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

Pedro Bruel*†, Arnaud Legrand*, Brice Videau*, Jean-Marc Vincent*, and Alfredo Goldman†
*University of Grenoble Alpes, CNRS, INRIA, Grenoble INP, LIG – 38000 Grenoble, France
Email: {arnaud.legrand, brice.videau, jean-marc.vincent}@imag.fr
†University of São Paulo – São Paulo, Brazil
Email: {phrb, gold}@ime.usp.br

***Abstract*—A large quantity of resources is spent writing, porting, and optimizing scientific and industrial High Performance Computing applications. Autotuning techniques have become therefore fundamental to lower the costs of leveraging the improvements on execution time and power consumption provided by the latest software and hardware platforms. Yet, most popular autotuning techniques still require a large budget of costly experimental measurements to provide good results while rarely providing exploitable knowledge about the problem after optimization. In this paper we present a user-transparent autotuning technique based on Design of Experiments that operates under tight budget constraints by significantly reducing the amount of measurements needed to find good optimizations. Our approach also enable users to make informed decisions on what optimizations to pursue and when to stop optimizing. We present experimental evaluations of our approach and show that, leveraging user decisions, it is capable of finding the global optimum of a GPU Laplacian kernel optimization using half of the measurement budget used by other common autotuning techniques. We also show that our approach is capable of finding speedups of up to $50\times$ for some kernels from the SPAPT benchmark using up to $10\times$ less 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. 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 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 single or multiple software of hardware metrics. The exploration of the search spaces defined by configurations and optimizations present interesting challenges to search strategies. These search spaces grow exponentially with the number of considered configuration parameters and their possible values. They are also difficult to extensively explore due to the often prohibitive costs of hardware utilization and 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 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 properties, but are generally incapable of fully exploiting global search space structures. Seymour *et al.* [1] and Knijnenburg *et al.* [2] 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 and frequent restarts to achieve good performance improvements. Search strategies based on gradient descent are also commonly used in autotuning and also rely on a large number of measurements. Their effectiveness diminishes significantly in search spaces with complex local structures. Completely automated machine learning autotuning strategies are promising in building models for predicting important optimization parameters, but still rely on a sizable data set for training. Large data sets are fundamental to strategies based on machine learning since they select models from a generally very large class.

ref

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 was explored with unknown biases.

In this paper we propose an autotuning strategy that leverages existing expert and approximate knowledge about a problem in the form of a performance model, and refines this initial model iteratively using empirical performance evaluations, statistical analysis and user input. Our strategy puts a heavy weight on decreasing the costs of autotuning by using efficient *Design of Experiments* strategies to minimize the number of

experiments needed to find good optimizations. Each optimization iteration uses *Analysis of Variance* (ANOVA) tests and *linear model regressions* to identify promising subspaces and the relative significance of each configurable parameter to the performance observations. An architecture- and problem-specific performance model is built iteratively and with user input, which enables making informed decisions on which regions of the search space are worth exploring.

We present the performance of our approach on a Laplacian Kernel for GPUs where the search space, global optimum and performance model approximation are known. The experimental budget on this kernel was tightly constrained. The speedups achieved and the budget utilization of our approach on this setting motivated a more comprehensive performance evaluation. We chose the *Search Problems in Automatic Performance Tuning* (SPAPT) [3] benchmark for this evaluation, where we obtained diverse results. Of the 17 SPAPT kernels benchmarked, we found no speedups for 3 kernels, and random sampling was very effective for 7 others. Speedups were harder to find by sampling for the remaining 8 kernels, where our approach was able to find speedups of up to $50\times$ while using up to $10\times$ less measurements than random sampling, despite using the same generic performance model for every kernel.

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 problems, on autotuning systems and on search space exploration strategies. Section III discusses Design of Experiments concepts and the ANOVA and linear regression methodology we use. Section IV presents a detailed description of the implementation of our approach. Section V presents our results with the GPU Laplacian Kernel and the SPAPT benchmark. Section VI discusses our conclusions and future work.

II. Background

A. Source-to-source transformation

B. Search Space Exploration Strategies

C. Autotuning

John Rice's Algorithm Selection framework [4] is the precursor of autotuners in various problem domains. In 1997, the PHiPAC system [5] 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.* [6] introduced the ATLAS project, that optimizes dense matrix multiplication routines. The OSKI [7] library provides automatically tuned kernels for sparse matrices. The FFTW [8] library provides tuned C subroutines for computing the Discrete Fourier Transform. Periscope [9] 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 [10] implements abstractions for OpenMP, MPI and OpenCL, and

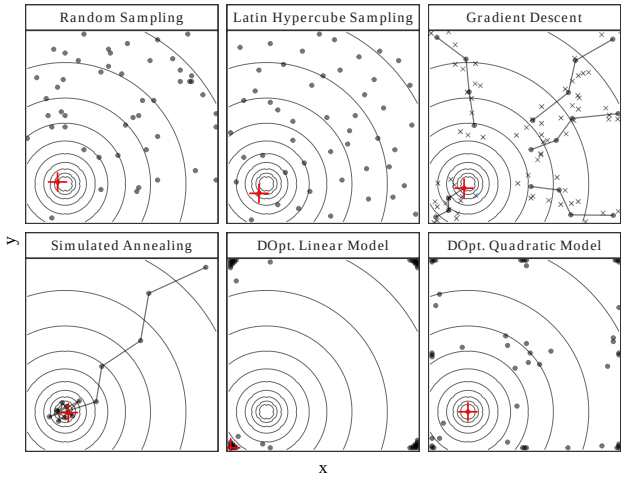

Figure 1: Exploration of the search space defined by $x^2 + y^2$, using a fixed budget of 50 points. The "+" represents the best point found by each strategy

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 [11] project provides a language, compiler and autotuner, enabling the definition and selection of multiple algorithms for the same problem. The ParamILS framework [12] applies stochastic local search algorithms to algorithm configuration and parameter tuning. The OpenTuner framework [13] provides ensembles of techniques that search the same space in parallel, while exploration is managed by a multi-armed bandit strategy.

III. 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 *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 first present the assumptions of a traditional Design of Experiments 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 this paper.

A. Screening & Plackett-Burman Designs

Screening designs provide a parsimonious way to identify 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 [14] in 1946, and is available in the `FrF2` package [15] of the `R` language [16].

Despite having strong restrictions on the number of factors they support, 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. For the following example, consider the linear relationship presented in Equation (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 can think of that intuitively by noting that less relevant factors will have corresponding values in $\boldsymbol{\beta}$ close to zero.

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

We now present an example to illustrate the screening methodology. Suppose we wish to minimize a performance metric Y of a problem with factors $x_1, \ldots, x_8$ assuming values in $\{-1, -0.8, -0.6, \ldots, 0.6, 0.8, 1\}$. Each $y_i \in Y$ is computed using Equation (2), but suppose that, for the purpose of this example, they are computed by a very expensive black-box procedure. Note that not all factors are included in the real computation. In this scenario we can think of the error term ϵ as representing not only noise but our uncertainty regarding the model as well. Higher amplitudes of ϵ might make it harder to justify isolating factors with low significance.

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

To efficiently study this problem we decide to construct a Plackett-Burman design, which minimizes the experiments needed to identify relevant factors. The analysis of this design will enable us to decrease 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 3 "dummy" factors $d_1, \ldots, d_3$ to complete the 12 columns needed to construct a Plackett-Burman design for 8 factors.

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

We use our initial assumption shown in Equation (1) to identify the most relevant factors by performing an ANOVA test. The resulting ANOVA table is shown in Table II, where the *significance* of each factor can be interpreted from the F-test and P($<$ 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 model for now, 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) values we decide that they are fairly smaller than the values of factors that had no significance at all, and we keep this factor.

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	$**$

Moving forward, 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 to predict the levels for 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.

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

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.

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.

B. D-Optimal Designs

The application of Design of Experiments 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 Plackett-Burman design for our example 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. 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 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 with experiments from a pool of 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, we use Fedorov's algorithm [17] for constructing D-Optimal designs, implemented in `R` in the `AlgDesign` package [18].

Going back to 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 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 -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.

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	-4.881
1.0	1.0	-1.0	-1.0	-1.0	1.133
-1.0	-1.0	1.0	1.0	-1.0	4.609
-1.0	1.0	1.0	1.0	-1.0	3.231
-1.0	-1.0	-1.0	-1.0	0.0	0.055
1.0	1.0	1.0	-1.0	0.0	7.132
-1.0	1.0	-1.0	1.0	0.0	-3.305
1.0	-1.0	1.0	1.0	0.0	-2.187
-1.0	-1.0	1.0	-1.0	1.0	7.854
-1.0	1.0	1.0	-1.0	1.0	7.336
1.0	-1.0	-1.0	1.0	1.0	-8.205
1.0	1.0	-1.0	1.0	1.0	-2.003

We are now going to 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 approach can achieve close model estimations using few resources, provided are able to use user input to identify relevant factors and knowledge about the problem domain to tweak the model.

Table VI compares the global minimum in 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 computing Y only 24 times. We were able to do that by first

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	0.424	2.384	0.076	.
x_1	-1.500	-1.287	-11.825	0.000	***
x_3	1.300	1.357	13.218	0.000	***
x_5	3.100	3.336	30.646	0.000	***
x_7	-1.400	-1.646	-15.124	0.000	***
x_8	0.000	0.111	0.886	0.426	
x_8^2	1.350	0.710	3.263	0.031	*
$x_1 x_3$	1.600	1.684	15.468	0.000	***

reducing the dimension of the problem when we eliminated irrelevant factors in the screening step. We then constructed a more careful exploration of this new problem subspace, helped by domain knowledge provided by an expert.

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

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.

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.

IV. 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 2 presents an overview of our approach.

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 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 numerical ordering. Factors like these are named *categorical*, and must be treated differently when constructing designs and analyzing the results.

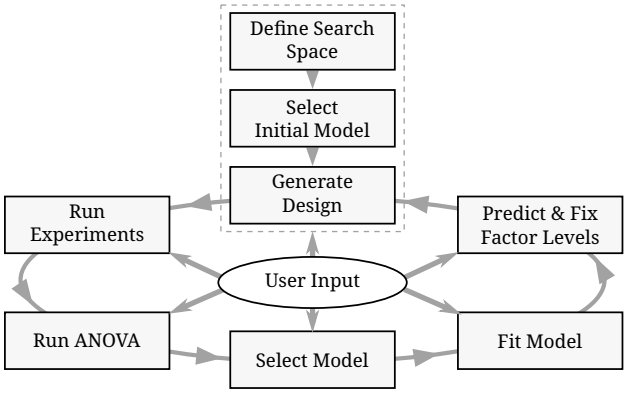

Figure 2: Overview of the Design of Experiments approach to autotuning proposed in this paper

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 a 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 [14] screening designs shown in the previous Section, the *contractive replacement* technique of Addelman-Kempthorne [19] and the *direct generation* algorithm by Grömping and Fontana [20]. 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 2, 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 but the user might still want to include them in the model to explore more of its levels, for example. 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 a quadratic term, for example.

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 [21] 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. In the next Section we present the performance of our approach in scenarios that differ on search space size, availability and complexity.

V. Performance Evaluation

In this Section we present performance evaluations of our approach in two scenarios.

A. GPU Laplacian Kernel

We first evaluated the performance of our approach in a Laplacian Kernel implemented using BOAST [22] 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
`vector_length`	$2^0, \ldots, 2^4$
`load_overlap`	*true*, *false*
`temporary_size`	$2, 4$
`elements_number`	$1, \ldots, 24$
`y_component_number`	$1, \ldots, 6$
`threads_number`	$2^5, \ldots, 2^{10}$
`lws_y`	$2^0, \ldots, 2^{10}$

We applied domain knowledge to construct the initial performance model shown in Equation (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 identical to our approach, described Section IV, except for the design generation step, where it uses a fixed-size random sample of the search space instead of generating designs. We compared the performance of our approach with the algorithms listed in Table VIII, using a budget of *at most* 125 measurements and 1000 repetitions.

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

Table VIII: Search algorithms compared in the GPU Laplacian Kernel

	Algorithm
RS	Random Sampling
LHS	Latin Hyper Square Sampling
GS	Greedy Search
GSR	Greedy Search w/ Restart
GA	Genetic Algorithm
LM	Iterative Linear Model
DLMT	D-Optimal Designs

Table IX 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 for the slowdowns found by each of the 1000 repetitions. Arrows point the maximum slowdown found by each algorithm.

All algorithms performed well in this kernel, with only Greedy Search (GS) not being able to find slowdowns smaller

Table IX: 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

than $4\times$ in some runs. As expected, other search algorithms had results similar to Random Sampling (RS). The LM algorithm was able to find the global minimum on most runs, but some runs found slowdowns of almost $4\times$. Our approach was able to find the global minimum in all of the 1000 runs while using *at most* less than half of the allotted budget.

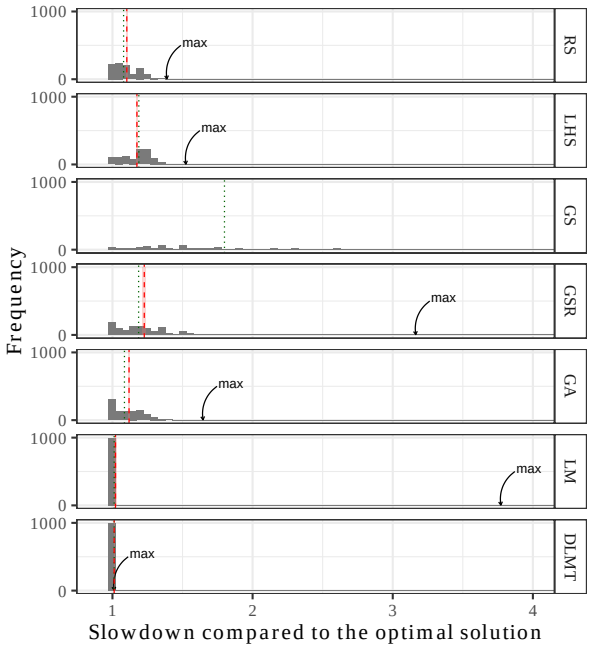

Figure 3: Histograms of 7 optimization methods on the Laplacian Kernel, using a budget of 125 points with 1000 repetitions

This kernel provides 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. Next we will analyze the performance of our approach in a larger and more comprehensive scenario.

B. SPAPT Benchmark

The SPAPT [3] benchmark provides parametrized kernels from different High Performance Computing domains. The kernels, shown in Table X, are implemented using the code annotation and transformation tools provided by Orio [23]. Search space sizes are overall larger than in the Laplacian Kernel example. Kernel factors are either integers in a range, such as loop unrolling and register tiling amounts, or binary flags that control parallelization and vectorization, for example.

We used the Random Sampling (RS) implementation available in Orio and implemented our approach (DLMT) in the system.

After the model is selected and fitted, prediction results will depend on the size of the data set available. If it is feasible to compute the fitted model on all possible factor combinations, we can be sure that the global optimum has a chance of being found. If the search space is too large to be generated, we have to adapt this step and run the prediction on a sample.

Table X: Set of kernels we used from the SPAPT benchmark

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}

VI. Conclusion

Acknowledgment

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 B. Norris, "Spapt: Search problems in automatic performance tuning," *Procedia Computer Science*, vol. 9, pp. 1959–1968, 2012.

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

[5] 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.

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

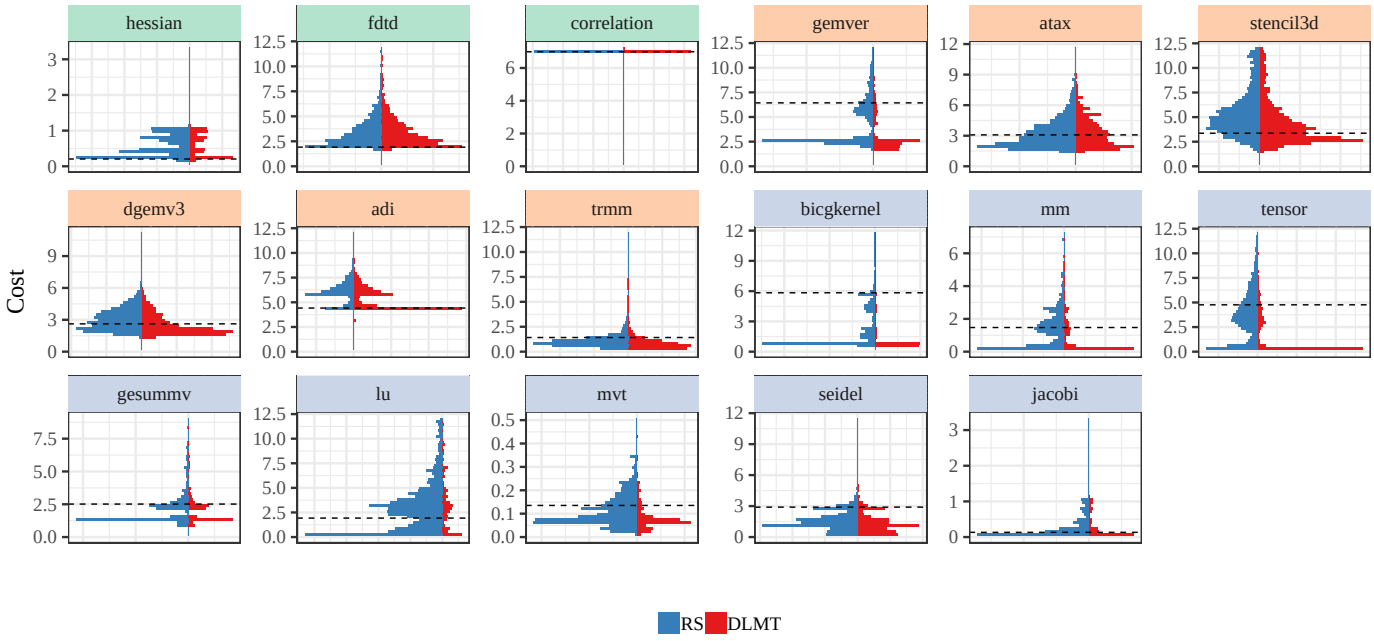

Figure 4: Histograms of explored search spaces

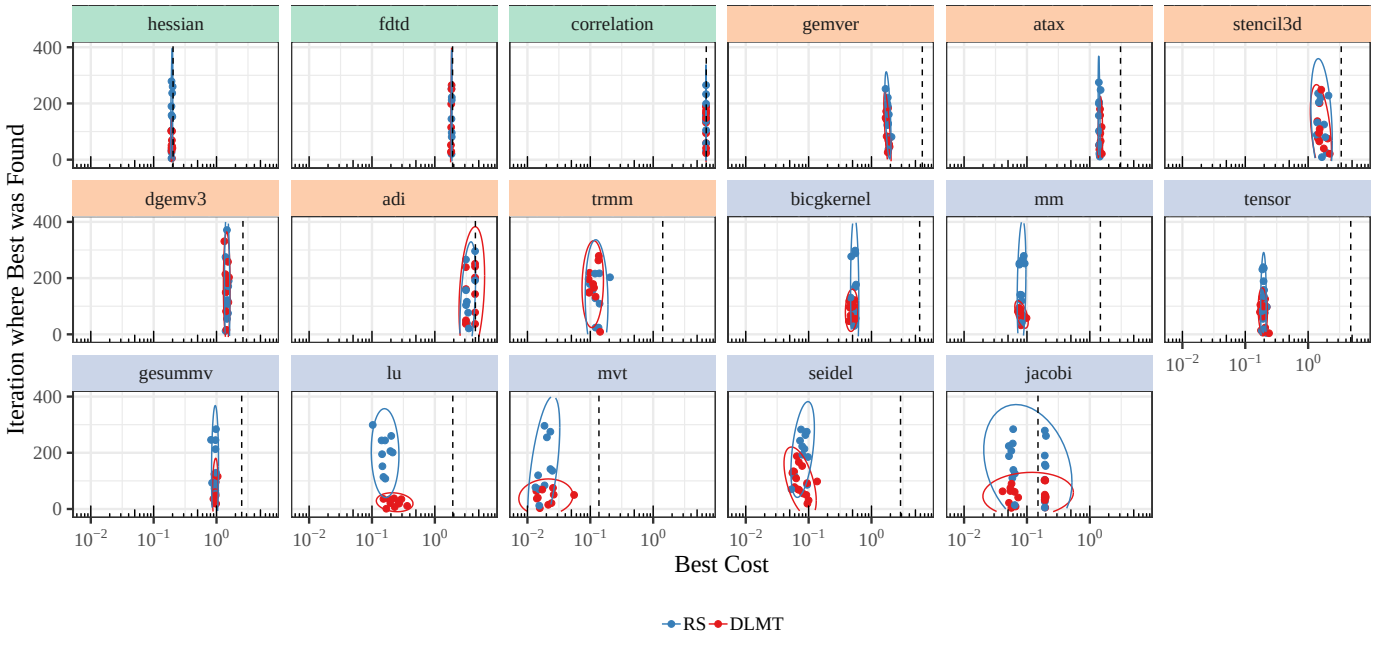

Figure 5: Cost of best point found on each run against the iteration where it was found

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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*. ACM, 2014, pp. 303–316.

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

[15] U. Gr"omping, "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/

[16] 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/

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

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

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

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

[21] 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

[22] 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.

[23] 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.