Exploiting historical data: pruning autotuning spaces and estimating the number of tuning steps

Jaroslav Oľha, Jana Hozzová, Jan Fousek, Jiří Filipovič

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Exploiting historical data: pruning autotuning spaces and estimating the number of tuning steps

Jaroslav Oľha[1,2] | Jana Hozzová[1] | Jan Fousek[1] | Jiří Filipovič[1]

[1]Institute of Computer Science, Masaryk University, Brno, Czech Republic
[2]Faculty of Informatics, Masaryk University, Brno, Czech Republic

Correspondence
Jaroslav Oľha
Email: olha@ics.muni.cz

Present Address
Jaroslav Oľha
Faculty of Informatics,
Masaryk University
Botanická 68A,
602 00 Brno, Czechia

Summary

Autotuning, the practice of automatic tuning of applications to provide performance portability, has received increased attention in the research community, especially in high performance computing. Ensuring high performance on a variety of hardware usually means modifications to the code, often via different values of a selected set of parameters, such as tiling size, loop unrolling factor or data layout. However, the search space of all possible combinations of these parameters can be large, which can result in cases where the benefits of autotuning are outweighed by its cost, especially with dynamic tuning. Therefore, estimating the tuning time in advance or shortening the tuning time is very important in dynamic tuning applications.

We have found that certain properties of tuning spaces do not vary much when hardware is changed. In this paper, we demonstrate that it is possible to use historical data to reliably predict the number of tuning steps that is necessary to find a well-performing configuration, and to reduce the size of the tuning space. We evaluate our hypotheses on a number of HPC benchmarks written in CUDA and OpenCL, using several different generations of GPUs and CPUs.

KEYWORDS:
Autotuning, prediction of tuning cost, tuning space pruning, sensitivity analysis

1 | INTRODUCTION

With ever-changing hardware architectures, it is difficult and costly to keep applications performing well on a wide range of hardware – in order to retain high performance, the implementation needs to be modified to adapt to a new execution environment.

In well-written code, it is often sufficient to change the values of a few pre-selected parameters, such as block size or loop unrolling factor. Since the manual search for the right combination of parameter values can be tedious and error-prone, an automatic method, called *autotuning*, has been developed to search the space of possible implementations and find the best one (a comprehensive survey of autotuning has been compiled by Balaprakash et al.[1]). This search can be performed before the application is launched (offline tuning), or at runtime, switching implementations on-the-fly whenever a faster configuration is found (dynamic tuning)[1] – the latter approach being particularly useful, for instance, in cases where changing the characteristics of the input affects optimization choices.

However, the tuning spaces of many problems are difficult to navigate[2,3,4] – discrete values of parameters influence each other in a non-linear way, and the tuning spaces have low locality (two similar configurations can perform very differently). As a result, finding a well-performing configuration of tuning parameters is a challenging task even with sophisticated search methods.

To improve this process, we propose guiding the search using information obtained from previous tuning of the same computational problem on different hardware. We analyze the spaces of tuning parameters and search for properties which do not change significantly across different hardware devices, with the intent of using them to make tuning predictions or modifications. We show two different (but not mutually exclusive) ways this information can be applied to improve autotuning.

Firstly, we can **estimate how many tuning iterations are needed** to achieve reasonable performance, which is essential in order to make certain tuning decisions. When tuning time is limited, such as in the case of dynamic tuning, the question of *tuning amortization* becomes important – in other words, whether the benefit of tuning outweighs its cost. Ignoring this aspect can result in cases where even if an autotuner improves the performance of a program, the tuning process itself ends up consuming more time and resources than was saved by the optimization. To prevent this, it is necessary to be able to estimate the number of tuning steps that a given search method will need for optimization.

Given the nature of the tuning spaces we use for testing (see the Benchmarks section), we chose random search as our preferred method of searching the tuning space – this search strategy also gives us the additional benefit of simplicity and predictability, making it a good fit for our future work in scheduling (see Section 6 – Conclusion and Future Work). In this paper, we demonstrate that the portion of tuning space composed of well-performing configurations remains similar for a given problem across different hardware for a majority of cases, which means that the number of tuning steps for random search will also be similar.

Secondly, we can **prune the tuning space**, helping an autotuner find a well-performing implementation more quickly. We propose that certain tuning parameters are more significant than others when it comes to performance, depending on the application, and their significance is portable across hardware. As a result, we are able to remove insignificant parameters and thus reduce the dimensionality of the tuning space without losing well-performing configurations. Unlike the method for predicting tuning time, this pruning approach is not bound to a specific search strategy. Although we demonstrate that we are able to shrink tuning spaces using the methods presented in this paper, which improves the speed of exhaustive tuning, non-exhaustive searchers do not seem to gain any significant performance benefit on the pruned tuning space.

Both of these approaches are model-free, using static historical data. While this type of work could benefit greatly from dynamic profiling and performance modeling, this is outside of the scope of this paper – we plan to focus on using performance models in the future (see Section 6 – Conclusion and Future Work).

In our current work, we focus on tuning spaces composed by expert programmers, which tend to be small (each tuning parameter only has a few possible values) and discontinuous (e. g. blocking size values of 2, 4, 8, 16 and 32 instead of all natural numbers between 1 and 32). These types of search spaces often pose a problem for traditional search methods relying in part on local search (such as gradient descent, simulated annealing or various genetic algorithms), which often perform similarly to simple random search[5,6,3]. Moreover, random search provides several advantages compared to more sophisticated search methods – not only is it easier to predict its performance based solely on the shape of the tuning space, it has also been shown to be very robust – whereas many search methods' performance is highly dependent on the tuned kernel or starting configuration (performing very well in some situations while failing in others), random search tends to maintain predictably good performance regardless of these factors[7]. Additionally, when it is impractical to use exhaustive search of the tuning space as a basis for historic information, random search is easy to apply on a smaller sample of the tuning space without significant deterioration in performance.

For all of these reasons, we apply our proposed improvements primarily using random search. Nevertheless, they are applicable to other search methods as well, as we discuss later in the paper.

To evaluate our methods, we use a comprehensive set of ten autotuned benchmarks[8], executed on five generations of GPU accelerators and four generations of CPUs.

We use the following terminology in the paper. A *tuning parameter* is a variable which affects the code in a user-defined way (e. g. determines loop unroll factor). The *tuning space* is a search space composed of all the possible values of all tuning parameters – each parameter has a set of possible values, and the tuning space is the Cartesian product of these sets. A *configuration* is a single point in the tuning space, which fully determines one possible implementation of the tuned code.

The rest of the paper is organized as follows. The overview of the benchmark set and the used hardware is given in Section 2. Our methods are described and evaluated in Section 3 (prediction of the portion of tuning space which needs to be searched) and Section 4 (pruning the tuning space). A comparison with related work is given in Section 5. We conclude and outline future research in Section 6.

2 | BENCHMARK SET

To show that our proposed hypotheses are not problem- or application-specific, we have used a rather wide set of benchmark problems implemented in a way that enables autotuning [8]. In this section, we briefly introduce the benchmarks and hardware that we have used for evaluation.

2.1 | Benchmarks

TABLE 1 A list of the benchmarks, including the size and dimensionality of their tuning spaces (i.e. how many tuning parameters they have).

Benchmark name	Configurations	Dimensions
BiCG	5 122	11
2D Convolution	3 928	10
3D Coulomb Sum	1 260	8
GEMM	241 600	14
GEMM Batched	424	8
Hotspot	480	6
Matrix Transpose	5 916	10
N-body	9 408	8
Reduction	175	5
3D Fourier Reconstruction[†]	430	8

Table 1 shows a basic overview of the benchmarks, including the size and dimensionality of their tuning spaces. A more detailed description of these benchmarks, including their implementation, tuning parameters and performance characteristics, can be found in Petrovič et al. [8]. Note that our tuning spaces are relatively small – however, this does not limit the code's ability to reach near-optimal performance – in Petrovič et al. [8], it is shown that the benchmarks are capable of reaching near-theoretical performance peaks on a variety of different GPU hardware.

The benchmarks are composed of important computational kernels spanning across multiple application domains: 3D Fourier reconstruction [9] and 2D convolution (adopted from CLTune [3]) are image processing kernels, BiCG, GEMM (adopted from CLTune [3]), GEMM batched, Matrix transpose and Reduction [10] are linear algebra kernels, Direct Coulomb summation [10] is a computational chemistry kernel, N-body (autotuned version of NVIDIA CUDA SDK sample) and Hotspot (based on implementation from Rodinia benchmark [11]) are differential equation solvers. These benchmarks autotune a variety of tuning parameters, changing implementation properties such as work-group size, cache blocking, thread coarsening, explicit caching in local memory, loop unrolling, explicit vectorization or data layout optimization (i. e. array of structures vs. structure of arrays). These tuning parameters were determined by expert programmers during development – we did not add or remove any tuning parameters afterwards, e. g. after the analysis of parameters' importance.

We believe that this type of benchmark design simulates the use-case that would benefit most from our approach – a programmer who is aware of the critical optimizations in their code, and can determine the sensible values for the given optimization (e.g. a blocking factor of 32, 64 or 128 rather than every natural number between 1 and 1024), but requires more dynamic performance portability between different hardware setups.

As a result of the parameters being designed by expert programmers, most of these search spaces are relatively small compared to some other autotuning works [12,13] – the number of tuning parameters is not needlessly excessive, and most parameters only have a few rationally-selected values to choose from. While such spaces are hard to search using methods that partially rely on local search (such as simulated annealing), because most of their dimensions are very small and discontinuous, they also contain a higher proportion of relatively fast configurations, which makes random search a viable strategy.

[†]The 3D Fourier Reconstruction benchmark only has a CUDA implementation, and is therefore only evaluated on GPU; all the other benchmarks are implemented in OpenCL and work on both CPU and GPU.

All the benchmarks have been evaluated on sufficiently large inputs, so that the available parallelism (especially on GPU) was utilized. The 3D Fourier benchmark processes a large number of small images and its performance is highly sensitive to the size of the images. Therefore, we have autotuned 3D Fourier Reconstruction for multiple image sizes: 32×32, 64×64 and 128×128, referred to as Fourier (32), Fourier (64) and Fourier (128) in the following text. The GEMM Batched benchmark performs batched matrix multiplication of small matrices – we have measured the performance on 16×16 matrices. All benchmarks use single-precision arithmetic.

The code-path of the kernels used in this work is not dependent on the content of input data (it only depends on the volume of input data). Therefore, the benchmarks' performance is dependent only on data size (the size of matrices in GEMM Batch, or the size of images in 3D Fourier Reconstruction).

Additionally, all of the benchmarks have been designed to run on a single node, not a cluster – in this work and others, we focus on node-level acceleration (even for a single laptop), rather than performance within a large networked environment.

The 3D Fourier reconstruction[9] is available in Xmipp software[‡], while the rest of the benchmarks are available in Kernel Tuning Toolkit as examples installed with the tuner[§]. The Kernel Tuning Toolkit[8] has been used to obtain the results for this paper.

2.2 | Hardware

We have evaluated all of the benchmarks on five GPUs of different architectures and performance, see Table 2.

TABLE 2 A list of the GPUs used in our tests.

GPU	Architecture	SP Performance	Bandwidth	Released	Drivers
AMD Radeon RX Vega 56	GCN 5	8,286 GFlops	410 GB/s	2017	2833.0
NVIDIA Tesla K20	Kepler	3,524 GFlops	208 GB/s	2012	418.40
NVIDIA GeForce GTX 750	Maxwell	1,044 GFlops	80 GB/s	2014	410.48
NVIDIA GeForce GTX 1070	Pascal	5,783 GFlops	256 GB/s	2016	418.67
NVIDIA GeForce RTX 2080Ti	Turing	11,750 GFlops	616 GB/s	2018	418.39

All but one of the benchmarks (see above) have also been evaluated on four different CPUs.

TABLE 3 A list of the CPUs used in our tests.

GPU	Architecture	SP Performance	Bandwidth	Released	Drivers
[Dual] Intel Xeon E5-2650	Sandy Bridge EP	512 GFlops	102 GB/s	2012	1.2.0.8
Intel Core i7-3820	Sandy Bridge E	230 GFlops	51.2 GB/s	2012	1.2.0.37
Intel Core i7-4790	Haswell	461 GFlops	25.6 GB/s	2014	18.1.0.0920
Intel Core i7-8700	Coffee Lake	307 GFlops	41.6 GB/s	2017	18.1.0.0920

In our choice of hardware, we were not only limited by the availability of certain kinds of processors, but also by their OpenCL support. This is consistent with the intended applicability of our methods in real circumstances – if a processing unit doesn't support OpenCL, it would require a separate kernel implementation with somewhat different tuning parameters, defeating the purpose of correlating tuning spaces.

Additionally, we do not use data from GPU to make assumptions about CPU, and vice versa. Even though OpenCL allows kernels to run on both types of processing units, the behavior and performance characteristics are too different, and reconciling these differences is outside of the scope of this paper – the same would apply for any other types of accelerators (TPU, Xeon Phi, ...).

[‡] https://github.com/I2PC/xmipp
[§] https://github.com/Fillo7/KTT/tree/master/examples

3 | ESTIMATING THE NUMBER OF TUNING STEPS

In theory, autotuning can help maintain a program's performance on new hardware or with changed input characteristics. However, to achieve that, the performance improvement (compared to the non-tuned version) must outweigh the tuning time – the tuning process needs to be amortized.

Whereas in some scenarios the tuning time is amortized in almost any case (e. g. long execution on supercomputers), in other scenarios the tuning time matters (e. g. if an application is not supposed to use many CPU hours, if it migrates to different models of computing devices, or if tuning decisions are sensitive to input data which are changing at runtime). Therefore, knowing how long it will take to find a well-performing configuration is vital in order to decide if autotuning is worthwhile. We define a well-performing configuration as a tuning configuration producing an implementation of sufficient performance – in this paper, we will consider "sufficient performance" to be 90% of the performance of the fastest implementation available within tuning space, but performance requirements are completely use case dependent and this number can be redefined by the user.

To predict tuning time, we need to know the number of tuning steps required to find a well-performing configuration, and the average time of a tuning step. Additionally, the amortization of tuning time is determined by how many times we plan to execute the tuned kernel and how much speedup is achieved by tuning. In this paper, we target the first question: predicting the number of steps required to search a tuning space.

We can do this by analyzing the tuning space on certain hardware, inferring the number of tuning steps needed to find a well-performing configuration, and using this data to predict the number of tuning steps on different hardware.

In this regard, random search provides a great advantage – we can calculate the probability of finding a configuration with a certain performance within a given number of tuning steps, without having to run the searcher. For example, if we know that 5 % of our search space contains configurations that perform above 90 % of the optimum, we can directly calculate how many tuning steps we need in order to get a 95 % chance of finding such a configuration (see Formula 2). All we need to do this is a performance histogram of configurations in the search space.

3.1 | Prediction method

Application parameter autotuning allows tuning parameters to be translated into virtually any property of the source code, from changing loop unrolling factors to selecting an entirely different algorithm. Therefore, tuning spaces can vary in their size and also in the effect of each tuning parameter. Whereas some tuning spaces may contain a high number of configurations with near-optimal performance [5], other tuning spaces may contain only a few well-performing configurations [3,4].

The performance distribution among all the various configurations in our benchmark set is shown in Figure 1. Here, the benchmarks' tuning spaces are shown using violin plots – histograms of computation times are plotted on the y-axis (the wider the graph, the more configurations fall within the given performance range), with the best-performing configurations at the bottom. The histograms vary significantly for different computational problems – for example, many implementations of the Batched GEMM benchmark have very good performance, whereas the Hotspot benchmark only has a few fast implementations. Therefore, estimating the number of tuning steps required to reach sufficient performance is not straightforward.

We hypothesize that the number of well-performing configurations remains similar for a computation problem across different hardware devices. The intuition behind this hypothesis is as follows. For a given processor, autotuning needs to balance many tuning variables. Some of them are very critical and must be set to an optimal value, while others may have a wider interval of well-performing values. When we change the processor, the optimal values may be shifted (e. g. by adding more cache, the optimal cache blocking factor may change), but the required precision of selection will be similar. Of course, hardware development may change the number of well-performing values – for example, adding more registers to the processor may lead to a wider range of efficient loop unrolling factors. However, those changes are not expected to be drastic, as hardware development is limited by the manufacturing process. In contrast, changes of tuning parameters may have a much higher impact on the shape of the tuning space and hence the relative amount of well-performing configurations. Therefore, the amount of well-performing configurations should be more stable with respect to the changing hardware than to the changing computational problems.

If the portion of well-performing configurations remains stable across different hardware – and therefore it can be predicted using historical data – we can use this to estimate the number of configurations we need to evaluate in order to find a good one.

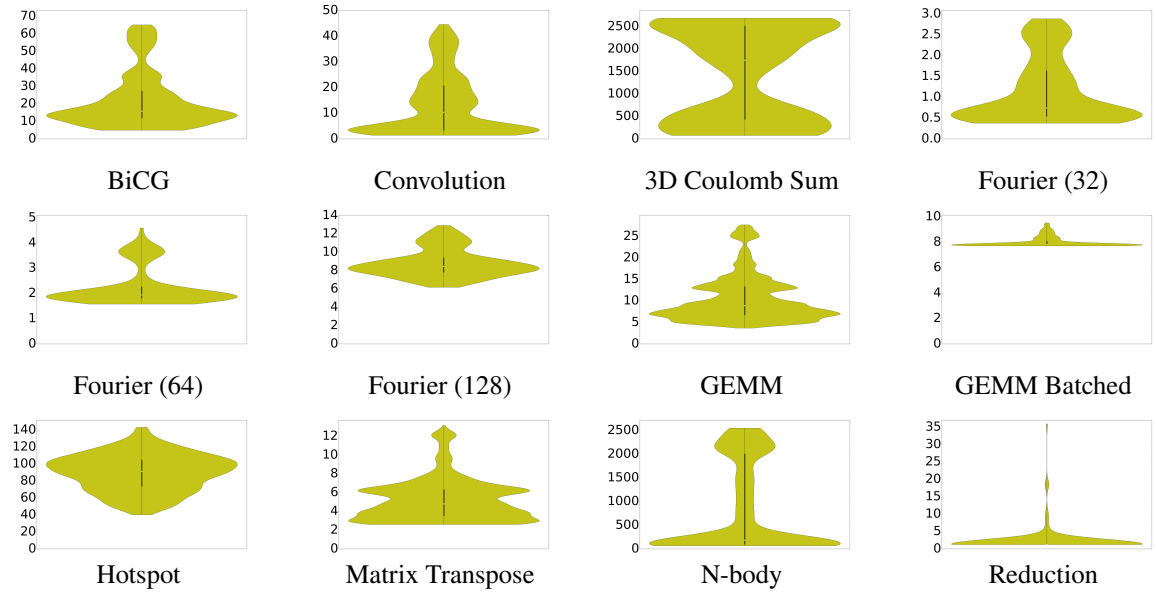

FIGURE 1 Violin plots of computation times for various benchmarks, measured on GeForce GTX 1070. The X-axis shows the amount of configurations, the Y-axis shows computation time (in ms).

When random search is used, we can easily infer Formula 1 – let p_b be the portion of the tuning space that is well-performing, and p_s be the desired probability of finding a well-performing configuration. Then the relationship between them is defined as

$$(1 - p_s) = (1 - p_b)^{n_{conf}} \tag{1}$$

where n_{conf} is the number of random attempts needed to reach a certain probability of finding a good-enough configuration. Therefore, to determine the number of attempts (i. e. the number of autotuning steps) required, we only need to calculate the corresponding logarithm using Formula 2:

$$n_{conf} = log_{(1-p_b)}(1 - p_s) \tag{2}$$

For example, if historical data show us that well-performing configurations comprise 1 % of the tuning space, then in order to have a 90 % chance of reaching a good solution, we need to explore $log_{(1-0.01)}(1 - 0.9) \approx 230$ configurations.

3.2 | Evaluation

To support our hypothesis, we have prepared the following experiment. We consider a configuration well-performing if it achieves at least 90 % of the performance of the best configuration. We have executed an exhaustive search on all available combinations of benchmarks and hardware (all GPUs as well as all CPUs). The results are shown in Table 4 and Table 5. For each combination of hardware and benchmark, the table shows the portion of the tuning space comprised of well-performing configurations (e. g. 0.05 means that 5 % of all configurations are well-performing)[¶]. Naturally, if the search spaces were too large for exhaustive search, we could perform the same process with random sampling, which would give us approximate results – note, however, that this assumes we are using random search. More sophisticated search methods could be significantly handicapped on a randomly sampled search space.

It can be seen in the tables that the variation of well-performing configurations is much smaller across hardware architectures than across different benchmarks. The differences across one family of hardware (e. g. GPUs) are not in orders of magnitude,

[¶]Note that some numbers are missing in the tables: the 3D Fourier Reconstruction is implemented in CUDA and is therefore not measured on Radeon Vega56 and all CPUs. It is also not measured on Nvidia Tesla K20, because we were unable to install Xmipp on the system. Some benchmarks have been executed with a smaller tuning space on Radeon Vega56, because AMD ROCm driver (OpenCL driver version 2833.0) crashed with some tuning configurations (mainly using vectors of size 16 and higher loop unrolling factors). Therefore, we have omitted those benchmarks as their tuning spaces differ significantly. On CPU i7-8700, the value of GEMM is missing, because the benchmark always caused a reboot on this particular hardware (with OpenCL driver version 18.1.0.0920), likely due to a driver bug

TABLE 4 The portion of each tuning space consisting of well-performing configurations (i. e. reaching 90 % of the best performance) on GPU. The average portion with standard deviation for each problem is shown in the last column.

	Vega56	K20	750	1070	2080Ti	avg ± stddev
BiCG	0.00459	0.00527	0.00332	0.0168	0.0181	0.00962 ± 0.0072
Convolution	0.00280	0.00321	0.00117	0.00204	0.00175	0.00274 ± 0.00082
3D Coulomb Sum	N/A	0.0603	0.0405	0.0389	0.0476	0.0468 ± 0.0098
Fourier (32)	N/A	N/A	0.0190	0.0286	0.0357	0.0278 ± 0.0084
Fourier (64)	N/A	N/A	0.0595	0.119	0.0357	0.0714 ± 0.043
Fourier (128)	N/A	N/A	0.0167	0.0571	0.0119	0.0286 ± 0.025
GEMM	N/A	0.000791	0.00107	0.00231	0.00793	0.00302 ± 0.0033
GEMM Batched	0.120	0.0943	0.151	0.818	0.642	0.365 ± 0.34
Hotspot	0.0169	0.00743	0.00495	0.00495	0.0149	0.00983 ± 0.0057
Matrix Transpose	0.0210	0.0461	0.0413	0.150	0.0101	0.0492 ± 0.052
N-body	N/A	0.0207	0.0576	0.0277	0.0492	0.0388 ± 0.017
Reduction	0.0775	0.327	0.463	0.715	0.368	0.390 ± 0.23

TABLE 5 The portion of each tuning space consisting of well-performing configurations (i. e. reaching 90 % of the best performance) on CPU. The average portion with standard deviation for each problem is shown in the last column.

	dual-E5-2650	i7-3820	i7-4790	i7-8700	avg ± stddev
BiCG	0.00203	0.00968	0.00156	0.00234	0.00391 ± 0.00387
Convolution	0.0145	0.00672	0.00152	0.00305	0.00644 ± 0.00577
3D Coulomb Sum	0.163	0.405	0.187	0.198	0.238 ± 0.112
GEMM	0.000765	0.000885	0.000331	N/A	0.000660 ± 0.000291
GEMM Batched	0.0125	0.0125	0.403	0.480	0.227 ± 0.250
Hotspot	0.0112	0.0179	0.0112	0.0156	0.0140 ± 0.00335
Matrix Transpose	0.0152	0.0319	0.0235	0.0128	0.0209 ± 0.00868
N-body	0.0231	0.111	0.0359	0.0119	0.0455 ± 0.0448
Reduction	0.0314	0.0307	0.145	0.130	0.0843 ± 0.0618

excepting some outliers (GEMM on 2080Ti, Matrix transpose on 1070 and Reduction on Vega56). Therefore, when the portion of the tuning space containing well-performing implementations is known for at least one hardware device, we can use it to predict the number of tuning steps on other devices. Surprisingly, the results are less consistent on CPU, where in most cases the standard deviation is nearly as high as the average value.

Note, however, that the approach is still valid even in cases of several-fold misprediction, because the scale of the error is much lower than if we made random assumptions, or if we made the predictions based on historical data from different benchmarks. For example, if we tried to guess the number of well-performing configurations on a GPU using data from different benchmarks on the same architecture (i. e. using the *columns* of the table rather than the *rows*), the difference in the portion of the tuning space containing well-performing configurations could reach two orders of magnitude or more.

3.3 | Using more sophisticated search methods

A similar approach could also be applied to more complex search strategies, but with a crucial variation – rather than using the shape of the tuning space (i. e. the number of well-performing configurations) from one hardware to make predictions about tuning on different hardware, we can instead use the number of tuning steps. In this scenario, we could launch a searcher 100 times, and observe how many times the searcher reached a certain performance in N tuning steps. This number would be equivalent to the probability of finding a configuration of certain performance in N tuning steps.

However, this requires an extra step – whereas with random search, knowing the shape of the search space translates directly into the probability of finding a good configuration in a given number of tuning steps (Formula 1), with a more sophisticated

search strategy the relationship is typically much more complicated. In such a case, instead of simply storing a histogram of the search space, we would need to store the searcher results for every possible combination of performance requirement, probability and number of steps, because we couldn't calculate its expected performance directly. Since our end-goal is dynamic optimization of tuning time vs. performance improvement, this would be a very impractical obstacle in realtime applications.

For this reason, we only use the results of random search for evaluation in this section.

4 | TUNING SPACE PRUNING

Autotuning of a computational kernel can take a long time if the tuning space is large and full of low-performing configurations. Pruning of the tuning space can accelerate the search. In previous research, authors proposed a method based on the assumption that biasing search towards configurations which perform well on one processor can speed up the search on another processor [14]. The method works well if the relative performance of the tuned codes correlates across different hardware (at least for the well-performing configurations). Often, this seems to be the correct assumption. However, we have found cases where the correlation is not good – this is demonstrated in Figure 2, where well-performing configurations have high correlation for Matrix transposition, but low correlation for the Fourier (128) benchmark.

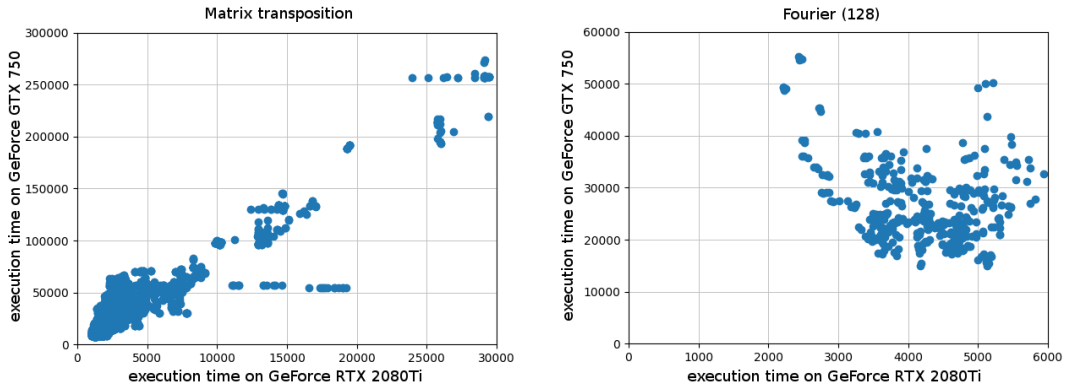

FIGURE 2 The correlation between kernel runtime on GeForce GTX 750 and GeForce RTX 2080Ti for the Matrix transposition benchmark (left) and the Fourier (128) benchmark (right) in milliseconds. Each dot represents a configuration.

4.1 | Proposed pruning method

Here, we propose an alternative method of tuning space pruning, which is based on the importance of tuning parameters. Even though the optimal *values* for parameters change from hardware to hardware, we hypothesize that the *significance* of parameters (how much they influence the resulting performance) remains more stable. For example, if a problem is cache-sensitive on one GPU, it will probably remain cache-sensitive on any other GPU, even though the exact value of optimal cache blocking size will change.

Note that even though our benchmarks' tuning parameters have been chosen by expert programmers (see the Benchmarks section), with emphasis on avoiding dummy parameters that have little impact on performance, this is not at odds with the idea of pruning unimportant parameters. One of the reasons for this is that different hardware architectures will differ in which parameters are important, which means, for instance, that a parameter that seems insignificant on a GPU will often prove to be important on CPU and vice versa. If we want to avoid maintaining a separate implementation of the kernel for each type of hardware, this allows us to have a single implementation with all the tuning parameters, which are then only pruned in the context of the given hardware set. Another reason for the presence of low-impact parameters is that even for an expert programmer, it is often impossible to know which parameters will have the highest impact, until the code can be tested on a multitude of hardware setups.

When the tuning space is evaluated for a hardware device, we can compute mutual information[#] between tuning variables and runtime.

Figure 3 shows the distributions of these mutual information values across all GPUs from our test set, and Figure 4 does the same for all tested CPUs. Since it would be impractical (and uninformative) to list the name of each parameter, we divided them into 10 categories, and the figures (in combination with Table 6) show which category the given parameter falls into[‖]. For more information, please refer to Petrovič et al.[8], where detailed descriptions of the benchmarks, including their tuning parameters, can be found.

Both boxplot figures seem to confirm our initial assumption that the mutual information (and therefore the significance of a given parameter for overall runtime) tends to remain stable regardless of the particular processor.

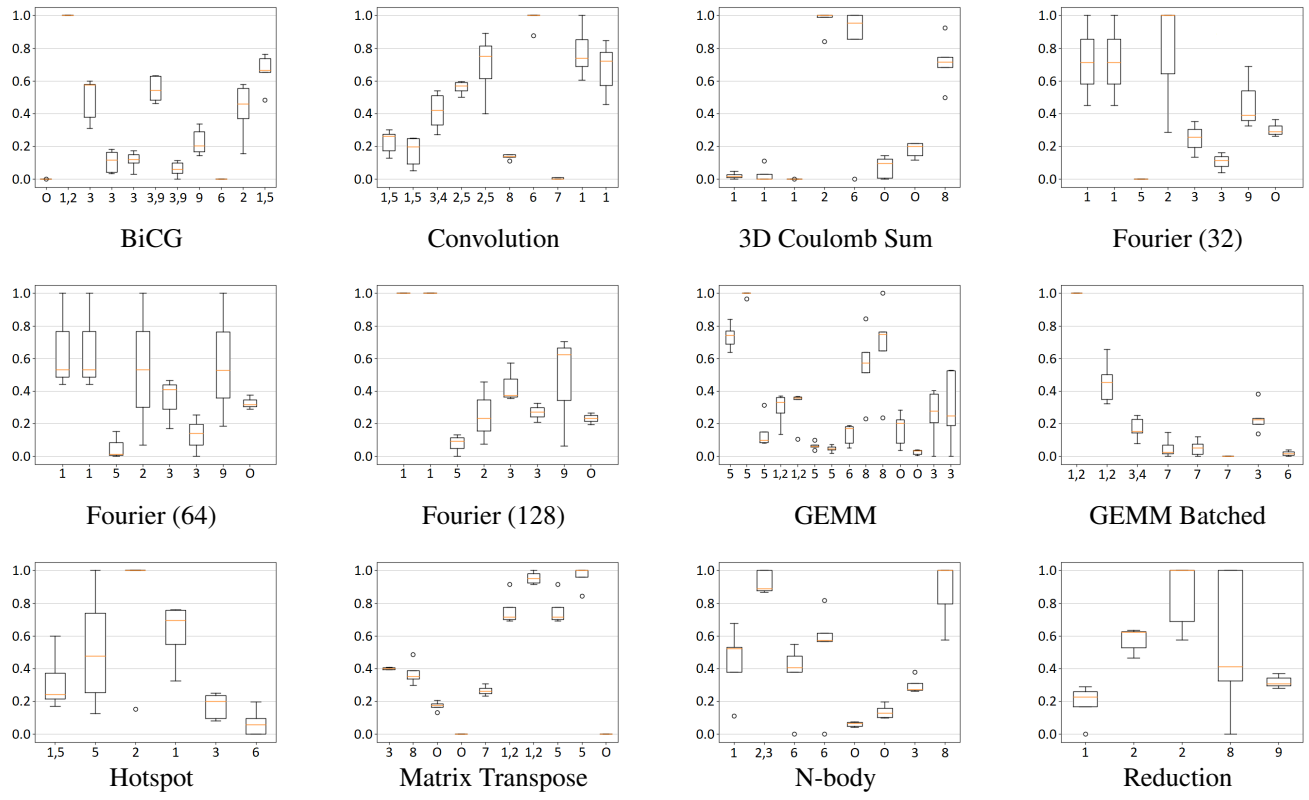

FIGURE 3 Boxplot distributions of the mutual information values of each parameter in each benchmark across different GPUs. Each box represents one parameter of the given benchmark – the X-axis shows the type of the given parameter (see Table 6 for a list of parameter types), while the Y-axis shows the relative mutual information between the tuning parameter and runtime. In most cases, the distributions are quite narrow, suggesting that the significance of a given parameter for overall performance is similar across different hardware.

Our proposal is to prune the less significant parameters from the tuning space by fixing their values on a median and not changing them during the search. Since every parameter represents one dimension of the search space, removing the insignificant parameters will reduce the number of dimensions that need to be searched, while keeping the well-performing configurations. It should be noted that our pruning approach is not designed as a brand new search method – rather, it can be used to improve the performance of already existing search algorithms.

[#] a sensitivity analysis metric which measures the dependency between variables – higher values mean higher dependency [15]

[‖] A parameter may simultaneously fall into two categories, e. g. by controlling both thread blocking and cache blocking. It is also common for multiple parameters within a benchmark to belong to the same type (e. g. two parameters determining two different dimensions of thread blocking).

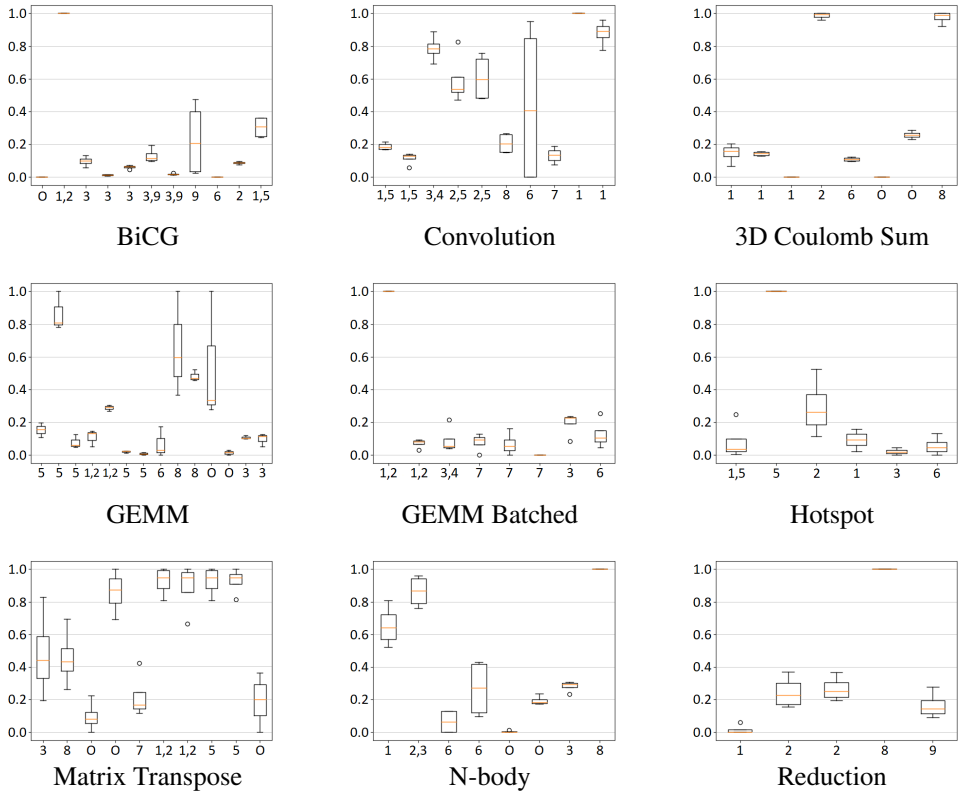

FIGURE 4 Boxplot distributions of the mutual information values of each parameter in each benchmark across different CPUs. Each box represents one parameter of the given benchmark – the X-axis shows the type of the given parameter (see Table 6 for a list of parameter types), while the Y-axis shows the relative mutual information between the tuning parameter and runtime. In most cases, the distributions are quite narrow, suggesting that the significance of a given parameter for overall performance is similar across different hardware.

TABLE 6 A list of parameter types within the tuning benchmarks, relating to Figures 3 and 4.

1	Workgroup size	6	Unrolling
2	Coarsening	7	Local memory padding
3	Local memory caching	8	Vectorization
4	Private memory caching	9	Atomics
5	Tile size	O	Other

We have devised two ways of choosing which parameters to prune based on their mutual information with runtime. Remember that in both cases, we are basing the selection of parameters on data that we already have from tuning the given problem on one hardware, and we plan to use the pruned space to autotune the same problem on different hardware. Although we have used exhaustive search to decide which parameters should be pruned, in cases where searching the entire tuning space is unfeasible (due to its size or the time it takes to test each configuration), random sampling can be used.

The first approach is simply to choose a cutoff point based on the parameter with the highest mutual information (MI_{max}) – all the parameters whose mutual information value is lower than a certain percentage of MI_{max} (in our case 20 %) are pruned. For the purposes of this paper, we will call this the *naive* pruning method. We have already evaluated this approach in our previous work [16], and while it worked very well for most problems, there was a case where too many parameters were pruned and the well-performing configurations were lost. This was, however, not caused by poor correlation in parameter significance between

different hardware generations – even on the hardware that was used to select the parameters, the space was overly pruned. We argued that this could be fixed by a slightly more sophisticated approach to parameter selection.

The second approach is to sort the parameters based on their mutual information with runtime, and iteratively prune them one by one, starting with the least-influential one. After each parameter is removed, we compare the best configuration in the pruned search space with the best configuration in the original search space. We repeat this process as long as the best performance in the pruned space stays within a certain performance threshold (90 % of the original optimum, in our case). This method solves the problem encountered with the previous approach, but in some cases it can create new problems by overpruning different spaces for different reasons. This will be explored further in the following section, where we will refer to this approach as the *aggressive* pruning method.

As a result of both approaches having a tendency to overly prune the search spaces in a few special cases, we have also tested a conservative combination of these two approaches – a parameter is pruned only if it meets both criteria: its mutual information is sufficiently low (relative to the highest mutual information in the benchmark), *and* pruning it will not degrade the best available performance beyond a certain threshold (relative to the best performance in the non-pruned space). In the following section, this combined approach will be referred to as the *conservative* pruning method.

Regardless of the pruning method, the resulting search space will not be more dense or more sparse than the original one, since we are pruning entire *dimensions* of the search space, rather than individual low-performing configurations. Additionally, there should not be a significant change in the proportion of fast and slow configurations, since we are mostly trying to reduce the number of configurations with similar performance. Referring back to the violin plots from Figure 1, the pruned tuning spaces would merely be "narrower", with a similar distribution of well-performing and poorly-performing configurations, but with a lower overall number of configurations.

4.2 | Evaluation of pruning space methods

To evaluate the proposed pruning methods, we first computed mutual information for data measured on GPU-1070 and CPU-i7-4790**. Then, we pruned the tuning spaces using the aggressive and the conservative pruning methods, resulting in four sets of results in Tables 7, 8, 9 and 10.

The tables show state space reduction (SSR)††, the number of pruned parameters, and performance retention (PR) in all the benchmarks after the less significant parameters have been pruned (e. g. SSR of 16 means that the state space has been reduced 16-fold, and PR of 0.92 means that the best configuration in the pruned tuning space performs at 92 % of the best performance reachable from the original tuning space).

TABLE 7 State space reduction (SSR), the number of pruned/all parameters (PP) and performance retention for all tested hardware/benchmark combinations, after **aggressive pruning** of parameters with low mutual information **on GPU**.

	SSR	PP	K20	Vega56	1070	750	2080Ti
BiCG	511.7	9/11	0.93	0.86	0.99	0.96	0.95
Convolution	2.75	1/10	0.99	1.0	1.0	1.0	0.85
3D Coulomb Sum	12.0	5/8	0.97	1.0	0.96	0.97	0.94
Fourier (32)	4.0	6/9	N/A	N/A	0.98	1.0	0.99
Fourier (64)	420.0	8/9	N/A	N/A	0.94	0.37	0.82
Fourier (128)	12.0	5/9	N/A	N/A	1.0	0.73	1.0
GEMM	12 080.0	12/15	0.56	0.91	0.96	0.79	0.91
GEMM Batched	141.33	10/11	0.91	0.76	1.0	0.98	0.99
Hotspot	6.97	2/6	0.98	0.83	1.0	1.0	1.0
Matrix Transpose	189.21	8/10	0.88	0.8	1.0	0.93	0.91
N-body	137.61	5/8	0.88	1.0	0.93	1.0	0.72
Reduction	27.5	3/5	0.98	0.96	0.99	1.0	0.99

**This mimics the situation when the developer has knowledge of a tuning space on that GPU/CPU only.

††State space reduction can simply be calculated as the total number of configurations in the original search space divided by the number of configurations in the pruned search space.

TABLE 8 State space reduction (SSR), the number of pruned/all parameters (PP) and performance retention for all tested hardware/benchmark combinations, after **conservative pruning** of parameters with low mutual information **on GPU**.

	SSR	PP	Vega56	K20	750	1070	2080Ti
BiCG	16.34	5/11	0.92	0.99	0.97	0.99	1.0
Convolution	2.75	1/10	1.0	0.99	1.0	1.0	0.85
3D Coulomb Sum	10.0	4/8	0.99	1.0	0.97	0.97	0.97
Fourier (32)	4.0	3/7	N/A	N/A	1.0	0.98	0.99
Fourier (64)	4.0	3/7	N/A	N/A	1.0	0.98	1.0
Fourier (128)	4.0	3/7	N/A	N/A	1.0	0.99	0.99
GEMM	47.19	6/15	0.97	0.92	0.90	1.0	0.97
GEMM Batched	22.32	8/11	0.96	0.92	0.99	1.0	0.99
Hotspot	6.97	2/6	0.83	0.98	1.0	1.0	0.99
Matrix Transpose	6.0	3/10	1.0	1.0	1.0	1.0	1.0
N-body	2.22	2/8	1.0	1.0	1.0	0.98	1.0
Reduction	1.0	0/5	1.0	1.0	1.0	1.0	1.0

TABLE 9 State space reduction (SSR), the number of pruned/all parameters (PP) and performance retention for all tested hardware/benchmark combinations, after **aggressive pruning** of parameters with low mutual information **on CPU**.

	SSR	PP	dual-E5-2650	i7-3820	i7-4790	i7-8700
BiCG	2.0	2/11	1.0	1.0	1.0	1.0
Convolution	4.55	1/10	0.97	0.97	0.93	0.9
3D Coulomb Sum	252.0	7/8	0.91	1.0	0.93	0.97
GEMM	94.38	7/15	0.92	0.94	0.95	N/A
GEMM Batched	133.33	10/11	0.81	0.81	0.94	0.96
Hotspot	2.11	1/6	0.9	0.97	0.96	0.95
Matrix Transpose	35.21	6/10	0.91	0.96	1.0	0.9
N-body	37.15	4/8	0.89	1.0	1.0	0.92
Reduction	5.89	2/5	0.75	1.0	1.0	1.0

As we can see in the tables, both methods are able to significantly reduce the size of the state space, while sacrificing only a few percent of performance in most cases.

On GPU, the two pruning strategies usually result in very different search spaces – in fact, the results of the conservative pruning strategy are very similar to the results of the naive pruning method from our previous work [16], since in most cases, the parameters were pruned based on low mutual information rather than performance degradation.

The aggressive pruning method results in severely reduced search spaces in the majority of cases (most notably in the GEMM benchmark), but it seems to be more sensitive to weak correlation between parameter significance – on benchmarks such as Fourier (64), where the distribution of mutual information values across different hardware is wide (see Figure 3), the space can be overly pruned.

Note the Reduction benchmark, which was not pruned at all by the conservative pruning method, but the aggressive method pruned it by a significant factor while barely decreasing the best performance. This suggests that the mutual information values of all parameters are too similar for any one of them to be pruned, even though they all have a low impact on the overall runtime.

In the case of CPUs, the difference between aggressive and conservative pruning approaches is much less significant, and the performance degradation caused by the aggressive pruning method is usually very low compared to the conservative approach.

The pruned space can be used in two scenarios. Firstly, we can execute exhaustive search on it, reducing the number of tuning iterations by a factor of SSR (see Tables 7, 8, 9 and 10), reaching good performance with a lower number of tuning iterations. Compared to a non-exhaustive search on an unpruned space, this provides the benefit of stable and repeatable results (i. e. it will

TABLE 10 State space reduction (SSR), the number of pruned/all parameters (PP) and performance retention for all tested hardware/benchmark combinations, after **conservative pruning** of parameters with low mutual information **on CPU**.

	SSR	PP	dual-E5-2650	i7-3820	i7-4790	i7-8700
BiCG	2.0	2/11	1.0	1.0	1.0	1.0
Convolution	4.55	1/10	0.97	0.97	0.93	0.9
3D Coulomb Sum	70.0	5/8	0.99	1.0	1.0	1.0
GEMM	94.38	7/15	0.92	0.94	0.95	N/A
GEMM Batched	66.67	9/11	0.81	0.81	0.94	0.96
Hotspot	2.11	1/6	0.9	0.97	0.96	0.95
Matrix Transpose	3.48	3/10	0.99	1.0	1.0	1.0
N-body	14.0	2/8	1.0	1.0	1.0	0.98
Reduction	4.97	1/5	0.77	1.0	1.0	1.0

never get "unlucky" runs where it would miss all the good configurations or get stuck in a local optimum). Secondly, we can search iteratively within the pruned space. We elaborate on the second use case in the following section.

4.3 | Convergence in pruned spaces

Naturally, an iterative approach will inevitably reach better performance on the full space once the majority of configurations have been searched, because it may contain optimal configurations which cannot be reached in the pruned space. However, we also wish to study whether the search will converge faster to a well-performing configuration (for instance, 90 % of the optimum) on the pruned space with fewer dimensions.

We expect the speed of convergence to be highly dependent on the chosen search method – in principle, any search algorithm can be run on the pruned space, but the benefits of this pruning approach may vary significantly depending on the search strategy used.

As part of our experiment, we have executed autotuning with random search as well as simulated annealing[‡‡] on both the full and the pruned space 1000× to get statistically significant results for each benchmark[§§] – all the results (i. e. the best performance found after N tuning steps for each of the 1000 tuning runs) were then averaged for each kernel using simple arithmetic mean.

We observe different rates of convergence towards good solutions between the full and the pruned spaces – see Figure 5 for examples (both from random search). While in the Matrix Transpose benchmark, the convergence is faster in the pruned space, in case of the Hotspot problem, the search converges faster in the full space of configurations.

To get results on the complete problem set, we have measured how quickly the tuning process converges to 90 % of the best performance for all combinations of search methods, processing units and problems. We have decided to only use the conservatively-pruned search spaces, since the aggressively-pruned spaces are intentionally pruned to only have a few best-performing values near 90 % of the optimum, and would therefore converge very slowly towards this target.

A brief summary of our results is shown in Table 11, which shows that even though pruning improves the convergence towards a good solution in a slight majority of the cases, the results are far from decisive.

When using random search on GPU, the pruning tends to improve the speed of convergence towards good performance, but it does not work universally. The fact that the pruning method has different impact on various searchers is apparent – in the case of simulated annealing, the pruning is detrimental to the speed of convergence for a majority of benchmarks.

On CPU, it is also evident that this approach does not provide the desired improvement in convergence speed – pruning decreases the convergence rate in almost half of the cases.

[‡‡] All the parameters of simulated annealing, such as the initial temperature and the cooling schedule, were adopted from the default CLTune implementation [3], with the computation time of kernels serving as the objective function for optimization. No additional constraints were added, except for the ones posed implicitly by the availability of configurations in the given search space. For both the simulated annealing and random search simulations, the pseudo-random numbers were generated by the C++ Standard Library's random function with uniform distribution of values.

[§§] We have used the rapid testing of the search method implemented in Kernel Tuning Toolkit – first, all of the configurations are executed and their performance data are gathered, then during searcher testing the autotuner reads measured times instead of performing empirical tuning. This allows for performing many experiments in reasonable time.

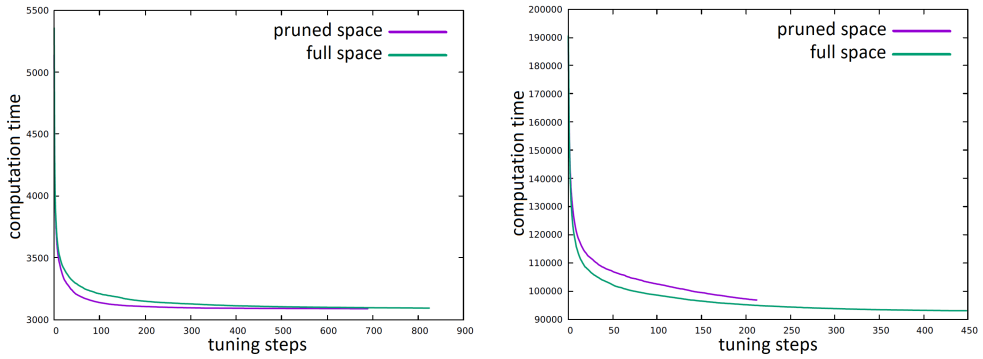

FIGURE 5 Random search convergence on full and pruned tuning spaces. The X-axis shows the number of tuning steps, the Y-axis shows the computation time of the fastest solution found so far. Both examples are the results of tuning on CPU-4790.

TABLE 11 The portion of benchmarks where conservative parameter pruning improves convergence (i. e. fewer tuning steps are required to reach 90 % of the best performance in the pruned space compared to the full space).

	Random Search	Simulated Annealing
GPU	71 %	38 %
CPU	55 %	57 %

5 | RELATED WORK

Numerous autotuning frameworks allow for the tuning of implementation parameters for heterogeneous computing [2,3,10,17,4]. All of these tuners are able to autotune OpenCL or CUDA code by altering their implementations, but the papers evaluate tuning spaces on rather limited benchmark sets. In this paper, we have used more comprehensive benchmarks [8].

To the best of our knowledge, there is no previous work attempting to predict the number of tuning search steps. Therefore, we limit the comparison of our work to methods improving tuning space search.

The question of parameter tuning is not exclusive to the field of kernel/hardware optimization – various heuristic algorithms are similarly controlled by sets of parameters which impact the algorithms' ability to find a good solution quickly. The resulting search spaces can be very large, and the stochastic nature of these heuristic algorithms often makes it difficult to compare various parameter configurations, requiring the use of sophisticated meta-heuristic approaches such as CRS-Tuning [18] or F-Race [19].

Search methods in the field of kernel autotuning can be *model-based* or *model-free*. Considering model-free methods, most autotuning papers show that random search performs similarly to more sophisticated search methods, such as simulated annealing or particle swarm optimization [5,6,3]. A promising improvement of model-free search has been introduced recently by van Werkhoven [4], outperforming other local and global search methods in most cases. Tuning space pruning introduced in this paper can be combined with any model-free search method.

Model-based methods attempt to take advantage of existing knowledge of the tuned system to predict the performance of a given implementation. Analytical solutions, which construct a mathematical model of performance, are specialized to a particular problem domain. A more scalable approach is to leverage empirical data, either from previous tuning runs or from concurrent profiling, to guide the tuning process.

Methods based on machine learning use historical data to build a performance model. Price and McIntosh-Smith built regression trees from an already explored part of the tuning space to steer a search method towards exploring more promising configurations [20]. However, no historical data from previous tuning runs on different hardware has been used. Data from previous runs are used for learning by Muralidharan et al. [21] and Cummins et al. [22] Those papers focus on dynamic selection from a very limited number of code variants [21], or optimization of a single tuning parameter at compilation time [22]. We are focusing on the usage of historical data in more complex tuning spaces.

Several works in the field of compiler autotuning have successfully tackled both the problem of optimization prediction [23] and the problem of pruning tuning spaces [13]. However, these compiler-oriented works focus on an entirely different aspect of performance optimization than our code-level autotuning framework, and use different methods, such as optimization clustering, to reduce their search spaces – it is therefore impossible to compare the results of these works with our own results.

Probably the closest method to our work uses historical data to prune tuning space or bias search towards a configuration which performs better on another hardware device [14]. The method is based on the assumption that configurations which perform well on one device also perform well on another (see Section 4 for a deeper discussion). In contrast, our method assumes that tuning space dimensions which have a low impact on performance on one device will also have a low impact on another.

6 | CONCLUSION AND FUTURE WORK

In this paper, we have introduced methods using historical data gained from previous tuning of the same problem on different hardware, allowing us to **predict the number of search steps necessary for tuning** and **prune the tuning space**.

In both of these use-cases, we assume relatively small, rationally-designed search spaces, which we search ahead of time to discover the amount of well-performing configurations and infer the importance of various tuning parameters.

As we discuss in the paper, for the first part of our work – prediction of the number of tuning steps – we chose to only predict the number of tuning steps required for random search. Although a modification could be made that would allow for predicting the number of tuning steps for other search methods, these are less suitable for our future work, wherein we intend to use the prediction method for dynamic scheduling of tuning in a runtime system. In other words, we intend to use the predictions to make decisions about whether the performance improvement achieved by autotuning a problem will outweigh the tuning cost – for this purpose, the robustness and predictability of random search is ideal (while still providing good performance).

Our results confirm our initial hypothesis that there is considerable correlation between tuning spaces on different hardware, which allows us to use static historical data to make tuning decisions on new hardware when using random search to find well-performing configurations. Although in this paper, we rely on knowing good configurations ahead of time (through exhaustive search or sampling), this could be improved by using profiling and performance modeling, which would not only allow us to make more accurate predictions with less preliminary data, but also to react dynamically to changing workload and other factors.

We also proposed several ways to prune search spaces using historical data from previous runs on different hardware, and we showed that in most cases, these can considerably reduce the dimensionality of the tuning space without losing well-performing configurations. This means that an exhaustive search of the tuning space would be much faster, while producing a solution which loses little to no performance compared to the best solution in the full tuning space.

We also hypothesized that search within the pruned spaces might converge more quickly towards relatively fast solutions, compared to the original, unpruned space. However, after testing this hypothesis using two different search strategies – random search and simulated annealing – the benefit of pruning to convergence proved negligible (and in many cases, actively counterproductive), so the main strength of this pruning approach remains in the exhaustive search of the pruned space.

We have used a set of ten benchmarks to test the usability of the proposed methods – the wide variety of tuning parameters within all of these benchmarks serve to demonstrate that the results are not restricted to a certain type of tuning parameters or to a narrow set of kernels with specific performance characteristics. The benchmark set is available to the community together with the Kernel Tuning Toolkit.

Aside from the future work outlined above, we also plan to further analyze the tuning spaces. We plan to categorize tuning parameters and study their importance for particular types of hardware, as well as their interactions. We also plan to test more search methods and more advanced pruning (e. g. using profiling counters) in order to speed up the tuning process.

Another possible research direction would be to use the historical data from benchmarks to build a machine learning model which might actually *predict* the correct optimizations for a new set of hardware, or at least to guide the search of the configuration space instead of using random search.

ACKNOWLEDGEMENTS

The work was supported from European Regional Development Fund-Project "CERIT Scientific Cloud" (No. CZ.02.1.01/0.0/0.0/16_013/0001802). Access to the CERIT-SC computing and storage facilities provided by the CERIT-SC Center, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CERIT Scientific Cloud LM2015085), is greatly appreciated.

References

1. Balaprakash P, Dongarra J, Gamblin T, et al. Autotuning in high-performance computing applications. In: *Proceedings of the IEEE*. 106. IEEE; 2018: 2068–2083.
2. Ansel J, Kamil S, Veeramachaneni K, et al. OpenTuner: An Extensible Framework for Program Autotuning. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*; 2014: 303–316
3. Nugteren C, Codreanu V. CLTune: A Generic Auto-Tuner for OpenCL Kernels. In: *Proceedings of the IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*; 2015: 195–202.
4. Werkhoven Bv. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 2019; 90: 347–358.
5. Kisuki T, Knijnenburg PMW, O'Boyle MFP. Combined selection of tile sizes and unroll factors using iterative compilation. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*; 2000: 237–246.
6. Seymour K, You H, Dongarra J. A comparison of search heuristics for empirical code optimization. In: *2008 IEEE International Conference on Cluster Computing*; 2008: 421–429
7. Balaprakash P, Wild SM, Hovland PD. An Experimental Study of Global and Local Search Algorithms in Empirical Performance Tuning. In: *High Performance Computing for Computational Science*. LNCS 7851. Springer; 2013: 261–269.
8. Petrovič F, Střelák D, Hozzová J, et al. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. *Future Generation Computer Systems* 2020; 108: 161–177.
9. Střelák D, Sorzano COS, Carazo JM, Filipovič J. A GPU Acceleration of 3D Fourier Reconstruction in Cryo-EM. *The International Journal of High Performance Computing Applications* 2019; 33(5): 948–959.
10. Filipovič J, Petrovič F, Benkner S. Autotuning of OpenCL Kernels with Global Optimizations. In: *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems (ANDARE '17)*; 2017: 1–6.
11. Che S, Boyer M, Meng J, et al. Rodinia: A benchmark suite for heterogeneous computing. In: *IEEE International Symposium on Workload Characterization (IISWC)*; 2009: 44–54.
12. Kelefouras V, Djemame K. A methodology correlating code optimizations with data memory accesses, execution time and energy consumption. *The Journal of Supercomputing* 2019; 75(10): 6710–6745.
13. Kelefouras V. A methodology pruning the search space of six compiler transformations by addressing them together as one problem and by exploiting the hardware architecture details. *Computing* 2017; 99(9): 865–888.
14. Roy A, Balaprakash P, Hovland PD, Wild SM. Exploiting Performance Portability in Search Algorithms for Autotuning. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*; 2016: 1535–1544.
15. Cover TM, Thomas JA. *Elements of information theory*. John Wiley & Sons. 2nd ed. 2012.
16. Oľha J, Hozzová J, Fousek J, Filipovič J. Exploiting Historical Data: Pruning Autotuning Spaces and Estimating the Number of Tuning Steps. In: *Euro-Par 2019: Parallel Processing Workshops*. LNCS 11997. Springer; 2020: 295–307.
17. Rasch A, Gorlatch S. ATF: A generic directive-based auto-tuning framework. *Concurrency and Computation: Practice and Experience* 2018: e4423.
18. Veček N, Mernik M, Filipič B, Črepinšek M. Parameter tuning with Chess Rating System (CRS-Tuning) for meta-heuristic algorithms. *Information Sciences* 2016; 372: 446–469.
19. Birattari M, Stützle T, Paquete L, Varrentrapp K. A Racing Algorithm for Configuring Metaheuristics. In: *Proceedings of the Genetic and Evolutionary Computation Conference*; 2002: 11–18.
20. Price J, McIntosh-Smith S. Improving Auto-Tuning Convergence Times with Dynamically Generated Predictive Performance Models. In: *9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*; 2015: 211–218.
21. Muralidharan S, Roy A, Hall M, Garland M, Rai P. Architecture-Adaptive Code Variant Tuning. *SIGARCH Computer Architecture News* 2016; 44(2): 325–338.
22. Cummins C, Petoumenos P, Wang Z, Leather H. End-to-End Deep Learning of Optimization Heuristics. In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*; 2017: 219–232
23. Ashouri AH, Bignoli A, Palermo G, Silvano C, Kulkarni S, Cavazos J. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. *ACM Transactions on Architecture and Code Optimization* 2017; 14(3): 1–28.