Boosted Regression Trees for Predictive Auto-Tuning

First Last
Affiliation line 1
Affiliation line 2
anon@mail.com

First Last
Affiliation line 1
Affiliation line 2
anon@mail.com

First Last
Affiliation line 1
Affiliation line 2
anon@mail.com

ABSTRACT

Auto-tuning of parameteric, instrumented code has proven to be a highly effective technique for tailoring an optimal compute kernel for a particular hardware **?**. However, a drawback of this approach is that exhaustive, or even thorough, auto-tuning requires compiling many kernels and calling each one many times, and this process is slow. Furthermore, it is desirable to provide users with unified library abstraction boundaries for operations such as image filtering and matrix multiplication, even those these operations actually correspond to a large set of potential problem configurations with a wide variety of memory access patterns and computational bottlenecks. How can we draw on data from previous empirical auto-tuning of related problems on related hardware to make a just-in-time implementation decision for a novel problem? This paper presents a machine learning approach to auto-tuning, in which features of the current hardware platform, the kernel configuration and the problem instance are passed to a regression model (boosted regression trees) which predicts how much faster this kernel will be than a reference baseline. Combinatorial optimization strategies for auto-tuning that would normally require evaluating large number of kernel configurations on the real hardware, are made orders of magnitude faster by evaluating the surrogate regression model instead. We validate our approach using the filterbank correlation kernel described in Pinto and Cox [2012], where we find that 0.1 seconds of hill climbing on the regression model (which we dub "*predictive* auto-tuning") can achieve an average of 95% of the speed-up brought by minutes of empirical auto-tuning. Our approach is not specific to filterbank correlation, nor even to GPU kernel auto-tuning, and can be applied to almost any templated-code optimization problem, spanning a wide variety of problem types, kernel types, and platforms.

1. INTRODUCTION

Due to power consumption and heat dissipation concerns, scientific applications have shifted from computing platforms where performance had been primarily driven by rises in the clock frequency of a single "heavy-weight" processor (with complex out-of-order control and cache structures) to a platform with ever increasing numbers of "light-weight" cores. Interestingly, this shift is now not only relevant to computational sciences but to the development of all computer systems: from ubiquitous consumer-facing devices (e.g. phones) to high-end computer farms for web-scale applications (e.g. social networks).

Although the future lies in low-power multi-core hardware designs, the field lacks consensus on exactly how the different subsystems (memory, communication and computation) should be efficiently integrated, modeled and programmed. These systems have exhibited varying degrees of memory hierarchy and multi-threading complexity and, as a consequence, they have been increasingly relying on flexible but low-level software-controlled cache management and parallelism [Asanovic et al., 2006] in order to better control and understand the various trade-offs among performance, reliability, energy efficiency, production costs, etc. This evolution has profoundly altered the landscape of application development: programmers are now facing a wide diversity of low-level architectural features to write high-performance *and* portable code.

1.1 Motivation

In this rapidly evolving landscape, the construction of general development tools and libraries that fully utilize system resources remains a daunting task. Even within specialized architectures from the *same* vendor, like NVIDIA's Graphics Processing Units (GPUs) and the Compute Unified Device Architecture (CUDA) [Nickolls et al., 2008, NVIDIA, 2011], many developers default to massive amounts of manual labor to optimize CUDA code to specific input domains. In addition, hand-tuning rarely generalizes well to new hardware generations or different input domains, and it can also be error-prone or far from optimal. One of the reason is that kernels can produce staggeringly large optimization spaces [Datta et al., 2008]. The problem is further compounded by the fact that these spaces can be highly discontinuous [Ryoo et al., 2008], difficult to explore, and quasi-optimal solutions lie at the edge of "performance cliffs" induced by hard device-specific constraints (e.g. register file size or low-latency cache size).

1.2 Auto-Tuning

One strategy for addressing these challenges is to use one of a variety of automatic methods known collectively as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

"auto-tuning." Two major auto-tuning approaches have emerged in the extensive literature covering the subject (see surveys in [Vuduc et al., 2001, Demmel et al., 2005, Vuduc et al., 2005, Williams, 2008, Datta et al., 2008, Cavazos, 2008, Li et al., 2009, Park et al., 2011]): analytical model-driven optimization and empirical optimization [Yotov et al., 2003].

The model-driven optimization approach uses analytical abstractions to model the hardware architectures, the possible code transformations and their complex interactions. Even though highly-accurate analytical models are generally intractable to build, this approach has been quite successful in the past at accelerating serial code with simplified but general abstractions. However, large speed-ups for parallel code require more accurate high-dimensional models and since this approach is bound by the quality and scalability of its abstraction, it has been less suited for highly-specialized kernels. This approach has been dominant in the compiler community, and as a result, it has generally been applied at compile-time where important run-time characteristics such as input domains are missing. These drawbacks render the model-driven optimization approach less attractive for high-performance library developers.

The empirical optimization approach, on the contrary, seeks to find the best performing code configuration by automatically generating many versions of a parametrized kernel and benchmarking them on the actual hardware (possibly at runtime, when contextual information about the hardware and software stack is the richest). This method directly optimizes the metric(s) of interest and not surrogates. One of its main advantage is that it allows any metric to be optimized without loss of generality. It is indeed possible to formulate the problem as a multi-objective optimization and minimize both speed *and* power consumption [Rahman et al., 2011], a feat that renders the analytical model-driven approach even more difficult. Due to its flexibility, empirical auto-tuning has been successfully applied to build high-performance domain-specific libraries including dense linear algebra [Clint Whaley et al., 2001, Bilmes et al., 1997], sparse linear algebra [Vuduc et al., 2005], signal processing [**?**], sorting [Li et al., 2004], general stencil operations [Kamil et al., 2010], etc.

The empirical approach is very sensitive to the choice of instrumented optimizations *and* to the search method. The size of the search space is so large that the current best empirical auto-tuners can only consider highly-specialized functions with a limited set of code transformations and compiler options, on very few input domains [Ganapathi et al., 2009]. Although searching for good code configurations in this highly-discontinuous space can be made embarrassingly parallel, it remains a very difficult and prohibitively expensive combinatorial optimization problem as many variants of the code must generated, compiled, and benchmarked on specific input domains with meaningful statistics (that may require multiple runs). Consequently, most proposed methods prune the space with hard-coded heuristics that offer little generalization guarantees. This has been the main drawback of the empirical approach compared to the model-driven approach where good code configurations can be directly derived from the analytical model.

To alleviate this major weakness, it is intuitively appealing to combine the two approaches by first constraining the search space with an analytical model and then exploring the reduced space empirically [Chen et al., 2005, Li et al., 2009]. Unfortunately, this hybrid approach is still bound by the quality of the analytical model, which remains hard to build by hand.

In this paper, we propose to *learn the model* using nonlinear regression modelling techniques instead of constructing a model manually. By learning the model, one can hope to achieve elements of the best of both approaches: the search speed of model-based auto-tuning with the broad applicability and ease of implementation of empirical auto-tuning.

Various statistical prediction techniques have been applied with success at compile-time for general programs on various CPU architectures [Monsifrot et al., 2002, Stephenson et al., 2003, Yotov et al., 2003, Kulkarni et al., 2004, Cooper et al., 2005, Franke et al., 2005, Hutter et al., 2006, Cavazos et al., 2007, Cavazos, 2008, Hartono et al., 2009, Park et al., 2011, Fursin et al., 2008]. Relative to this work, our contribution is to show how to do fast predictive auto-tuning that satisifies the requirements to: (a) handle the variety of recent multicore architectures like GPUs [Schaa and Kaeli, 2009], (b) provide high-performance domain-specific libraries [Nukada and Matsuoka, 2009, Li et al., 2009, Kamil et al., 2010], (c) that select good implementations at run-time [Klöckner et al., 2011, Pinto and Cox, 2012], and (d) for the full input domain of a library routine [Liu et al., 2009, Grauer-Gray and Cavazos, 2011].

The paper is organized as follows: Section 2.2 describes the boosted regression tree model and the procedure for fitting it to empirical timing data. Section 4 describes the sort of kernel we employ for our benchmarking, Section **??** presents the results of our benchmarking experiments, which compare a reference implementation to a) empirical auto-tuning over a domain-specific grid, b) empirical auto-tuning over a hill-climbing search, and c) predictive autotuning. Section **??** summarizes previous and related work involving machine learning with performance auto-tuning. Section **??** summarizes our findings and outlines directions for future work.

2. PREDICTIVE AUTO-TUNING

This work shows that auto-tuning can be accelerated by orders of magnitude by using a regression model built offline as a surrogate for actual computations on the real hardware. The general form of an auto-tuning based library routine is illustrated in Figure 1 (top). An auto-tuning based routine must operate on three sets of variables:

$\mathcal{A}$**:** task description (argument shapes, physical layout)

$\mathcal{B}$**:** implementation description (auto-tuning parameters)

$\mathcal{C}$**:** platform description (capabilities, micro-benchmarks)

The hypothetical auto-tuning routine described at the top of Figure 1 might take many minutes or hours to perform the argmin at step 3 (during which time it computes the desired result many times!) so it would not be suitable for a normal library subroutine implementation. However, the form of the auto-tuning routine suggests the potential for enormous acceleration: if only there were a fast (even approximate) surrogate for the costly $MeasureTime(\cdot)$ function, then the argmin could be done in a fraction of a second and the routine could be used normally (Figure 1, bottom).

```
AUTOTUNE_EMPIRICAL(shapes, strides)
  1    $a \leftarrow TaskFeatures(shapes, strides)$
  2    $c \leftarrow PlatformFeatures()$
  3    $b^* \leftarrow \mathrm{argmin}_{b \in \mathcal{B}}\, MeasureTime(a, b, c)$ ▷ slow
  4    return $b^*$
```

```
AUTOTUNE_PREDICTIVE(shapes, strides)
  1    $a \leftarrow TaskFeatures(shapes, strides)$
  2    $c \leftarrow PlatformFeatures()$
  3    $f \leftarrow TimingModel()$
  4    $b^* \leftarrow \mathrm{argmin}_{b \in \mathcal{B}}\, f(a, b, c)$ ▷ fast
  5    return $b^*$
```

Figure 1: Pseudo-code template for empirical and predictive auto-tuning. Empirical auto-tuning (above) is inevitably slow because dynamically-generated code must be compiled and run on a number of actual-size inputs. Predictive auto-tuning (below) can be orders of magnitude faster. We show that it can also be accurate.

2.1 Learning a Regression Model

The heart of predictive auto-tuning is a regression model that acts as surrogate for a hand-written hardware model or empirical timing estimates. In our experiments this regression model fits empirically measured timing information for a *subset* of the configuration space and interpolates / extrapolates that timing across the remainder of the space. To fit this model, we form a training set $\mathcal{X}, \mathcal{Y}$ where each point $x^{(i)} \in \mathcal{X}$ is a tuple $(a^{(i)}, b^{(i)}, c^{(i)})$ and each target $y^{(i)} \in \mathcal{Y}$ reflects the speed of implementation $b^{(i)}$ on inputs $a^{(i)}$ on platform $c^{(i)}$.

The effectiveness of predictive auto-tuning depends on the mapping between the raw kernel timings $t(a, b, c)$ (i.e. in seconds) and the utility y associated with that timing. Anticipating that regression involves minimizing the squared error of our predictor (see Eq. **??**) it is important to choose y so that differences of a given numerical *magnitude* correspond to *improvements* of a certain utility. In program optimization we are interested in improving the speedup over a reference implementation $b^{(\mathrm{ref})}$, so it is natural to choose

$$y^{(i)} = \log\left(\frac{\mathrm{speed}(a, b, c)}{\mathrm{speed}(a, b^{(\mathrm{ref})}, c)}\right) = \log\left(\frac{t(a, b^{(\mathrm{ref})}, c)}{t(a, b, c)}\right) \quad (1)$$

One unique aspect of our setting compared with standard regression is that not all kernel implementations (b) are *valid* for all input configurations. One option for dealing with these invalid configurations would be to simply omit them from $\mathcal{X}$ and $\mathcal{Y}$, but that would lead to a biased regression model. Instead, we chose to associate invalid (a, b, c) tuples with a constant $y = \zeta$. It makes sense to choose $\zeta < 0$ so that invalid configurations are treated as being worse than the reference, but the question of how much worse these points should be is an empirical one. Our experiments evaluate $\log(0.01) \leq \zeta \leq \log(.99)$.

2.2 Regression Trees

A regression tree is a piece-wise constant function from one vector space to another, that works by recursively subdividing the input space into constant regions [Breiman et al., 1984, Hastie et al., 2001]. They are widely used in statistics and data-mining applications because the fitting algorithm is quick and reliable, and the form of the tree can provide insight into the relevant input variables. We use a standard fitting procedure, which takes a set of $(x, y) \in \mathbb{R}^k \times \mathbb{R}$ pairs and constructs a tree with a low mean squared error. To construct each node of a regression tree, we sort the set $\mathcal{D}$ of (x, y) pairs along each of the k features to find the best partitioning $f_{i,\gamma}$ of the input space along feature i at point γ (Eqs. 2, 3).

$$f_{i,\gamma}(x) = \begin{cases} \alpha & \text{if } x_i < \gamma \\ \beta & \text{if } x_i \geq \gamma \end{cases} \quad (2)$$

$$i^*, \gamma^* = \underset{i,\gamma}{\mathrm{argmin}}\, \hat{\mathbb{E}}\left[(y - f_{i,\gamma}(x))^2\right] \quad (3)$$

One disadvantage of the regression tree is that it does not make full use of broad patterns in the data – each partition formed by the fitting procedure is fit independently in the recursive training procedure, so it impossible for the model to extract more than one bit of information from each training partition. This disadvantage is mitigated to a significant extent by the practice of *boosting*.

2.3 Boosted Regression Trees

Boosting is an iterative procedure for constructing an *ensemble* of regression trees that is coordinated to fit training examples as accurately as possible. [Schapire, 2001, Friedman, 2002] In a recent empirical study of a range of machine learning regression problems, boosted decision trees were found to be among the best and easiest models to apply [Caruana and Niculescu-Mizil, 2006]. On each boosting iteration, a regression tree is fit to the residual error remaining after all previously-fit models have made their predictions. There are essentially three parameters that control the boosted regression tree training procedure: 1) the depth of tree constructed on each boosting iteration, 2) the minimum number of examples to allow at a regression tree leaf, and 3) the number of trees constructed by boosting. We did not attempt a systematic study of the effect of these variables on performance. We chose a maximum depth of 4, a minimum number of examples of 10, and 100 iterations of boosting.

2.4 Search Algorithms

Once an accurate regression model has been fit to the data, it remains to be optimized for novel arguments (Fig. **??**, bottom, step 4). An exhaustive search is the most reliable if it can be afforded, but in our experiments (as in general) an exhaustive search is prohibitively expensive. In our experiments we compare two strategies: (1) a generic stochastic hill-climbing search, and (2) a hand-chosen grid provided by the authors of the kernel used in our experiments [Pinto and Cox, 2012]. The *hill-climing* (HC) search algorithm starts from the reference implementation and resamples each of the parameters of the current best implementation randomly with probability 0.25 (keeping the current best setting with probability 0.75). On each hill-climbing iteration, if the speed of the newly sampled point is greater than the previous point, then it becomes the current point. We show results for search variants HC25, HC50, and HC75, which correspond do hill-climbin for 25, 50, and 75 iterations respectively. The *grid* algorithm is specific to the kernel used in our case study, the details of the grid are provided with our experimental results in Section 4.1

3. FILTERBANK CORRELATION

Filterbank correlation is a simple spatial image filtering operation that is an important subroutine in many image processing applications. It has a relatively high arithmetic intensity which makes it a natural fit for GPU platforms [Pinto and Cox, 2012].

Mathematically, we define filterbank correlation in terms of an image x and a filterbank f. The image x has R rows, C columns, and D channels (e.g. color channels) that we call its *depth*. We index x like $\mathbf{x}[i, j, d]$ where $0 \leq i < R$, $0 \leq j < C$, and $0 \leq d < D$. The filterbank f has F filters that are like little images: each has a height H, a width W, and D channels. We will restrict ourselves to what are called *valid* correlations, in which the image is larger in both rows and columns than the filters. The result of filterbank correlation of x with f is an image-like array z with $R-H+1$ rows, $C-W+1$ columns, and depth F, whose elements are defined according to Equation 4:

$$\mathbf{z}[r, c, k] = \sum_{w=0}^{W-1} \sum_{h=0}^{H-1} \sum_{d=0}^{D-1} \mathbf{x}[r+h, c+h, d]\ \mathbf{f}[k, h, w, d]. \quad (4)$$

In terms of floating point operations, a filterbank correlation requires the inner sums to be computed for each output pixel, yielding the quantity in Eq. 5:

$$\text{FLOPS} = 2FHWD(R-H+1)(C-W+1) \quad (5)$$

The multiplicative factor of 2 arises because we must first multiply an element of x with an element of f and then add the result to an element of z.

The memory transfer requirements of filterbank correlation are more difficult to quantify. Assuming three kinds of non-register memory – constant, shared, and global – and assuming optimistically that the entire filterbank fits into the GPU's constant memory, then we can establish a lower bound (Eq. 6) on the amount of memory that must be moved in order to store the computed result to global memory starting from arguments in global memory:

$$\begin{aligned}\text{Bytes} = &\ 4RCD \\ &+ 4FHWD \\ &+ 4(R-H+1)(C-W+1)F. \end{aligned} \quad (6)$$

In short, we must read the filterbank and image once, and store the result.

The arithmetic intensity of filterbank correlation, assuming our lower bound on memory transfers is therefore approximately

$$\text{intensity} \approx \frac{FDHW}{2(D+F)}, \quad (7)$$

for images that are large relative to filters. Each F output writes corresponds to approximately D input reads and F inner products between DHW elements.

The high potential for arithmetic intensity makes the GPU an ideal platform for computing filterbank correlations, and and filterbank correlation is used extensively in image and video processing, where it is often a computational bottleneck. One might expect then, that it would be easy to implement a library providing this operation as a simple function that takes pointers and strides for x, f, and z and performs the computation. However, as shown in Pinto and Cox [2012] and numerous articles on related stencil operations, it is challenging to provide an implementation or even an implementation strategy that provides satisfactory performance across the range of inputs (shapes, physical layouts) that occur in typical usage. Kamil et al. [2009] summarize a related situation related to general stencil computations in their abstract: "Although the auto-tuning strategy has been successfully applied to libraries, generalized stencil kernels are not amenable to packaging as libraries."

XXXXXXXXXXX

Our contribution of the present work is to show that boosted regression tree models, a powerful machine learning technique for function approximation, can meet the requirements of this application. We call this approach *predictive* auto-tuning to contrast it with the standard measurement-based approach which we will call *empirical* auto-tuning. We show that the kernel from Pinto and Cox [2012] can be modeled on a variety of hardware and get good performance compared to empirical auto-tuning.

4. GPU IMPLEMENTATION

The strategy we use for computing filterbank correlation on the GPU using CUDA follows Pinto and Cox [2012]. The overall strategy is to load the filterbank into constant memory, which is relatively fast and visible to all threads, and then launch a grid of blocks that tiles the output image. Each thread computes $4 \times$ *n_output_4s* channels for some column and row of z. Each block of threads computes $4 \times$ *n_output_4s* channels for a subrectangle of the output image (z). When there are more than $4 \times$ *n_output_4s* channels in z, or if the filterbank is too large to fit into constant memory, then multiple kernel executions perform the full computation. Our approach permits splitting the filterbank along the number-of-filters dimension (F) and the height dimension (H). All the filterbanks in our study are small enough that at least one row of a single filter can fit into constant memory. Pseudo-code for the kernel is given in Figure 2.

The kernel is parametrized by 10 parameters:

$$\begin{aligned}
\text{block\_h} &\in (4, 8, 16, 32, 64, 128) \\
\text{block\_w} &\in (4, 8, 16, 32, 64, 128) \\
\text{n\_filter\_r} &\in (1, 2) \\
\text{n\_output\_4s} &\in (\text{all}, 1, 2) \\
\text{spill} &\in (False, True) \\
\text{imul\_fast} &\in (False, True) \\
\text{pad\_shared} &\in (False, True) \\
\text{use\_tex1d} &\in (False, True) \\
\text{maxrreg} &\in (8, 16, 20, 24, 28, 32, \infty) \\
\text{fast\_math} &\in (False, True)
\end{aligned}$$

The block height ("block_h") and block width ("block_w") parameters control the number of threads that run within each block. Each kernel call loads some number of filter rows ("n_filter_r") into constant memory and processes the correlation of the image with just those rows, incrementing the output buffer. Each thread can compute several output elements at once, in multiples ("n_output_4s") of 4; this in-

```
THREAD_FBCORR($gX, cF, gZ$)
shared $sX \leftarrow$ all channels of region ($\beta$) of $gX$
$x, y \leftarrow$ position of this thread in output image
__syncthreads()
$v[0:N] \leftarrow 0$, for $N = 4 \times n\_output\_4s$
for $d \leftarrow 0$ to $D$,
  for $h \leftarrow 0$ to $H/n\_filter\_r$,
    for $w \leftarrow 0$ to $W$,
      $u \leftarrow sX[x+h, y+w, d]$
      for $n \leftarrow 0$ to $n\_output\_4s - 1$,
        $v[n] \leftarrow v[n] + cF[n, h, w, d]$
for $n \leftarrow 0$ to $n\_output\_4s - 1$,
  gZ[x][y][4n:4n+n] += v[4n:4n+n], (float4)
```

Figure 2: Kernel pseudo-code for filterbank correlation. Input gX is a pointer to x in global memory, input cF is a pointer to f in either constant or texture memory, and output gZ is a pointer to z in global memory. Each block of threads modifies $4 \times n\_output\_4s$ channels of a rectangle (called β in code listing) within z. A grid of blocks covers all rows and columns of z. Multiple calls can be used to apply all filters of a large filterbank f to x.

creases the efficiency of each thread, but can lead to lower occupancy. Registers are a precious commodity on the GPU, and this kernel accumulates elements of v in registers. The "spill" parameter controls whether the current thread's output position in gZ is stored in a register (faster access) or in shared memory (frees up a register). The "imul_fast" parameter controls whether integer multiplication is done in 24-bit (True) or 32-bit (False) precision. The "pad_shared" parameter controls whether the sX shared memory buffer is padded, which wastes space in shared memory but reduces bank conflicts. The "use_tex1d" parameter controls whether the image is loaded into shared memory with global pointer dereferences or texture fetches. The "maxrreg" and "fast_math" parameters are passed to the nvcc compiler to limit the number of registers available to each thread, and to enable XXX, respectively.

When the entire filterbank does not fit into the GPU's constant memory, P passes are necessary to compute all of z, where

$$P = \frac{FH}{4 \cdot n\_output\_4s \cdot n\_filter\_r}.$$

In such cases, the number of bytes moved to and from global memory is much higher than the theoretical lower limit.

$$\begin{aligned}\text{Bytes} = {} & 4RCDP \\ & + 4FHWD \\ & + 8(R - H + 1)(C - W + 1)FP.\end{aligned}$$

These passes make the I/O requirements increase quadratically in F and H. At the same time, the total number of floating-point operations (Eq. 5) is quadratic in H and W. In our experiments, we only considered square filters so in our setting the total number of flops is proportional to H^4.

Critically: what makes this kernel interesting as a case study is that the arithmetic intensity, shared storage, and register requirements of this kernel change significantly and in a complicated platform-dependent way with the argument parameters (R, C, D, F, H, W) and with the implementation parameters, especially "block_w", "block_h", "n_output_4s" and "n_filter_rows."

4.1 Reference and Grid

Pinto and Cox [2012] recommends as a reference implementation: block_w 8, block_h = 8, n_filter_rows 1, n_output_4s all, spill $False$, imul_fast $True$, pad_shared $True$, text1dfetch $True$, maxrregcount ∞, and fast_math $False$. This reference implementation was chosen manually based on good performance across a range of platforms from older-generation cards such as the 8600GT all the way to current-generation flagship cards such as the GTX 580 and C2070. Given that parameters were hand-chosen for the reference kernel, no claims are made as to the optimality nor universality of this reference (indeed, different programmers would undoubtedly arrive at different results). We use this kernel configuration as a reasonable indicator of typical performance made possible by ad hoc experimentation with parameters.

Additionally, Pinto and Cox [2012] advocate a particular grid search over what was estimated to be the most relevant part of the configuration space. This grid iterates over all combinations of n_filter_rows, n_output_4s, spill_l, pad_shared_l for three different block_h, block_w choices: (16, 8), (16, 16), and (32, 8). In our experiments, we call this algorithm the *grid* search procedure. The grid included 72 points in addition to the reference implementation, for a total of 73 points.

4.2 Software Stack

This kernel was implemented in the meta-programming style advocated in Pinto and Cox [2012] in Python using Cheetah for string processing [?] and PyCUDA [Klöckner et al., 2009] for dynamic kernel compilation and interfacing with CUDA.

5. EXPERIMENT SETUP

Recall from the introduction (Eq. ??) that auto-tuning can be seen as a conditional optimization problem in whic we seek an implementation ($b \in \mathcal{B}$) that minimizes runtime or some other scalar-valued cost function for given arguments ($a \in \mathcal{A}$) on a particular platform ($c \in \mathcal{C}$). In order to perform predictive auto-tuning with a regression model, it is necessary to characterize these three types of variables with *features*. We describe the arguments to a filterbank correlation with the 6-tuple (R, C, D, F, H, W). We randomly sampled arguments (uniformly) from the following product space:

$$\begin{aligned} R = C &\in \{256, 512, 1024, 2048, 4096\} \\ H = W &\in \{3, 5, 7, 9, 11\} \\ D &\in \{1, 4, 8, 16, 32, 64, 128, 256\} \\ F &\in \{1, 4, 8, 16, 32, 64, 128, 256\} \end{aligned}$$

A library implementation of this operation would ideally support all image and filter sizes as well as variations due to strided memory layouts. In such a setting it would be useful to characterize the arguments with features such as whether the inputs are Fortran-style contiguous, C-style contiguous, or row-padded to various byte alignments. These additional options would make our approach of automatic auto-tuning even more important, because there would be a greater variety in the kinds of computations and memory transfers to perform. Our experiments consider a somewhat simplified setting in which the arguments are always stored with depth channels being contiguous in memory, followed by columns, then rows, and then filters having the largest stride.

The product space in our study includes 1600 argument combinations, but we restricted our experiments to correlations that represented between 1 and 50 gigaflops of arithmetic. Smaller problems do not fully utilize GPU hardware and are handled equally well by many kernel settings. Larger take so long to evaluate that there is negligible inefficiency in implementing them via multiple calls with smaller images and fewer filters. With the experiments searched an argument space included 602 configurations with between 1 and 50 gigaflops.

For the implementation features b, we directly used the integer and binary values (block_w, block_h, etc.) that parametrized the kernels. We did not use platform features (c) in our experiments. We leave the investigation of cross-platform predictive auto-tuning for future work.

6. RESULTS

Figure 3 shows the effectiveness of empirical auto-tuning is in this setting. Taking the GTX 295 as an example, and averaging across the range of problem configurations in our study, we find that empirically auto-tuned implementations are on average about 50% faster than the reference implementation. The reference in turn, is about 50% faster than implementations that were empirically auto-tuned for a [randomly chosen] different argument configuration. This shows that it is genereally not enough to auto-tune for particular argument configurations; instead it is important to choose the right kernel for the job for each unique argument configuration (input-dependent auto-tuning).

Comparing the *grid* to *HC25*, *HC50* and *HC75* we found very little difference in performance. The HC25 was slightly poorer, but the grid, HC50, and HC75 algorithms delivered similar average results. None of the algorithms was strictly better than the others. In our predictive auto-tuning experiments we used HC75.

Figure **??** shows how accurate predictive auto-tuning is compared with empirical auto-tuning. The training set $(\mathcal{X}, \mathcal{Y})$ for the regression model comprised all of the $< (a, b, c), y >$ pairs observed during grid search and hill-climbing search. So for each training argument configuration a, c there were 148 different values of b and thus 148 training points. Figure **??** shows that as more argument configurations are used for training, the performance of predictive auto-tuning on test configurations (a', c) quickly approaches the average performance of empirical auto-tuning. We took care to partition the train and test sets so that there were no overlapping configurations. The key difference between the predictive and empirical auto-tuning, is that predictive auto-tuning typically took about 0.1 seconds per test example, whereas empirical auto-tuning took about 1-3 minutes.

In some cases, predictive auto-tuning yields an invalid implementation. We dealt with this scenario by backtracking through the various best-estimates discovered during the hill-climbing search. Some invalid kernels can only be discovered after compiling code and attempting to run the compiled code, so in these cases predictive auto-tuning took up to 3 seconds. Even in these cases the predictive model is much faster than empirical auto-tuning, because the first kernel that actually runs is a fast one.

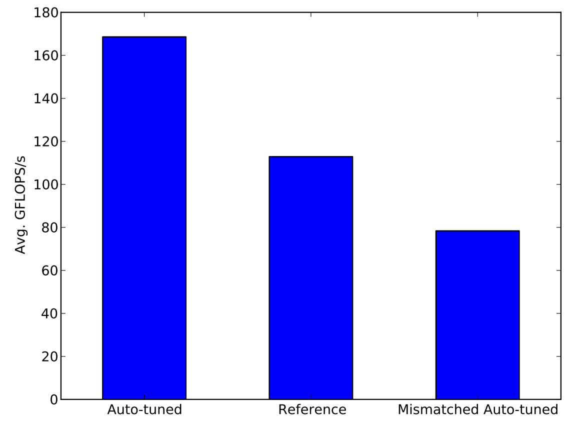


Figure 3: Different arguments call for different kernels: *left* is the average argument-specific empirical auto-tuned performance across 100 random argument configurations, *middle* is the average speed of our reference implementation, *right* is the average speed of kernels auto-tuned for different argument configurations than the one being tested. For 38/100 random confurations, the kernel auto-tuned for another problem could not even run on the GTX 295 hardware. These points contributed a speed of 0, bringing down the average much lower than the reference. Good performance across a variety of inputs requires *input-dependent auto-tuning*.

Figure **??** also shows that random hill-climbing is at least competitive and often better than the hand-chosen grid approach of Pinto and Cox [2012]. Figure **??** shows how the number of hill-climbing iterations affects the quality of the auto-tuned model. The quicker HC25 approach is clearly inferior but the quality of the 75 iteration search (HC75) is statistically similar to the shorter HC50, suggesting that XXX. Still, HC75 is not always better than the grid, so evidently neither search algorithm is perfect.

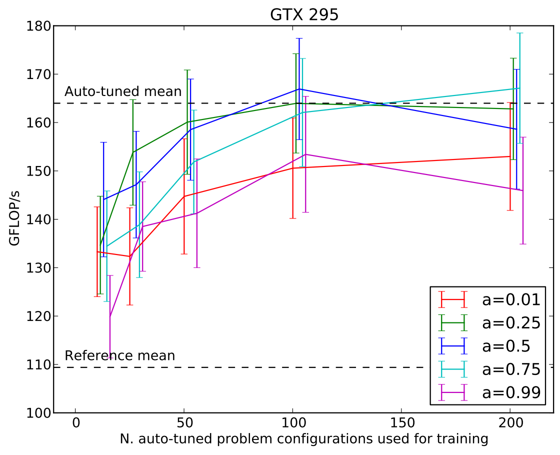
The previous figures establish that auto-tuning on a problem-by-problem basis can achieve good performance improvements over a high-quality reference baseline, but they do not show how long it takes to find these implementations. A search of 75 iterations took an average of about 120 seconds XXX on a fast computer, because it requires compiling 75 CUDA kernels, and evaluating up to 750 filterbank correlations to get reliable timing information for each candidate implementation.

Our contrib

TL;DR: Hill-climbing is a good way to search the implementation space, can search the full space and do about as well as grid search in the more restricted space.

7. DISCUSSION

In this paper, we've demonstrated a boosted regression



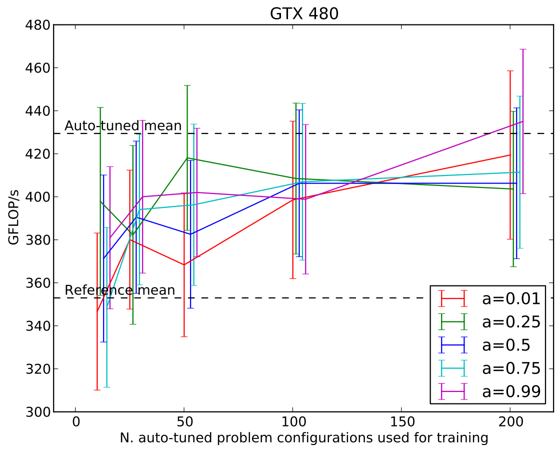
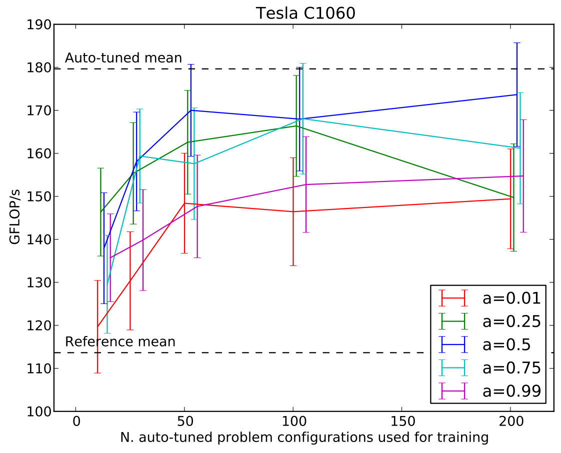


Figure 4: The effect of the invalid configuration score (a) and training set size on simulated auto-tuning. All candidates timed during the grid and hill-climbing search procedures were used as training points, so the training set sizes ranged from an average of 1500 (10 problem configurations) to 30 thousand (200 problem configurations). Training on 10 or 25 configurations was useful (higher mean than the reference), but not as useful as training on 50 or more configurations. The results on the GTX 295 suggest that moderate values of a between 0.25 and 0.75 might be best, but a had no significant effect on the GTX 480.



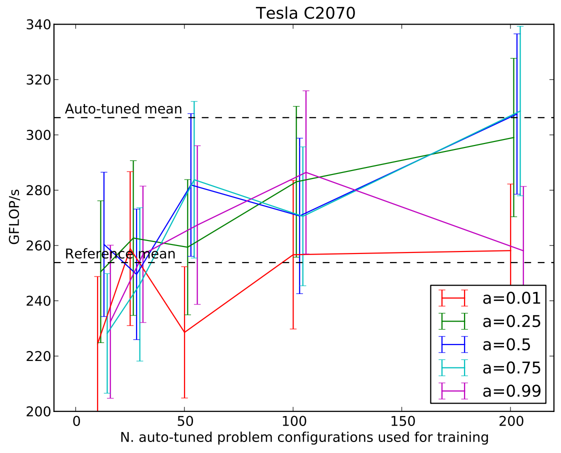
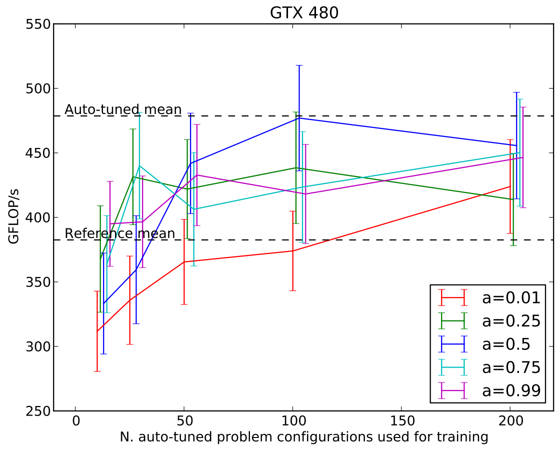


Figure 5: more timing on munctional0



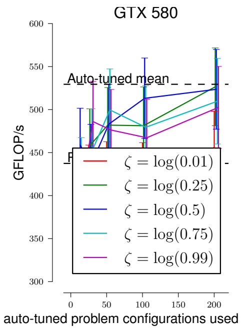
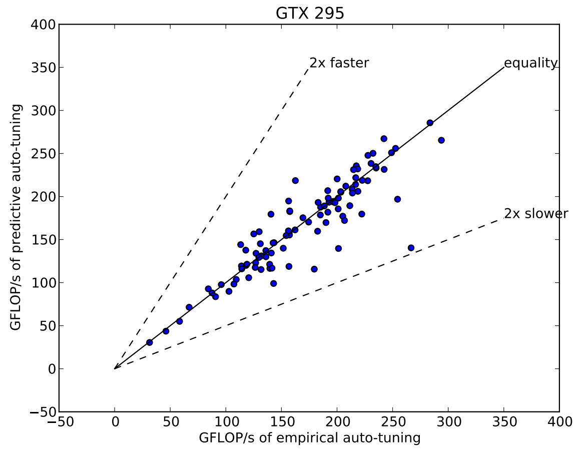


Figure 6: more timing on munctional0

tree-based auto-tuning method, wherein empirical performance data is used to train a machine learning model of performance for an instrumented GPU kernel. In contrast to traditional model-based auto-tuning, where an explicit model of performance is built on the basis of an understanding of hardware inner workings, and empirical auto-tuning, where an exhaustive set of implementation configurations are tried, the present approach generates, from scratch, a model of kernel performance on the basis of timing data from a user-definable number of kernel evaluations. This approach allows significant flexibility to navigate tradeoffs between offline and run-time costs, and final auto-tuning performance. Importantly, this method treats kernels as black-boxes, allowing the user to auto-tune in the absence of deep knowledge of hardware details (which may even been unknowable, in the case of hardware not available at the time of kernel creation). This approach also frees auto-tuning performance from a strong dependency on the accuracy (or inaccuracy) of a pre-defined analytical model.

An important use case for the tools described here is in the development of user-facing numerical libraries. Such libraries are a critical component of scientific computing infrastructure, since they abstract away implementation details and make algorithms available to a much wider audience. However, the abstraction provided by libraries represents a double-edged sword: one hand, *using* the library is easier, because it presents a unified abstraction of related functionality. However, at the same time, any given library routine might represent a wide range of substantially different problem configurations, each with distinct computa-



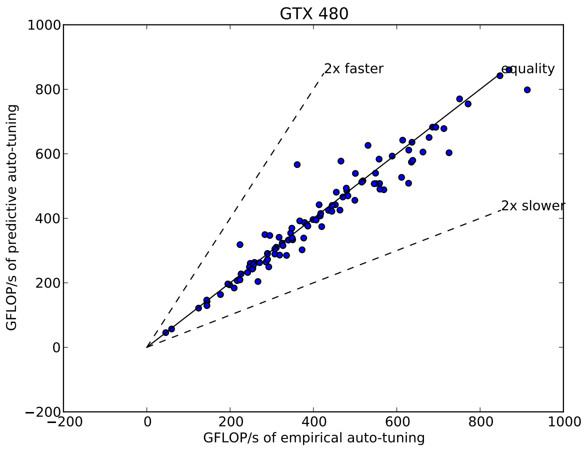


Figure 7: Computation speed for novel problem configurations using predictive vs. empirical code specialization. The scatterplots are roughly symmetric about the axis of equality (main diagonal) with occasional outliers, indicating that for most problem configurations the predictive approach gives as good a solution as the empirical approach. The predictive approach typically requires about 0.1 seconds to suggest an implementation, whereas the empirical approach requires about 2 minutes.

tional issues and bottlenecks. Auto-tuning has long provided a solution that finesses these two issues, providing multiple implementations under the hood, for multiple problem settings, and then using heuristics or explicit, handcrafted models to select the appropriate implementation for a given set of inputs. The development of such auto-tuned libraries, while extremely successful, is also very difficult. The machine-learning-based techniques described here provide a middle ground, where a library implementor can simply create an instrumented kernel, and allow generic tools to automatically generate appropriate auto-tuned implementations, with small (and controllable) run-time costs.

8. FUTURE WORK

While the present work serves as a basic demonstration of the value of using machine learning models to predict optimal implementation parameters, there are many avenues for taking these ideas further. A natural route to extend of our current approach is to include more features as inputs to the predictive model. In addition to further instrumentation of the kernel in question, input features could include a much broader range of hardware-related information, from more detailed information about device capabilities to micro-benchmarks [Wong et al., 2010], and results from performance limiter analyses [Micikevicius, 2010]. Such additional features would increase the predictive model's ability to adapt to a wide range of different kinds of hardware, including new devices not available at the time of kernel creation.

Another potentially interesting avenue of research is in interpreting the model learned by predictive auto-tuning. While traditional model-based auto-tuning approaches by design assume a given model, and empirical auto-tuning approaches are completely model free, the predictive approach described here *generates* a model from performance data. Because this generated model can be interrogated by a variety of means, a significant opportunity exists to learn about the factors that drive the performance of a given kernel. These insights can be used to further guide the development and instrumentation of the kernel, potentially yielding even greater gains.

9. CONCLUSION

Our

References

K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, et al. The landscape of parallel computing research: A view from berkeley. *EECS Department University of California Berkeley Tech Rep UCBEECS2006183*, 18(UCB/EECS-2006-183), 2006.

J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM, 1997.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, CA, 1984.

R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *ICML 2006*, 2006.

J. Cavazos. Intelligent compilers. In *Cluster Computing, 2008 IEEE International Conference on*, pages 360–368. IEEE, 2008.

J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 185–197. IEEE, 2007.

C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 111–122. IEEE, 2005.

R. Clint Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *ACM SIGPLAN Notices*, volume 40, pages 69–77. ACM, 2005.

K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.

B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *ACM SIGPLAN Notices*, volume 40, pages 78–86. ACM, 2005.

J. H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367 – 378, 2002. doi: 10.1016/S0167-9473(01)00065-2.

G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, et al. Milepost gcc: machine learning based research compiler. 2008.

A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 1–1. USENIX Association, 2009.

S. Grauer-Gray and J. Cavazos. Optimizing and auto-tuning belief propagation on the gpu. *Languages and Compilers for Parallel Computing*, pages 121–135, 2011.

A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Verlag, New York, 2001.

F. Hutter, Y. Hamadi, H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. *Principles and Practice of Constraint Programming-CP 2006*, pages 213–228, 2006.

S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, and P. E. W. Bethel. A generalized framework for auto-tuning stencil computations. In *Cray User Group Conference*, May 2009.

S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU run-time code generation for high-performance computing. arXiv, 2009.

A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 2011.

P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *ACM SIGPLAN Notices*, volume 39, pages 171–182. ACM, 2004.

X. Li, M. Garzarán, and D. Padua. A dynamically tuned sorting library. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 111–122. IEEE, 2004.

Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning gemm for gpus. *Computational Science–ICCS 2009*, pages 884–892, 2009.

Y. Liu, E. Zhang, and X. Shen. A cross-input adaptive framework for gpu program optimizations. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

P. Micikevicius. Analysis-driven optimization. In *GPU Technology Conference*. NVIDIA, 2010.

A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. *Artificial Intelligence: Methodology, Systems, and Applications*, pages 389–409, 2002.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

A. Nukada and S. Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 30. ACM, 2009.

NVIDIA. Compute unified device architecture (cuda) programming guide. 2011.

E. Park, S. Kulkarni, and J. Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 65–74. ACM, 2011.

N. Pinto and D. D. Cox. GPU metaprogramming: A case study in biologically inspired machine vision. In *GPU Computing Gems*, volume 2. Morgan Kauffmann, 2012. to appear.

S. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 107–116. ACM, 2011.

S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204. ACM, 2008.

D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

R. Schapire. The boosting approach to machine learning: An overview. In *In MSRI Workshop on Nonlinear Estimation and Classification*. 2001.

M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *ACM SIGPLAN Notices*, 38(5):77–90, 2003.

R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for automatic performance tuning. *Computational ScienceâĂŤICCS 2001*, pages 117–126, 2001.

R. Vuduc, J. Demmel, and K. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

S. Williams. *Auto-tuning performance on multicore computers*. ProQuest, 2008.

H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.

K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Notices*, volume 38, pages 63–76. ACM, 2003.