each individual search we keep a running maximum, which represents the best configuration found so far by that particular search. Finally we take the average and 10th/90th percentile range across all application/architecture/initial seed combinations.

The headline result is that the tuning knob search method is significantly better than the other two methods. For almost the entire range of tests the 10th percentile quality for the tuning knob search is higher than the mean for either of the other two methods.

Interestingly, it seems that the purely random search does better than the tuning knob search with the trivial approach

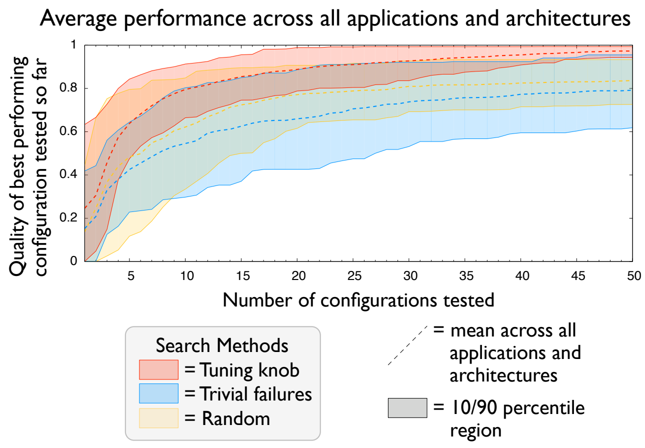

Figure 11: Given a particular number of tests, the complete tuning knob strategy clearly finds higher quality configurations on average than either the pseudorandom method or the method without failure prediction.

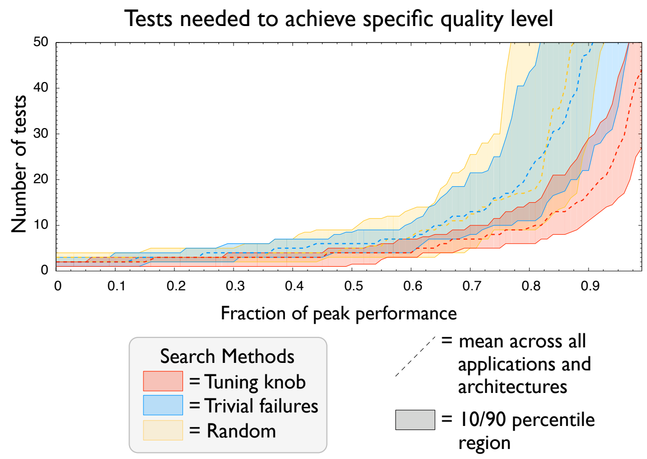

Figure 12: Number of tests required to achieve a specific fraction of peak performance, averaged across all application/architecture combinations.

to failures. Our intuition for this result is that the tuning knob search that assigns a constant low quality to all failing points does not "understand" the underlying cause for the failures and chooses to test too many points that end up failing. To reemphasize, without failures in the picture at all, some completely different search strategy might be better than the tuning knob search. However, it is very interesting that for these applications that have a large fraction of failing configurations there is a very large gap between a reasonably good search method that makes smart predictions about failure probability and one that uses essentially the same predictions, but treats failures trivially.

Figure 12 shows the same search results summarized in a different way. This figure has the axes swapped compared to Figure 11, which shows the number of tests required to achieve a specific fraction of peak performance. To produce this plot, for each individual search (application/architecture/initial seed) we calculated how many tested configurations it took to achieve a certain quality level. We then aggregated across all the applications, architectures and initial seeds for each search strategy and computed the 30th percentile, median and 70th percentile[6] at each quality level. The essential difference between these two plots is which dimension the averaging is done in.

The important result that this plot shows clearly is that for the interesting quality range, from about 50% to 90% of peak, the random and trivial failure strategies take at least twice as many tests on average to achieve the same level of quality. After 50 tests, the median search for both of the less good strategies are still reasonably far from the peak, and without running many more tests it is hard to know how many it would take to approach peak performance.

It is interesting to observe that the mean performance after 50 tests for the trivial failure strategy is below the mean performance for the random strategy (Figure 11). However, when we look at Figure 12, we see the the performance level at which the median number of tests required is 50 is higher for the trivial failure strategy than the random strategy. The reason for this apparent contradiction is the difference between mean and median. The trivial failure strategy had a relatively small number of searches that ended with a very low quality configuration, which drags down the mean more than it drags down the median.

As mentioned earlier, we have run some preliminary experiments on the contribution of individual features of the tuning knob search algorithm, like the target quality adjustment and failure probability scaling. Turning these features off has a small negative impact on search quality, but the effect is much smaller than using the trivial approach to failures. We leave a more detailed analysis and tuning of the tuning algorithm to future work.

9. RELATED WORK

There are several methods for tuning applications to particular target architectures:

- Mostly manual human effort
- Mostly automatic optimization by an aggressive transforming compiler
- Using domain knowledge to calculate parameter settings from an architecture/system description of some kind
- Empirical auto-tuning

All of these methods are appropriate under certain circumstances. The strengths of empirical auto-tuning make it a good choice for compiling high-level languages to accelerators. We will consider some of the important weaknesses of the other methods for this task.

9.1 Mostly manual human effort

It is clearly possible for a programmer to tune an application to a particular architecture by hand. Manual programmer effort has the obvious cost of requiring lots of human time, which can be quite expensive. In particular, when programs have to be retuned by a human for each architecture, it is not possible just to recompile an application when a new architecture is released. So if portability is a concern, fully manual tuning is generally not the best choice.

[6]The reason that this visualization has a 30/70 range and the Performance has 10/90 is that the data has more "spread" in one dimension than the other. Look at Figure 11 and imagine a horizontal line at any point, and notice how much wider a slice of the shaded region it is than a vertical slice.

9.2 Mostly automatic compiler optimization

Fully automatic optimization has the extremely desirable feature of not requiring any extra tuning effort from the programmer. Purely static compiler optimization faces the daunting challenge of not only adjusting the size of various loop bounds and buffers, but deciding what higher-level transformations should be applied as well. There is little to no opportunity for human guidance in this situation. The space of all possible transformations of even a simple program can be intractably large, and decades of research on auto-parallelization so far has not led to compilers that can reliably navigate this space well.

9.3 Using deterministic models

Deriving application-level parameters from architecture-level parameters with formulas like "use half of the available memory for buffer X" are sometimes a good tradeoff between human effort and application performance. However, the more complex the application and architecture in question, the more complex these relationships are. Even the interactions between two levels in a memory hierarchy can be complex enough to produce relationships that are hard to capture with formal models [26].

Another approach that fits in this category and is reasonably common in the FPGA space is writing a "generator" (usually in some scripting language) that takes a few parameters and produces an implementation tuned to the given parameters. This can be effective, but has the obvious limitation that each generator only works for a single application.

9.4 Empirical tuning

The space of empirical auto-tuners includes: (1) self-tuning libraries like ATLAS[23], PhiPAC[3], OSKI[22], FTTW[10] and SPIRAL[14]; (2) compiler-based auto-tuners that automatically extract tuning parameters from a source program; and (3) application-level auto-tuners that rely on the programmer to identify interesting parameters. Our tuning knob search fits primarily into category 3, though our search methods could certainly be used in either of the other contexts.

One pervious application of statistical methods in the auto-tuning space is Vuduc, et al.[21]. That paper used statistical models for different purposes, specifically, to decide when a search has reached the point of diminishing returns and should be terminated, and to decide which of a set of pre-computed variants of an algorithm should be selected for a particular data set.

9.5 Compiler tuning

In this work we are tuning application-specific parameters. Many of the techniques used are similar to work on tuning compiler optimizations and runtime systems either to improve the performance of a single application, or a set of applications. Some of the most closely related work of this kind is [7, 6, 4].

10. CONCLUSIONS

Adapting applications to architectures is critical for parallel coprocessor accelerators, and tuning sizing parameters to fit specific processors is an important part of the overall adaptation process. Accelerators have hard constraints (like the sizes of instruction and data memories) that are related to application-level tuning parameters in complex ways.

We proposed a probabilistic tuning method that dynamically adjusts the predicted likelihood of untested points (a) being high quality and (b) satisfying all the constraints, then combines these predictions in a smart way to choose a good sequence of configurations to test. We demonstrated that simply treating all failing configurations as "low quality", and ignoring the causes of the failures leads to inferior search performance. There is still much research to be done on auto-tuning methods, but we believe that the use of probabilistic math to combine multiple program features in a sophisticated way is a clear step forward.

11. REFERENCES

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[2] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. Technical Report MIT-CSAIL-TR-2010-032, Computer Science and ArtificialIntelligence Laboratory, MIT, July 2010.

[3] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM.

[4] J. Cavazos and M. F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 229–240, New York, NY, USA, 2006. ACM.

[5] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, September 2006.

[6] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. *SIGPLAN Not.*, 40(7):69–77, 2005.

[7] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

[8] M. Forina, S. Lanteri, and C. Casolino. Cluster analysis: significance, empty space, clustering tendency, non-uniformity. II - empty space index. *Annali di Chimica*, 93(5-6):489–498, May-June 2003.

[9] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. SPR: an architecture-adaptive CGRA mapping tool. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 191–200, New York, NY, USA, 2009. ACM.

[10] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[11] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[12] A. Jain, X. Xu, T. K. Ho, and F. Xiao. Uniformity testing using minimal spanning tree. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 4, pages 281–284 vol.4, 2002.

[13] J. C. Park, H. Shin, and B. K. Choi. Elliptic gabriel graph for finding neighbors in a point set and its application to normal vector estimation. *Comput. Aided Des.*, 38(6):619–626, 2006.

[14] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[15] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning.* Massachusetts Institute of Technology Press, 2006.

[16] C. A. Schaefer, V. Pankratius, and W. F. Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In S. B. . Heidelberg, editor, *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, volume LNCS, pages 9–20, Aug. 2009.

[17] B. Schlkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* The MIT Press, 2001.

[18] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[19] R. A. van de Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[20] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck. Static versus scheduled interconnect in Coarse-Grained Reconfigurable Arrays. In *International Conference on Field-Programmable Logic and Applications*, pages 268–275, 31 2009-Sept. 2 2009.

[21] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl.*, 18:65–94, February 2004.

[22] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics:

Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[23] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[24] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 1–8, March 2007.

[25] B. Ylvisaker. *"C-Level" Programming of Parallel Coprocessor Accelerators*. PhD thesis, University of Washington, 2010.

[26] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 141–150, New York, NY, USA, 2005. ACM Press.

[27] H. Zima, M. Hall, C. Chen, and J. Chame. Model-guided autotuning of high-productivity languages for petascale computing. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 151–166, New York, NY, USA, 2009. ACM.