Autotuning under Tight Budget Constraints: A Transparent Design of Experiments Approach

Pedro Bruel*†, Steven Quinito Masnada‡, Brice Videau*, Arnaud Legrand*, Jean-Marc Vincent*, Alfredo Goldman†

†University of São Paulo
São Paulo, Brazil
{phrb, gold}@ime.usp.br

‡University of Grenoble Alpes
Inria, CNRS, Grenoble INP, LJK
38000 Grenoble, France
steven.quinito-masnada@inria.fr

*University of Grenoble Alpes
CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
{arnaud.legrand, brice.videau, jean-marc.vincent}@imag.fr

***Abstract*—A large amount of resources is spent writing, porting, and optimizing scientific and industrial High Performance Computing applications, which makes autotuning techniques fundamental to lower the cost of leveraging the improvements on execution time and power consumption provided by the latest software and hardware platforms. Despite the need for economy, most autotuning techniques still require a large budget of costly experimental measurements to provide good results, while rarely providing exploitable knowledge after optimization. The contribution of this paper is a user-transparent autotuning technique based on Design of Experiments that operates under tight budget constraints by significantly reducing the measurements needed to find good optimizations. Our approach enables users to make informed decisions on which optimizations to pursue and when to stop. We present an experimental evaluation of our approach and show it is capable of leveraging user decisions to find the best global configuration of a GPU Laplacian kernel using half of the measurement budget used by other common autotuning techniques. We show that our approach is also capable of finding speedups of up to 50×, compared to `gcc`'s `-O3`, for some kernels from the SPAPT benchmark suite, using up to 10× ~~less~~ fewer measurements than random sampling.**

I. Introduction

Optimizing code for objectives such as performance and power consumption is fundamental to the success and cost-effectiveness of industrial and scientific endeavors in High Performance Computing (HPC). A considerable amount of highly specialized time and effort is spent in porting and optimizing code for GPUs, FPGAs and other hardware accelerators. Experts are also needed to leverage bleeding edge software improvements in compilers, languages, libraries and frameworks. The objective of techniques for the automatic configuration and optimization of ~~High Performance Computing~~ HPC applications, or *autotuning*, is to decrease the cost and time needed to adopt efficient hardware and software. Typical autotuning targets include algorithm selection, source-to-source transformations and compiler configuration.

Autotuning can be studied as a search problem where the objective is to minimize software or hardware metrics. The exploration of the search spaces defined by code and compiler configurations and optimizations presents interesting challenges. Such spaces grow exponentially with the number of parameters~~and their possible values. They~~ , and are also difficult to ~~extensively explore~~ explore extensively due to the often prohibitive costs of hardware utilization, program compilation and execution times. Developing autotuning strategies capable of producing good optimizations while minimizing resource utilization is therefore essential. The capability of acquiring knowledge about an optimization problem is also ~~a desired feature of~~ crucial in an autotuning strategy, since this knowledge can decrease the cost of subsequent optimizations of the same application or for the same hardware.

It is common and usually effective to use search meta-heuristics such as genetic algorithms and simulated annealing in autotuning. These strategies attempt to exploit local search space properties, but are generally incapable of exploiting global structures. Seymour *et al.* [1], Knijnenburg *et al.* [2], and Balaprakash *et al.* [3], [4] report that these strategies are not more effective than a naive uniform random sample of the search space, and usually rely on a large number of measurements or restarts to achieve performance improvements. Search strategies based on gradient descent are also commonly used in autotuning, and also rely on a large number of measurements~~. Their~~ , but their effectiveness diminishes significantly in search spaces with complex local structures. Automated machine learning autotuning strategies [5], [6], [7] are promising ~~in~~ for building models for predicting important ~~optimization~~ parameters, but still rely on a sizable data set for training.

~~Search~~ In summary, search strategies based on meta-heuristics, gradient descent and machine learning require a large number of measurements to be effective, and are usually incapable of providing knowledge about search spaces to users. Since these strategies are not transparent, at the end of each autotuning session it is difficult to decide if and where further exploration is warranted, and often impossible to know which parameters are responsible for the observed improvements. After exploring a search space, ~~it is impossible to confidently deduce its global properties since its~~ deducing any of the space's global properties confidently is impossible, since the space was automatically explored with unknown biases.

The contribution of this paper is an autotuning strategy that leverages existing knowledge about a problem by using an initial performance model that is refined iteratively using performance measurements, statistical analysis, and user input. Our strategy places a heavy weight on decreasing autotuning costs by using a *Design of Experiments* (DoE) methodology to minimize the number of experiments needed to find optimizations. Each iteration uses *Analysis of Variance* (ANOVA) tests

and *linear model regressions* to identify promising subspaces and parameter significance. An architecture- and problem-specific performance model is built iteratively and with user input, which enables making informed decisions about which regions of the search space are worth exploring.

We evaluate the performance of our approach by optimizing a Laplacian Kernel for GPUs, where the search space, the global optimum, and a performance model approximation are known. The budget of measurements was tightly constrained on this experiment. Speedups and budget utilization reduction achieved by our approach on this setting motivated a more comprehensive performance evaluation. We chose the *Search Problems in Automatic Performance Tuning* (SPAPT) [8] benchmark suite for this evaluation, where we obtained diverse results. Out of the 17 SPAPT kernels benchmarked, no speedup could be found for three kernels, but uniform random sampling performed well on all others. For eight of the kernels, our approach found speedups of up to 50×, compared to `gcc`'s `-O3` with no code transformations, while using up to 10× ~~less~~ fewer measurements than random sampling.

The rest of this paper is organized as follows. Section ~~??~~ II presents related work on source-to-source transformation, which is the main optimization target in SPAPT kernels, on autotuning systems and on search space exploration strategies. Section ~~?? discusses the Design of Experiments~~III discusses the implementation of our approach in detail. Section IV discusses the DoE, ANOVA, and linear regression methodology we used to develop our approach. Section ~~?? discusses the implementation of our approach in detail. Section ??~~ V presents the results on the performance evaluation on the GPU Laplacian Kernel and on the SPAPT benchmark suite. Section ~~??~~ VI discusses our conclusions and future work.

II. Background

This section presents the background and related work on source-to-source transformation, autotuning systems and search space exploration strategies.

A. Source-to-Source Transformation

Our approach can be applied to any autotuning domain that expresses optimization as a search problem, although the performance evaluations we present in Section ~~??~~ V were obtained in the domain of source-to-source transformation. Several frameworks, compilers and autotuners provide tools to generate and optimize architecture-specific code [9], [10], [11], [12], [13]. We used BOAST [10] and Orio [9] to perform source-to-source transformations targeting parallelization on CPUs and GPUs, vectorization, loop transformations such as tiling and unrolling, and data structure size and copying.

B. Autotuning

John Rice's Algorithm Selection framework [14] is the precursor of autotuners in various problem domains. In 1997, the PHiPAC system [15] used code generators and search scripts to automatically generate high performance code for matrix multiplication. Since then, systems approached different domains with a variety of strategies. Dongarra *et al.* [16] introduced the ATLAS project, that optimizes dense matrix multiplication routines. The OSKI [17] library provides automatically tuned kernels for sparse matrices. The FFTW [18] library provides tuned C subroutines for computing the Discrete Fourier Transform. Periscope [19] is a distributed online autotuner for parallel systems and single-node performance. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [20] implements abstractions for OpenMP, MPI and OpenCL, and generates optimized parallel code for heterogeneous multi-core architectures.

A different approach is to combine generic search algorithms and problem representation data structures in a single system that enables the implementation of autotuners for different domains. The PetaBricks [13] project provides a language, compiler and autotuner, enabling the definition and selection of multiple algorithms for the same problem. The ParamILS framework [21] applies stochastic local search algorithms to algorithm configuration and parameter tuning. The OpenTuner framework [22] provides ensembles of techniques that search the same space in parallel, while exploration is managed by a multi-armed bandit strategy.

C. Search Space Exploration Strategies

Figure ~~??~~ 1 shows the contour of a search space defined by a function of the form $z = x^2 + y^2 + \varepsilon$, where ε is a local perturbation, and the exploration of that search space by six different strategies. In such a simple search space, even a uniform random sample can find points close to the optimum, despite not exploiting geometry. A Latin Hypercube [23] sampling strategy covers the search space more evenly, but still does not exploit ~~its~~ the space's geometry. Strategies based on neighborhood exploration such as simulated annealing

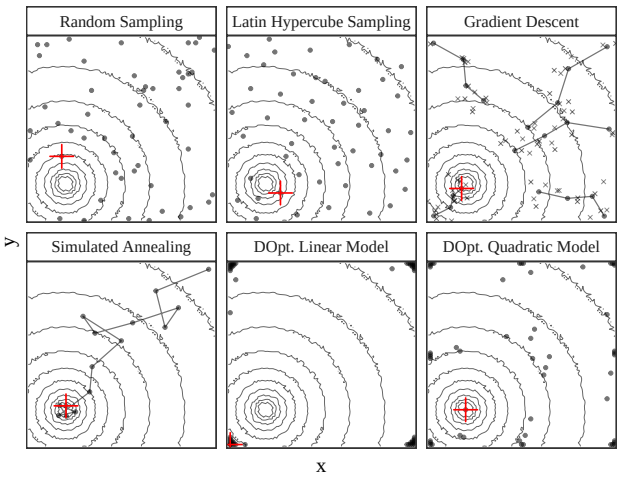

Figure 1: Exploration of the search space, using a fixed budget of 50 points. The red "+" represents the best point found by each strategy, and "×"s denote neighborhood exploration

and gradient descent can exploit local structures, but may get trapped in local minima. Their performance is strongly dependent on search starting point. These strategies do not leverage global search space structure, or provide exploitable knowledge after optimization.

Measurement of the kernels optimized on the performance evaluations in Section ~~??~~ V can exceed 20 minutes, including the time of code transformation, compilation, and execution. Measurements in other problem domains can take much longer to complete. This strengthens the motivation to consider search space exploration strategies capable of operating under tight budget constraints. These strategies have been developed and improved by statisticians for a long time, and can be grouped under the ~~Design of Experiments~~ DoE term.

The D-Optimal sampling strategies shown on the two right-most bottom panels of Figure ~~??~~ 1 are based on the ~~Design of Experiments~~ DoE methodology, and leverage previous knowledge about search spaces for an efficient exploration. These strategies provide transparent analyses that enable focusing on interesting subspaces. In the next section we ~~present the Design of Experiments methodology used to implement our approach .~~ describe our approach to autotuning based on the DoE methodology.

III. Autotuning with Design of Experiments

An *experimental design* determines a selection of experiments whose objective is to identify the relationships between *factors* and *responses*. While factors and responses can refer to different concrete entities in other domains, in computer experiments factors can be configuration parameters for algorithms and compilers, ~~for example,~~ and responses can be the execution time or memory consumption of a program. Each possible value of a factor is called a *level*. The *effect* of a factor on the measured response, without ~~its~~ the factor's *interactions* with other factors, is the *main effect* of that factor. Experimental designs can be constructed with different goals, such as identifying the main effects or building an analytical model for the response.

In this section we discuss in detail our iterative DoE approach to autotuning. Figure 2 presents an overview of our approach, with numbered steps. In step 1 we define the factors and levels that compose the search space of the target problem, in step 2 we select an initial performance model, and in step 3 we generate an experimental design. We run the experiments in step 4 and then, as we discuss in the next section, we identify significant factors with an ANOVA test in step 5. This enables selecting and fitting a new performance model in steps 6 and 7. The new model is used in step 8 for predicting levels for each significant factor. We then go back to step 3, generating a new design for the new problem subspace with the remaining factors. Informed decisions made by the user at each step guide the outcome of each iteration.

Step 1 of our approach is to define target factors and which of their levels are worth exploring. Then, the user must select an initial performance model in step 2. Compilers typically expose many 2-level factors in the form of configuration flags, and the performance model for a single flag can only be a linear term, since there are only 2 values to measure. Interactions between flags and numerical factors such as block sizes in CUDA programs or loop unrolling amounts are also common. Deciding which levels to include for these kinds of factors requires more careful analysis. For example, if we suspect the performance model has a quadratic term for a certain factor, the design should include at least three factor levels. The ordering between the levels of other compiler parameters, such as `-O(0,1,2,3)`, is not obviously translated to a number. Factors like these are named *categorical*, and must be treated differently when constructing designs in step 3 and analyzing results in step 5.

We decided to use D-Optimal designs because their construction techniques enable mixing categorical and numerical factors in the same screening design, while biasing sampling according to the performance model. This enables the autotuner to exploit global search space structures if we use the right model. When constructing a D-Optimal design in step 3 the user can require that specific points in the search space are included, or that others are not. Algorithms for constructing D-Optimal designs are capable of adapting to these requirements by optimizing a starting design. Before settling on D-Optimal designs, we explored other design construction techniques such as the Plackett-Burman [24] screening designs shown in the next section, the *contractive replacement* technique of Addelman-Kempthorne [25] and the *direct generation* algorithm by Grömping and Fontana [26] . These techniques have strong requirements on design size and level mixing, so we opted for a more flexible technique that would enable exploring a more comprehensive class of autotuning problems.

After the design is constructed in step 3, we run each selected experiment in step 4. This step can run in parallel since experiments are independent. Not all target programs run successfully in their entire input range, making runtime failures common in this step. The user can decide whether to construct a new design using the successfully completed

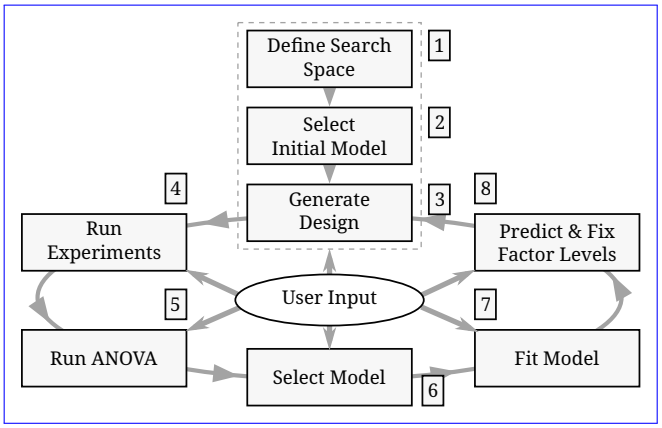

Figure 2: Overview of the DoE approach to autotuning proposed in this paper

experiments or to continue to the analysis step if enough experiments succeed.

After running the ANOVA test in step 5, the user should apply domain knowledge to analyze the ANOVA table and determine which factors are significant. Certain factors might not appear significant and should not be included in the regression model. Selecting the model after the ANOVA test in step 6 also benefits from domain knowledge.

A central assumption of ANOVA is the *homoscedasticity* of the response, which can be interpreted as requiring the observed error on measurements to be independent of factor levels and of the number of measurements. Fortunately, there are statistical tests and corrections for lack of homoscedasticity. Our approach uses the homoscedasticity check and correction by power transformations from the `car` package [27] of the `R` language.

We fit the selected model to our design's data in step 7, and use the fitted model in step 8 to find levels that minimize the response. The choice of the method used to find these levels depends on factor types and on the complexity of the model and search space. If factors have discrete levels, neighborhood exploration might be needed to find levels that minimize the response around predicted levels. Constraints might put predicted levels on an undefined or invalid region on the search space. This presents challenge, because the borders of valid regions would have to be explored.

In step 8 we also fix factor levels to those predicted to achieve best performance. The user can also decide the level of trust placed on the prediction at this step, by keeping other levels available. In step 8 we perform a reduction of problem dimension by eliminating factors and decreasing the size of the search space. If we identified significant parameters correctly, we will have restricted further search to better regions. In the next section we present a simple fictional application our approach that illustrates the fundamentals of the DoE methodology, screening designs and D-Optimal designs.

IV. Design of Experiments

In this section we first present the assumptions of a traditional ~~Design of Experiments~~ DoE methodology using an example of *2-level screening designs*, ~~which are~~ an efficient way to identify main effects. We then discuss ~~some~~ techniques for the construction of efficient designs for factors with arbitrary numbers and types of levels, and present *D-Optimal* designs, the technique ~~we use in the approach presented in~~ used in this paper.

A. Screening & Plackett-Burman Designs

~~Screening designs provide a parsimonious way to identify~~ Screening designs identify parsimoniously the main effects of 2-level factors in the initial stages of studying a problem. While interactions are not considered at this stage, identifying main effects early enables focusing on a smaller set of factors on subsequent ~~more detailed~~ experiments. A specially efficient design construction technique for screening designs was presented by Plackett and Burman [24] in 1946, and is available in the `FrF2` package [28] of the `R` language [29].

Despite having strong restrictions on the number of factors ~~they support~~supported, Plackett-Burman designs enable the identification of main effects of n factors with $n+1$ experiments. Factors may have many levels, but Plackett-Burman designs can only be constructed for 2-level factors. Therefore, before constructing a Plackett-Burman design we must identify *high* and *low* levels for each factor.

Assuming a ~~crude~~ linear relationship between factors and the response is fundamental for running ANOVA tests using a Plackett-Burman design. Consider the ~~following linear relationship :~~ linear relationship

$$\mathbf{Y} = \boldsymbol{\beta}\mathbf{X} + \varepsilon, \tag{1}$$

where ε is the error term, $\mathbf{Y}$ is the observed response, $\mathbf{X} = \{1, x_1, \ldots, x_n\}$ is the set of n 2-level factors, and $\boldsymbol{\beta} = \{\beta_0, \ldots, \beta_n\}$ is the set with the *intercept* β_0 and the corresponding *model coefficients*. ANOVA tests can rigorously compute the significance of each factor~~. We~~, we can think of that intuitively by noting that less ~~relevant~~ significant factors will have corresponding values in $\boldsymbol{\beta}$ close to zero.

~~We now present an example to illustrate~~ The next example illustrates the screening methodology. Suppose we wish to minimize a performance metric Y of a problem with factors $x_1, \ldots, x_8$ assuming values ~~are~~ in $\{-1, -0.8, -0.6, \ldots, 0.6, 0.8, 1\}$. Each $y_i \in Y$ is ~~computed using the following equation:~~ defined as

$$\begin{aligned} y_i = &- 1.5x_1 + 1.3x_3 + 3.1x_5 + \\ &- 1.4x_7 + 1.35x_8^2 + 1.6x_3x_5 + \varepsilon. \end{aligned} \tag{2}$$

Suppose that, for the purpose of this example, the computation is done by a very expensive black-box procedure. Note that factors $\{x_2, x_4, x_6\}$ have no contribution to the response, and we can think of the error term ε as representing not only noise, but our uncertainty regarding the model. Higher amplitudes of ε might make ~~it harder to justify~~ isolating factors with low significance harder to justify.

To ~~efficiently~~ study this problem ~~we decide~~ efficiently we decided to construct a Plackett-Burman design, which minimizes the experiments needed to identify significant factors. The analysis of this design will enable decreasing the dimension of the problem. Table I presents the Plackett-Burman design we generated. It contains high and low values, chosen to be -1 and 1, for the factors $x_1, \ldots, x_8$, and the observed response $\mathbf{Y}$. ~~As is common when constructing screening designs, we had to add~~ It is a required step to add the 3 "dummy" factors $d_1, \ldots, d_3$ to complete the 12 columns needed to construct a Plackett-Burman design for 8 factors [24].

So far, we have performed steps 1, 2, and 3 from Figure 2. We use our initial assumption ~~shown~~ in Equation (1) to identify the most ~~relevant~~ significant factors by performing an ANOVA test~~. The resulting ANOVA table is~~, which is step 5 from Figure 2. The results are shown in Table II, where the *significance* of each factor ~~can be~~ is interpreted from the F-test and ~~P(< F)~~ Pr(> F) values. Table II uses "$*$", as is convention

in the R language, to represent the significance values for each factor.

We see on Table II that factors $\{x_3, x_5, x_7, x_8\}$ have at least one "$*$" of significance. For the purpose of this example, this is sufficient reason to include them in our linear model for the next step. We decide as well to discard factors $\{x_2, x_4, x_6\}$ ~~in our modelfor now~~from our model, due to their low significance. We see that factor x_1 has a significance mark of "$\cdot$", but comparing ~~its~~ F-test and ~~P(< F)~~ Pr(> F) values we decide that they are fairly smaller than the values of factors that had no significance~~at all~~, and we keep this factor.

Moving forward ~~,~~ to steps 6, 7, and 8 in Figure 2, we will build a linear model using factors $\{x_1, x_3, x_5, x_7, x_8\}$, fit the model using the values of Y we obtained when running our design, and use ~~the coefficients of this fitted model~~ model coefficients to predict the levels ~~for~~ of each factor that minimize the real response. We can do that because these factors are numerical, even though only discrete values are allowed.

We now proceed to the prediction step, where we wish to identify the levels of factors $\{x_1, x_3, x_5, x_7, x_8\}$ that minimize our fitted model, without running any new experiments. This can be done by, for example, using a gradient descent algorithm or finding the point where the derivative of the function given by the linear regression equals to zero.

Table III compares the prediction for Y from our linear model with the selected factors $\{x_1, x_3, x_5, x_7, x_8\}$ with the actual global minimum Y for this problem. Note that factors $\{x_2, x_4, x_6\}$ are included for the global minimum. This happens here because of the error term ε, but could also be interpreted as due to model uncertainty.

Using 12 measurements and a simple linear model, the predicted best value of Y was around $10\times$ larger than the global optimum. Note that the model predicted the correct levels for x_3 and x_5, and almost ~~predicted correctly~~ for x_7. The linear model predicted wrong levels for x_1, perhaps due to this factor's interaction with x_3, and for x_8. Arguably, it would be impossible to predict the correct level for x_8 using this linear model, since a quadratic term composes the true formula of Y. As we showed in Figure ??1, a D-Optimal design using a linear model could detect the significance of a quadratic term, but the resulting regression will often ~~predict the wrong minimum point~~lead to the wrong level.

We can improve upon this result if we introduce some information about the problem and use a more flexible design construction technique. Next, we will discuss the construction of efficient designs using problem-specific formulas and continue the optimization of our example.

Table I: Randomized Plackett-Burman design for factors $x_1, \ldots, x_8$, using 12 experiments and "dummy" factors $d_1, \ldots, d_3$, and computed response $\mathbf{Y}$

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	d_1	d_2	d_3	Y
1	-1	1	1	1	-1	-1	-1	1	-1	1	13.74
-1	1	-1	1	1	-1	1	1	1	-1	-1	10.19
-1	1	1	-1	1	1	1	-1	-1	-1	1	9.22
1	1	-1	1	1	1	-1	-1	-1	1	-1	7.64
1	1	1	-1	-1	-1	1	-1	1	1	-1	8.63
-1	1	1	1	-1	-1	-1	1	-1	1	1	11.53
-1	-1	-1	1	-1	1	1	-1	1	1	1	2.09
1	1	-1	-1	-1	1	-1	1	1	-1	1	9.02
1	-1	-1	-1	1	-1	1	1	-1	1	1	10.68
1	-1	1	1	-1	1	1	1	-1	-1	-1	11.23
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5.33
-1	-1	1	-1	1	1	-1	1	1	1	-1	14.79

Table II: Shortened ANOVA table for the fit of the naive model, with significance intervals from the R language

	F value	Pr(> F)	Signif.
x_1	8.382	0.063	$\cdot$
x_2	0.370	0.586	
x_3	80.902	0.003	**
x_4	0.215	0.675	
x_5	46.848	0.006	**
x_6	5.154	0.108	
x_7	13.831	0.034	*
x_8	59.768	0.004	**

B. D-Optimal Designs

The application of ~~Design of Experiments~~ DoE to autotuning problems requires design construction techniques that support factors of arbitrary types and number of levels. Autotuning problems typically combine factors such as binary flags, integer and floating point numerical values, and unordered enumerations of abstract values. ~~Previously, to construct a~~ Because Plackett-Burman ~~design for our example~~ designs only support 2-level factors, we had to restrict ~~our factors to the extremes of their levels in the interval $\{-1, -0.8, -0.6, \ldots, 0.6, 0.8, 1\}$, because such designs only support 2-level factors~~factor levels to interval extremities in our example. We have seen that this restriction makes it difficult to measure the significance of quadratic terms~~in the model~~. We will now show how to ~~further~~ optimize our example further by using *D-Optimal designs*, which increase the number of levels we can efficiently screen for and enables detecting the significance of more complex ~~model~~ terms.

To construct a D-Optimal design it is necessary to choose an initial model, which can be done based on previous experiments or on expert knowledge of the problem. Once a model is selected, algorithmic construction is performed by searching for the set of experiments that minimizes *D-Optimality*, a measure of the *variance* of the *estimators* for the *regression coefficients* associated with the selected model. This search is usually done by swapping experiments from the current candidate ~~set~~ design with experiments from a pool of

Table III: Comparison of the response Y predicted by the linear model and the true global minimum. Factors used in the model are bolded

	$\boldsymbol{x_1}$	x_2	$\boldsymbol{x_3}$	x_4	$\boldsymbol{x_5}$	x_6	$\boldsymbol{x_7}$	$\boldsymbol{x_8}$	Y
Lin.	-1.0	–	-1.0	–	-1.0	–	1.0	-1.0	-1.046
Min.	1.0	-0.2	-1.0	0.6	-1.0	0.4	0.8	0.0	-9.934

Table IV: D-Optimal design constructed for the factors $\{x_1, x_3, x_5, x_7, x_8\}$ and computed response Y

x_1	x_3	x_5	x_7	x_8	Y
-1.0	-1.0	-1.0	-1.0	-1.0	2.455
-1.0	1.0	1.0	-1.0	-1.0	6.992
1.0	-1.0	-1.0	1.0	-1.0	-7.776
1.0	1.0	1.0	1.0	-1.0	4.163
1.0	1.0	-1.0	-1.0	0.0	0.862
-1.0	1.0	1.0	-1.0	0.0	5.703
1.0	-1.0	-1.0	1.0	0.0	-9.019
-1.0	-1.0	1.0	1.0	0.0	2.653
-1.0	-1.0	-1.0	-1.0	1.0	1.951
1.0	-1.0	1.0	-1.0	1.0	0.446
-1.0	1.0	-1.0	1.0	1.0	-2.383
1.0	1.0	1.0	1.0	1.0	4.423

possible experiments, according to certain rules, until some stopping criterion is met. In the example in this section ~~, as well as in the approach presented in this paper,~~ and in our approach we use Fedorov's algorithm [30] for constructing D-Optimal designs, implemented in `R` in the `AlgDesign` package [31].

~~Going back to~~ In our example, suppose that in addition to using our previous screening results we decide to hire an expert in our problem's domain. The expert confirms our initial assumptions that the factor x_1 should be included in our model since it is usually ~~relevant~~ significant for this kind of problem and has a strong interaction with factor x_3. She also mentions we should replace the linear term for x_8 by a quadratic term for this factor.

Using our previous screening and the domain knowledge provided by our expert, we choose a new performance model and use it to construct a D-Optimal design using Fedorov's algorithm. Since we need enough degrees of freedom to fit our model, we construct the design with 12 experiments shown in Table IV. Note that the design includes levels -1, 0, and 1 ~~levels~~ for factor x_8. The design will sample from different regions of the search space due to the quadratic term, as was shown in Figure ~~??~~1.

We ~~are now going to~~ now fit this model using the results of the experiments in our ~~D-Optimal~~ design. Table V shows the model fit table and compares the estimated and real model coefficients. This example illustrates that the ~~Design of Experiments~~ DoE approach can achieve close model estimations using few resources, provided ~~it~~ the approach is able to use user input to identify ~~relevant factors~~significant factors, and knowledge about the problem domain to tweak the model.

Table VI compares the global minimum ~~in~~ of this example with the predictions made by our initial linear model from the screening step, and our improved model~~from this step~~. Using screening, D-Optimal designs, and domain knowledge, we found an optimization within 10% of the global optimum while computing Y only 24 times. We were able to do that by first reducing the ~~dimension of the problem~~problem's dimension when we eliminated ~~irrelevant~~ insignificant factors in the screening step. We then constructed a more careful exploration of this new problem subspace, ~~helped~~ aided by domain knowledge provided by an expert. Note that we could have reused some of the 12 experiments from the previous step to reduce the size of the new design ~~even~~ further.

We are able to explain the performance improvements we obtained in each step of the process, because we finish steps with a performance model and a performance prediction. Each factor is included or removed using information obtained in statistical tests, or expert knowledge. If we need to optimize this problem again, for a different architecture or with larger input, ~~for example,~~ we could start exploring the search space with a less naive model. We could also continue the optimization of this problem by ~~further~~ exploring levels of factors $\{x_2, x_4, x_6\}$. The significance of these factors could now be detectable by ANOVA tests since the other factors are now fixed. If we still cannot identify any significant factor, it might be advisable to spend the remaining budget using another exploration strategy such as uniform random or ~~lating~~ latin hypercube sampling.

The process of screening for factor significance using ANOVA and fitting a new model using acquired knowledge is ~~essentially a step in the transparent Design of Experiments approach we present in the next section.~~

V. ~~Autotuning with Design of Experiments~~

~~In this section we discuss in detail our iterative Design of Experiments approach to autotuning. At the start of the process it is necessary to define the factors and levels that compose the search space of the target problem, select an initial performance model, and generate an experimental design. Then, as discussed in the previous section, we identify relevant factors by running an ANOVA test on the results. This enables selecting and fitting a new performance model, which is used for predicting levels for each relevant factor. The process can then restart, generating a new design for the new problem subspace with the remaining factors. Informed decisions made by the user play a central role in each iteration, guiding and speeding up the process. Figure ?? presents an overview of our approach.~~

~~Overview of the Design of Experiments approach to autotuning proposed in this paper~~

~~The first step of our approach is to define which are the target factors and which levels of each factor are worth exploring. Then, the user must select an initial performance~~

Table V: Correct model fit comparing real and estimated coefficients, with significance intervals from the `R` language

	Real	Estimated	t value	Pr(> |t|)	Signif.
Intercept	0.000	0.050	0.305	0.776	
x_1	-1.500	-1.452	-14.542	0.000	***
x_3	1.300	1.527	15.292	0.000	***
x_5	3.100	2.682	26.857	0.000	***
x_7	-1.400	-1.712	-17.141	0.000	***
x_8	0.000	-0.175	-1.516	0.204	
x_8^2	1.350	1.234	6.180	0.003	**
$x_1 x_3$	1.600	1.879	19.955	0.000	***

Table VI: Comparison of the response Y predicted by our models and the true global minimum. Factors used in the models are bolded

	$\boldsymbol{x_1}$	x_2	$\boldsymbol{x_3}$	x_4	$\boldsymbol{x_5}$	x_6	$\boldsymbol{x_7}$	$\boldsymbol{x_8}$	Y
Quad.	1.0	–	-1.0	–	-1.0	–	1.0	0.0	-9.019
Lin.	-1.0	–	-1.0	–	-1.0	–	1.0	-1.0	-1.046
Min.	1.0	-0.2	-1.0	0.6	-1.0	0.4	0.8	0.0	-9.934

model. Compilers typically expose many 2-level factors in the form of configuration flags. The performance model for a single flag can only be a linear term, since there are only 2 values to measure. Interactions between flags can also be considered in an initial model. Numerical factors are also common, such as block sizes for CUDA programs or loop unrolling amounts. Deciding which levels to include for these kinds of factors requires a more careful analysis. For example, if we suspect the performance model has a quadratic term for a certain factor, we should include at least three of its levels. The ordering between the levels of other compiler parameters, such as `-O(0,1,2,3)`, is not obviously translated to a number. Factors like these are named *categorical*, and must be treated differently when constructing designs and analyzing the results.

We decided to use D-Optimal designs because their construction techniques enable mixing categorical and numerical factors in the same screening design, while biasing sampling according to the performance model. This enables the autotuner to exploit global search space structures if we use the right model. When constructing a D-Optimal design the user can require that specific points in the search space are included, or that others are not. Algorithms for constructing D-Optimal designs are capable of adapting to these requirements by optimizing a starting design. Before settling on D-Optimal designs, we explored other design construction techniques such as the Plackett-Burman [24] screening designs shown in the previous section, the *contractive replacement* technique of Addelman-Kempthorne [25] and the *direct generation* algorithm by Grömping and Fontana [26]. These techniques have strong requirements on design size and level mixing, so we opted for a more flexible technique that would enable exploring a more comprehensive class of autotuning problems.

After the design is constructed we run each selected experiment. This step can be done in parallel since experiments are independent. Runtime failures are common in this step due to problems such as incorrect output. The user can decide whether to construct a new design using the successfully completed experiments or to continue to the analysis step if enough experiments succeed.

The next four steps of an iteration, shown in Figure **??**, were discussed in detail in the previous section. User input is fundamental to the success of these steps. After running the ANOVA test, the user should apply domain knowledge to analyze the ANOVA table and determine which factors are relevant. Certain factors might not appear relevant, in which case the user should not include them in the regression model, but save them for further exploration. Selecting the model after the ANOVA test also benefits from domain knowledge. The impact of the number of threads used by a parallel program on its performance is usually modeled using an inverse term, which accounts for the speedup of adding more threads, plus a linear term, which accounts for the overhead of their management.

A central assumption of ANOVA is the *homoscedasticity* of the response, which can be interpreted as requiring the observed error on measurements to be independent of factor levels and of the number of measurements. Fortunately, up to a point, there are statistical tests and corrections for lack of homoscedasticity. Our approach uses the homoscedasticity check and correction by power transformations from the `car` package [27] of the `R` language before every ANOVA step.

The prediction step uses the fitted model to find factor levels that minimize the response. The choice of the method to find these levels depends on factor types and model and search space complexity. If factors have discrete levels, neighborhood exploration might be needed to find valid levels that minimize the response around the predicted levels. Validity constraints might put predicted levels on an undefined or invalid region on the search space. This presents a harder challenge, where the borders of valid regions would have to be explored.

The last step of an iteration is fixing factor levels to those predicted to have best performance. The user can also decide the level of trust that will be placed on the model and ANOVA at this step by allowing other levels. This step performs a reduction on the dimension of the problem by eliminating factors and decreasing the size of the search space. If we identify relevant parameters correctly, we will have restricted further search to better regions of the search space. equivalent to steps 5, 6, and 7 in Figure 2. In the next section we present evaluate the performance of our approach in scenarios that differ on search space size, availability and complexityDoE approach in two scenarios.

V. Performance Evaluation

In this section we present performance evaluations of our approach in two scenarios that differ on search space size and complexity.

A. GPU Laplacian Kernel

We first evaluated the performance of our approach in a Laplacian Kernel implemented using BOAST [10] and targeting the *Nvidia K40c* GPU. The objective was to minimize the *time to compute each pixel* by finding the best level combination for the factors listed in Table **??**VII. Considering only factors and levels, the size of the search space is 1.9×10^5, but removing points that fail at runtime yields a search space of size 2.3×10^4. The complete search space took 154 hours to be evaluated on *Debian Jessie*, using an *Intel Xeon E5-2630v2* CPU, `gcc` version `4.8.3` and *Nvidia* driver version `340.32`.

Table VII: Parameters of the Laplacian Kernel

Factor	Levels	Short Description
`vector_length`	$2^0, \ldots, 2^4$	Size of support arrays
`load_overlap`	*true*, *false*	Load overlaps in vectorization
`temporary_size`	$2, 4$	Byte size of temporary data
`elements_number`	$1, \ldots, 24$	Size of equal data splits
`y_component_number`	$1, \ldots, 6$	Loop tile size
`threads_number`	$2^5, \ldots, 2^{10}$	Size of thread groups
`lws_y`	$2^0, \ldots, 2^{10}$	Block size in y dimension

We applied domain knowledge to construct the following initial performance model:

$$\begin{aligned}\texttt{time\_per\_pixel} \sim\ & \texttt{y\_component\_number} + \frac{1}{\texttt{y\_component\_number}} + \\ & \texttt{load\_overlap} + \texttt{temporary\_size} + \\ & \texttt{vector\_length} + \texttt{lws\_y} + \frac{1}{\texttt{lws\_y}} + \\ & \texttt{elements\_number} + \texttt{threads\_number} + \\ & \frac{1}{\texttt{elements\_number}} + \frac{1}{\texttt{threads\_number}}.\end{aligned} \tag{3}$$

This performance model was used by the Iterative Linear Model (LM) algorithm and by our D-Optimal Design approach (DLMT). ~~The LM algorithm is~~ LM is almost identical to our approach, described Section ~~??, except for the design generation step, where~~ III, but it uses a fixed-size random sample of the search space instead of generating D-Optimal designs. We compared the performance of our approach with the following algorithms: uniform Random Sampling (RS); Latin Hypercube Sampling (LHS); Greedy Search (GS); Greedy Search with Restart (GSR); and Genetic Algorithm (GA). Each algorithm performed *at most* 125 measurements over 1000 repetitions, without user intervention.

Since we measured the entire valid search space, we could use the *slowdown* relative to the *global minimum* to compare ~~the performanceof algorithms~~algorithm performance. Table VIII shows the mean, minimum and maximum slowdowns in comparison to the global minimum, for each algorithm. It also shows the mean and maximum budget used by each algorithm. Figure ~~??~~ 3 presents histograms with the count of the slowdowns found by each of the 1000 repetitions. Arrows point the maximum slowdown found by each algorithm. Note that GS's maximum slowdown was left out of range to help the comparison between the other algorithms.

Table VIII: Slowdown and budget used by 7 optimization methods on the Laplacian Kernel, using a budget of 125 points with 1000 repetitions

	Mean	Min.	Max.	Mean Budget	Max. Budget
RS	1.10	1.00	1.39	120.00	120.00
LHS	1.17	1.00	1.52	98.92	125.00
GS	6.46	1.00	124.76	22.17	106.00
GSR	1.23	1.00	3.16	120.00	120.00
GA	1.12	1.00	1.65	120.00	120.00
LM	1.02	1.01	3.77	119.00	119.00
DLMT	1.01	1.01	1.01	54.84	56.00

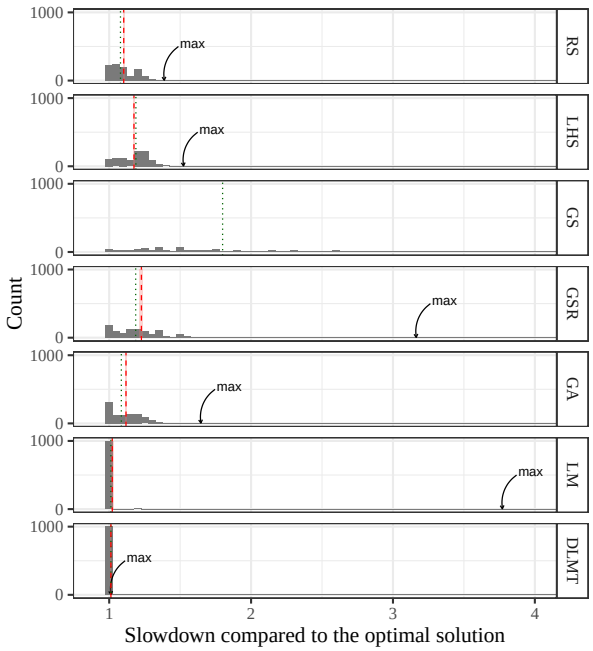

Figure 3: Distribution of slowdowns in relation to the global minimum for 7 optimization methods on the Laplacian Kernel, using a budget of 125 points over 1000 repetitions

All algorithms performed relatively well in this kernel, with only ~~Greedy Search (GS)~~ GS not being able to find slowdowns smaller than $4\times$ in some runs. As expected, other search algorithms had results similar to ~~Random Sampling (RS). The LM algorithm~~ RS. LM was able to find slowdowns close to the global minimum on most runs, but some runs could not find slowdowns smaller than $4\times$. Our approach ~~was able to find~~ reached a slowdown of 1% from the global minimum ~~in all~~ *in all* of the 1000 runs while using *at most* ~~less~~ fewer than half of the allotted budget.

We implemented a simple approach for the prediction step in this problem, choosing the best value of our fitted models on the complete set of valid level combinations. This was possible for this problem since all valid combinations were known~~and fit in memory~~. For problems were the search space is too large to be generated, we would have to either adapt this step and run the prediction on a sample or minimize the model using the differentiation strategies mentioned in Section ~~??~~IV-A.

This kernel provided ideal conditions for using our approach, where the performance model is approximately known and the complete valid search space is small enough to be ~~stored and~~ used for prediction. The global minimum also appears to not be isolated in a region of points with bad performance, since our approach was able to exploit ~~search~~ space geometry. We will now present a performance evaluation of our approach in a larger and more comprehensive benchmark.

B. SPAPT Benchmark Suite

The SPAPT [8] benchmark suite provides parametrized CPU kernels from different ~~High Performance Computing~~ HPC domains. The kernels shown in Table ~~??~~ IX are implemented

Table IX: Kernels from the SPAPT benchmark used in this evaluation

Kernel	Operation	Factors	Size
atax	Matrix transp. & vector mult.	18	2.6×10^{16}
dgemv3	Scalar, vector & matrix mult.	49	3.8×10^{36}
gemver	Vector mult. & matrix add.	24	2.6×10^{22}
gesummv	Scalar, vector, & matrix mult.	11	5.3×10^{9}
hessian	Hessian computation	9	3.7×10^{7}
mm	Matrix multiplication	13	1.2×10^{12}
mvt	Matrix vector product & transp.	12	1.1×10^{9}
tensor	Tensor matrix mult.	20	1.2×10^{19}
trmm	Triangular matrix operations	25	3.7×10^{23}
bicg	Subkernel of BiCGStab	13	3.2×10^{11}
lu	LU decomposition	14	9.6×10^{12}
adi	Matrix sub., mult., & div.	20	6.0×10^{15}
jacobi	1-D Jacobi computation	11	5.3×10^{9}
seidel	Matrix factorization	15	1.3×10^{14}
stencil3d	3-D stencil computation	29	9.7×10^{27}
correlation	Correlation computation	21	4.5×10^{17}

using the code annotation and transformation tools provided by Orio [9]. Search space sizes are ~~overall~~ larger than in the Laplacian Kernel example. Kernel factors are either integers in an interval, such as loop unrolling and register tiling amounts, or binary flags that control parallelization and vectorization.

We used the Random Sampling (RS) implementation available in Orio and integrated an implementation of our approach (DLMT) to the system. We omitted the other Orio algorithms because other studies using SPAPT kernels [3], [4] showed that their performance is similar to RS regarding budget usage. The global minima are not known for any of the problems, and ~~problem~~ search spaces are too large to allow complete measurements. Therefore, we used the performance of each application compiled with `gcc`'s `-O3`, with no code transformations, as a *baseline* for computing the *speedups* achieved by each strategy. We performed 10 autotuning repetitions for each kernel using ~~random sampling and our approach~~RS and DLMT, using a budget of *at most* 400 measurements. ~~Our approach~~ DLMT was allowed to perform only 4 of the iterations shown in Figure ~~??~~2. Experiments were performed using Grid5000 [32], on *Debian Jessie*, using an *Intel Xeon E5-2630v3* CPU and `gcc` version `6.3.0`.

The time to measure each kernel varied from a few seconds to up to 20 minutes. ~~We discovered in testingthat~~ In testing, some transformations caused the compiler to enter an internal optimization process that did not stop for over 12 hours. We did not study why these cases ~~took so longto complete~~delayed for so long, and implemented an execution timeout of 20 minutes, considering cases that took longer than that to compile to be runtime failures.

Similar to the previous example, we automated factor elimination based on ANOVA tests so that a comprehensive evaluation could be performed. We also did not tailor initial performance models, which were the same for all kernels. Initial models had a linear term for each factor with two or more levels, plus quadratic and cubic terms for factors with sufficient levels. Although automation and identical initial models might have limited the improvements at each step of our application, our results show that it still succeeded in decreasing the budget needed to find significant speedups for some kernels.

Figure ~~??~~ 4 presents the *speedup* found by each run of RS and DLMT, plotted against the algorithm *iteration* where that speedup was found. We divided the kernels into 3 groups according to the results. The group where no algorithm found any speedups contains 3 kernels and is marked with "[0]" and *blue* headers. The group where both algorithms found similar speedups, in similar iterations, contains 6 kernels and is marked with "[=]" and *orange* headers. The group where DLMT found similar speedups using a significantly smaller budget than RS contains 8 kernels and is marked with "[+]" and *green* headers. Ellipses delimit an estimate of where 95% of the underlying distribution lies, and a dashed line marks the `-O3` baseline. In comparison to RS, our approach significantly decreased the average number of iterations needed to find speedups for the 8 kernels in the green group.

Figure ~~??~~ 5 shows the search space exploration performed by RS and DLMT. It uses the same color groups as Figure ~~??~~4, and shows the distribution of the speedups that ~~where~~ were found during all repetitions of the experiments. Histogram areas corresponding to DLMT are usually smaller because it always stopped at 4 iterations, while RS always performed 400 measurements. This is particularly visible in `lu`, `mvt`, and `jacobi`. We also observe that the quantity of configurations with high speedups found by DLMT is higher, even for kernels on the orange group. This is noticeable in `gemver`, `bicgkernel`, `mm` and `tensor`, and means that ~~our approach~~ DLMT spent less of the budget exploring configurations with small speedups or slowdowns, in comparison with RS.

Analyzing the significant performance parameters identified by our automated approach for every kernel, we were able to identify interesting relationships between parameters and performance. In `bicgkernel`, for example, DLTM identified a linear relationship for OpenMP and scalar replacement optimizations, and quadratic relationships between register and cache tiling, and loop unrolling. This is an example of the transparency in the optimization process that can be achieved with a DoE approach.

Our approach used a generic initial performance model for all kernels, but since it iteratively eliminates factors and model terms based on ANOVA tests, it was still able to exploit global search space structures for kernels in the orange and green groups. Even in this automated setting, the results with SPAPT kernels illustrate the ability our approach has to reduce the budget needed to find good speedups by efficiently exploring search spaces.

~~Cost of best points found on each run, and the iteration where they were found. RS and DLMT found no speedups with similar budgets for kernels marked with "0" and *blue* headers, and similar speedups with similar budgets for kernels marked with "=" and *orange* headers. DLMT found similar speedups using smaller budgets for kernels marked with "+" *green* headers. Ellipses delimit an estimate of where 95% of the underlying distribution lies~~

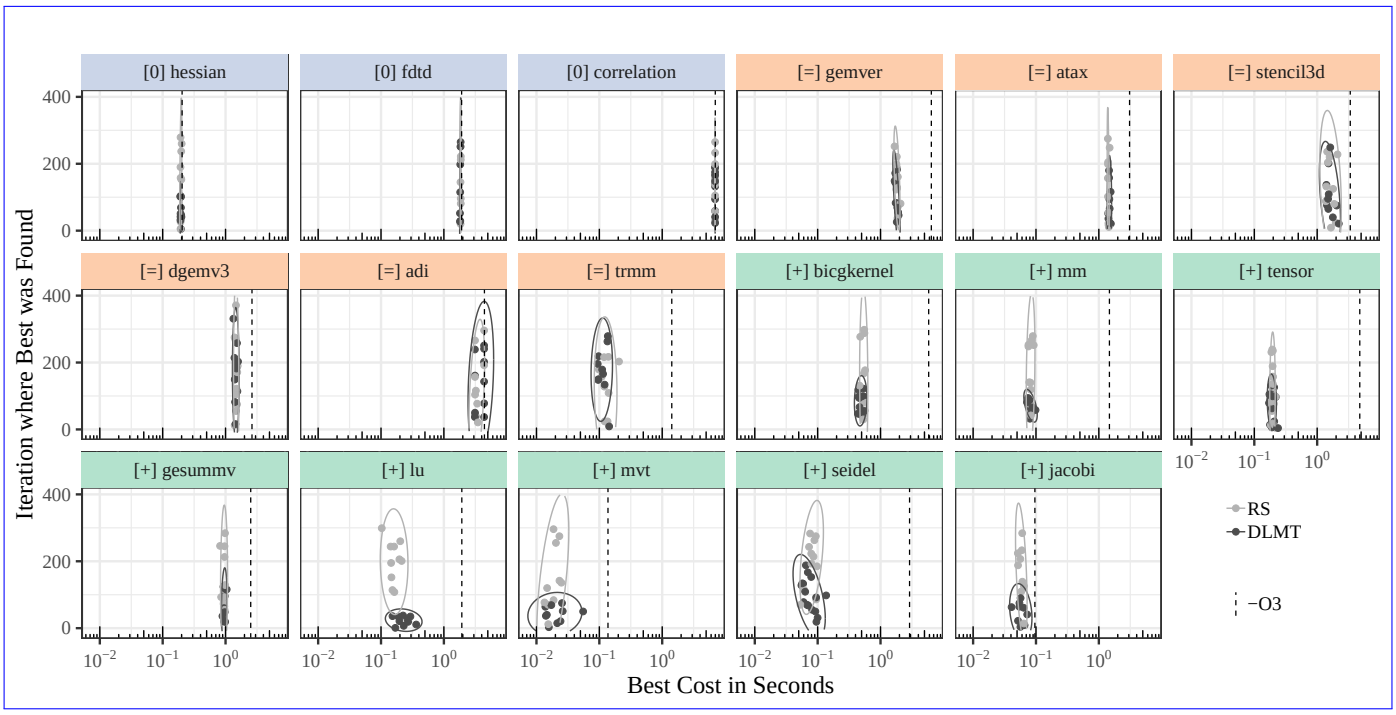

Figure 4: Cost of best points found on each run, and the iteration where they were found. RS and DLMT found no speedups with similar budgets for kernels marked with "[0]" and *blue* headers, and similar speedups with similar budgets for kernels marked with "[=]" and *orange* headers. DLMT found similar speedups using smaller budgets for kernels marked with "[+]" *green* headers. Ellipses delimit an estimate of where 95% of the underlying distribution lies

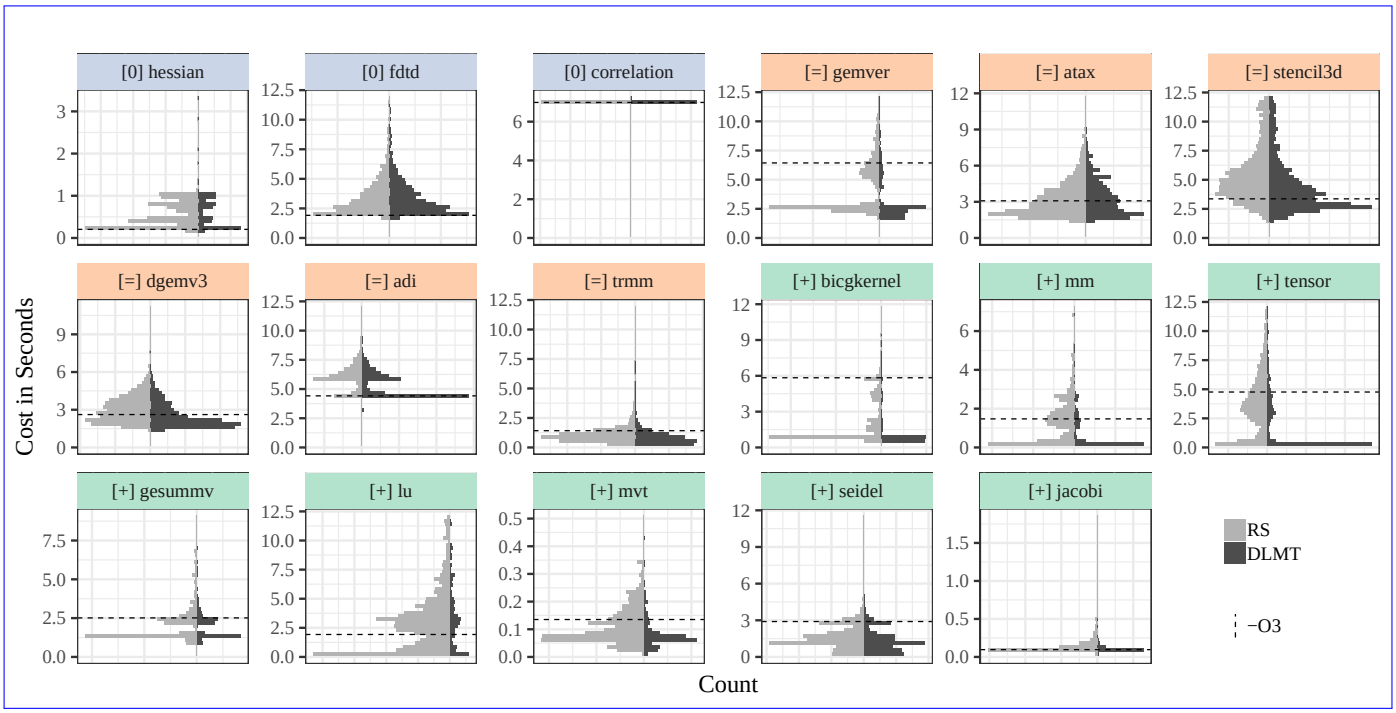

Figure 5: Histograms of explored search spaces, showing the real count of measured configurations. Kernels are grouped in the same way as in Figure 4. DLMT spent fewer measurements than RS in configurations with smaller speedups or with slowdowns, even for kernels in the orange group. DLMT also spent more time exploring configurations with larger speedups

~~Histograms of explored search spaces, showing the real count of measured configurations. Kernels are grouped in the same way as in Figure ??. DLMT spent less measurements than RS in configurations with smaller speedups or with slowdowns, even for kernels in the orange group. DLMT also spent more time exploring configurations with larger speedups~~

VI. Conclusion

~~We presented in this paper a transparent Design of Experiments~~ The contribution of this paper is a transparent DoE approach for program autotuning under tight budget constraints. We discussed the underlying concepts that enable our approach to ~~significantly reduce~~ reduce significantly the measurement budget needed to find good optimizations consistently over different kernels exposing configuration parameters of source-to-source transformations. We have made efforts to make our results, figures and analyses reproducible by hosting all our scripts and data publicly [33].

Our approach outperformed six other search heuristics, always finding a slowdown of 1% from the global optimum of the search space defined by the optimization of a Laplacian kernel for GPUs, while using at most half of the allotted budget. In a more comprehensive evaluation, using kernels from the SPAPT benchmark, our approach was able to find the same speedups as ~~random sampling~~ RS while using up to $10\times$ ~~less~~ fewer measurements. We showed that our approach explored search spaces more efficiently, even for kernels where it performed similarly to random sampling.

We presented a completely automated version of our approach in this paper so that we could perform a thorough ~~evaluation of its performance~~ performance evaluation on comprehensive benchmarks. Despite using the same generic performance model for all kernels, our approach was able to find good speedups by eliminating insignificant model terms at each iteration. This means that our approach can still improve the performance of applications using unspecialized models that incorporate only general knowledge about algorithm performance. We would incur some budget overhead in this case while insignificant terms are removed.

In future work we will explore the impact of user input and expert knowledge in the selection of the initial performance model and in the subsequent elimination of factors using ANOVA tests. We expect that tailored initial performance models and assisted factor elimination will improve the solutions found by our approach and decrease the budget needed to find them.

Our current strategy eliminates completely from the model the factors with low significance detected by ANOVA tests. In future work we will also explore the effect of adding random experiments with randomized factor levels. We expect this will decrease the impact of removing factors wrongly detected to have low significance.

Decreasing the number of experiments needed to find optimizations is a desirable property for autotuners in problem domains other than source-to-source transformation. We intend to evaluate the performance of our approach in domains such as High-Level Synthesis and compiler configuration for FPGAs, where search spaces can get as large as 10^{126}, and where we already have some experience [34].

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. This work was partly funded by CAPES, *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*, Brazil, funding code 001.

References

[1] K. Seymour, H. You, and J. Dongarra, "A comparison of search heuristics for empirical code optimization." in *CLUSTER*, 2008, pp. 421–429.

[2] P. M. Knijnenburg, T. Kisuki, and M. F. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," *The Journal of Supercomputing*, vol. 24, no. 1, pp. 43–67, 2003.

[3] P. Balaprakash, S. M. Wild, and P. D. Hovland, "Can search algorithms save large-scale automatic performance tuning?" in *ICCS*, 2011, pp. 2136–2145.

[4] ——, "An experimental study of global and local search algorithms in empirical performance tuning," in *International Conference on High Performance Computing for Computational Science*. Springer, 2012, pp. 261–269.

[5] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *Parallel & Distributed Processing, 2017. IPDPS 2017. IEEE International Symposium on*. IEEE, 2017, pp. 307–316.

[6] T. L. Falch and A. C. Elster, "Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 8, 2017.

[7] P. Balaprakash, A. Tiwari, S. M. Wild, and P. D. Hovland, "AutoMOMML: Automatic Multi-objective Modeling with Machine Learning," in *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, M. J. Kunkel, P. Balaji, and J. Dongarra, Eds. Springer International Publishing, 2016, pp. 219–239.

[8] P. Balaprakash, S. M. Wild, and B. Norris, "SPAPT: Search problems in automatic performance tuning," *Procedia Computer Science*, vol. 9, pp. 1959–1968, 2012.

[9] 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*. IEEE, 2009, pp. 1–11.

[10] B. Videau, K. Pouget, L. Genovese, T. Deutsch, D. Komatitsch, F. Desprez, and J.-F. Méhaut, "BOAST: A metaprogramming framework to produce portable and efficient computing kernels for hpc applications," *The International Journal of High Performance Computing Applications*, p. 1094342017718068, 2017.

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

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

[13] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice*. ACM, 2009, vol. 44, no. 6.

[14] J. R. Rice, "The algorithm selection problem," in *Advances in Computers 15*, 1976, pp. 65–118.

[15] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology," in *Proceedings of International Conference on Supercomputing, Vienna, Austria*, 1997.

[16] J. J. Dongarra and C. R. Whaley, "Automatically tuned linear algebra software (ATLAS)," *Proceedings of SC*, vol. 98, 1998.

[17] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[18] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.

[19] M. Gerndt and M. Ott, "Automatic performance analysis with Periscope," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 736–748, 2010.

[20] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12.

[21] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[22] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2014, pp. 303–316.

[23] R. Carnell, *lhs: Latin Hypercube Samples*, 2018, R package version 0.16. [Online]. Available: https://CRAN.R-project.org/package=lhs

[24] R. L. Plackett and J. P. Burman, "The design of optimum multifactorial experiments," *Biometrika*, vol. 33, no. 4, pp. 305–325, 1946.

[25] S. Addelman and O. Kempthorne, "Some main-effect plans and orthogonal arrays of strength two," *The Annals of Mathematical Statistics*, pp. 1167–1176, 1961.

[26] U. Grömping and R. Fontana, "An algorithm for generating good mixed level factorial designs," Beuth University of Applied Sciences, Berlin, Tech. Rep., 2018.

[27] J. Fox and S. Weisberg, *An R Companion to Applied Regression*, 2nd ed. Thousand Oaks CA: Sage, 2011. [Online]. Available: http://socserv.socsci.mcmaster.ca/jfox/Books/Companion

[28] U. Grömping, "R package FrF2 for creating and analyzing fractional factorial 2-level designs," *Journal of Statistical Software*, vol. 56, no. 1, pp. 1–56, 2014. [Online]. Available: http://www.jstatsoft.org/v56/i01/

[29] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2018. [Online]. Available: https://www.R-project.org/

[30] V. V. Fedorov, *Theory of optimal experiments*. Elsevier, 1972.

[31] B. Wheeler, *AlgDesign: Algorithmic Experimental Design*, 2014, R package version 1.1-7.3. [Online]. Available: https://CRAN.R-project.org/package=AlgDesign

[32] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[33] P. Bruel, "Git repository with all scripts and data," https://github.com/phrb/ccgrid19, accessed: 2018-10-14.

[34] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic, "Autotuning high-level synthesis for FPGAs using OpenTuner and LegUp," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2017.