Model-based Autotuning of Discretization Methods in Numerical Simulations of Partial Differential Equations

Nesrine Khouzami[a], Friedrich Michel[a], Pietro Incardona[a,b,c], Jeronimo Castrillon[a] and Ivo F. Sbalzarini[a,b,c]

[a]Technische Universität Dresden, Faculty of Computer Science, Dresden, Germany
[b]Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany
[c]Center for Systems Biology Dresden, Dresden, Germany

ARTICLE INFO

Keywords:
Numerical Simulations
Discretization Methods
Autotuning
Domain-Specific Languages
Performance Models
Compilers

ABSTRACT

We present an autotuning approach for compile-time optimization of numerical discretization methods in simulations of partial differential equations. Our approach is based on data-driven regression of performance models for numerical methods. We use these models at compile time to automatically determine the parameters (e.g., resolution, time step size, etc.) of numerical simulations of continuum spatio-temporal models in order to optimize the tradeoff between simulation accuracy and runtime. The resulting autotuner is developed for the compiler of a Domain-Specific Language (DSL) for numerical simulations. The abstractions in the DSL enable the compiler to automatically determine the performance models and know which discretization parameters to tune. We demonstrate that this high-level approach can explore a large space of possible simulations, with simulation runtimes spanning multiple orders of magnitude. We evaluate our approach in two test cases: the linear diffusion equation and the nonlinear Gray-Scott reaction-diffusion equation. The results show that our model-based autotuner consistently finds configurations that outperform those found by state-of-the-art general-purpose autotuners. Specifically, our autotuner yields simulations that are on average 4.2x faster than those found by the best generic exploration algorithms, while using 16x less tuning time. Compared to manual tuning by a group of researchers with varying levels of expertise, the autotuner was slower than the best users by not more than a factor of 2, whereas it was able to significantly outperform half of them.

1. Introduction

Computer simulations are the third pillar of science, alongside theory and experiment. They allow studying nonlinear theories and predicting dynamics at scales inaccessible by direct experimentation (e.g., weather forecasting and astrophysics). Computer simulations are notoriously computationally intensive and have thus motivated much research in numerical algorithms, software libraries, compiler optimization, programming languages, and computer architecture. Optimization, in particular, has the potential to speed up simulations by orders of magnitude and has thus become a key enabler for scientific progress.

Today, however, the complexity of computer architectures makes it hard to predict the impact of software optimizations. For this reason, classic compiler optimizations, based on static information, struggle to consistently find transformations that lead to significant performance improvement. This motivated a large body of research on autotuning compilers, which iteratively explore the space of implementation variants by applying transformations and directly measuring their impact. Most classic autotuners aim to improve performance for specific hardware architectures to achieve performance portability. In numerical simulations, such systems are well known for linear algebra kernels and generic matrix and vector operations [1, 2] as well as Fourier transforms [3, 4, 5]. While all these approaches achieve improved performance, they operate on a concrete specification of a mathematical expression, e.g., a concrete matrix shape. More generic autotuning requires higher levels of abstraction, as for example provided by Domain-Specific Languages (DSLs) for numerical computing.

*Corresponding authors
nesrine.khouzami@tu-dresden.de (N. Khouzami); friedrich.michel@tu-dresden.de (F. Michel); incardon@mpi-cbg.de (P. Incardona); jeronimo.castrillon@tu-dresden.de (J. Castrillon); ivo.sbalzarini@tu-dresden.de (I. F. Sbalzarini)
ORCID(s):

Modern DSLs offer a unique opportunity to explore such high-level optimizations. This is especially true in declarative DSLs that allow users to express the governing equations of a simulation. In this paper, we leverage the high-level semantics of the Open Particle Mesh Environment (OpenPME) [6], a DSL for numerical simulations, the successor of the Parallel Particle-Mesh Environment (PPME) [7]. This DSL describes the behavior of continuous fields in time and space according to partial differential equations (PDEs). During compilation, fields are sampled on discretization points and time is discretized using a configurable time step. Many typical numerical methods can be expressed in this language to discretize and numerically solve PDEs. Here, we exploit the high-level information from this DSL to automatically configure a numerical discretization method across different PDEs.

Discretization methods are configured by parameters (e.g., resolution, time step size) that collectively determine the accuracy and runtime of the simulation. Here, we propose a multi-objective autotuning approach to automatically determine these parameters while balancing simulation runtime and accuracy. Our approach leverages predictive performance models of numerical discretization methods. These are calibrated at compile time by regression of theoretically known error scaling to empirical measurements. We therefore propose a performance-model-based compiler autotuning strategy that proves more efficient than generic autotuning algorithms that do not leverage performance models.

A comparison of our proposed approach with generic optimization techniques shows that data-driven performance models help find better configurations faster. Specifically, our approach outperforms state-of-the-art autotuners by a factor of 4.2 on average, while using 16x less optimization time. Additionally, we compare with hand-optimized configurations provided by users. While some of them were able to find faster configurations by at most a factor of 1.8, our autotuner outperformed approximately half of the manual results and required less tuning time. This suggests that model-based autotuning is adept at finding performant configurations of numerical methods in simulation codes.

The remainder of the paper is organized as follows: In Section 2, we provide the background necessary to understand the discretization methods and the target autotuning environment. Section 3 describes the autotuning system behind the optimization approach introduced in Section 3.4. The evaluation of the optimized algorithm and the obtained results are presented in Section 4. Related work is discussed in Section 5, followed by conclusions and future work in Section 6.

2. Background

We provide essential background to understand the spatial and temporal discretization methods considered, and we give an overview of our autotuning target DSL and its library.

2.1. Discretization methods

Continuous models, such as PDEs, need to be numerically discretized before they can be simulated on a digital computer. Discretization chooses a finite number of representative points in space and time at which the solution is to be computed. These points are called *discretization points* or *sampling points*. The discretization points are often chosen to lie on a computational grid, such as a regular Cartesian mesh or an unstructured triangulation, but they can also be chosen as arbitrary points in space. We here focus on the latter case and consider mesh-free discretizations of space [8] with explicit time discretization.

The simplest explicit time discretization method is the explicit Euler method. It approximates the time derivative of a quantity y at simulation time step $t_n = n\delta t$ as:

$$\left.\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right|_{t=t_n} \approx \frac{y(t_n) - y(t_{n-1})}{\delta t},$$

where $\delta t = t_n - t_{n-1}$ is the simulation time step size. The smaller δt, the more accurate the simulation becomes. The explicit Euler method has convergence rate 1, meaning that the simulation error reduces proportionally with δt.

Derivatives in space are contained in PDE models as differential operators

$$D^\beta = \frac{\partial^{|\beta|}}{\partial x_1^{\beta_1} \partial x_2^{\beta_2} \cdots \partial x_d^{\beta_d}},$$

where d is the dimension of the space and β is the order of the derivative. Several classic methods exist to approximate the result of applying a differential operator to a continuous field $f(\mathbf{x})$ discretized over a set of discretization points.

Smoothed Particle Hydrodynamics (SPH) [9, 10] for instance approximates spatial derivatives as:

$$D^\beta f(\mathbf{x}) \approx \sum_p f(\mathbf{x_p})[D^\beta W_\epsilon](\mathbf{x} - \mathbf{x}_p),$$

where W_ϵ is an analytically known smoothing kernel of width ϵ that sets the spatial resolution of the simulation. The error convergence rate depends on the smoothing kernel. A popular choice is to use Gaussian kernels, leading to a convergence rate of 2, meaning that the simulation error reduces with the square of the spatial resolution ϵ. Throughout this paper, we use Gaussian kernels. Since Gaussians have infinite support, the kernels are truncated at distances $|\mathbf{x} - \mathbf{x}_p| > r_c$ with a user-defined cutoff radius r_c. The method parameters are illustrated in Fig. 1 for evenly spaced particles (inter-particle distance h) in 1D.

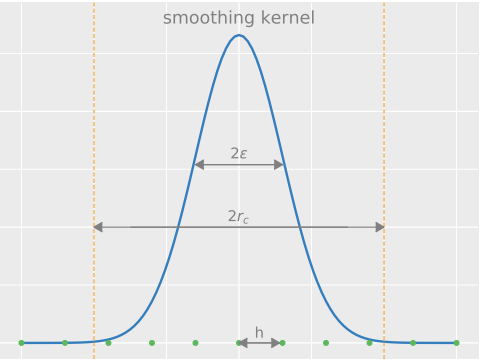


Figure 1: Illustration of spatial discretization parameters in 1D.

An alternative to SPH is Particle Strength Exchange (PSE) [11, 12]. In PSE, the approximation is:

$$D^\beta f(\mathbf{x}) \approx \frac{1}{\epsilon^{|\beta|}} \sum_p (f(\mathbf{x}_p) \pm f(\mathbf{x}))\eta_\epsilon^\beta(\mathbf{x} - \mathbf{x}_p),$$

where the negative sign is used for odd derivatives (i.e., odd $|\beta|$, e.g., first derivative and third derivative), and the plus sign for even derivatives. PSE uses a different kernel η_ϵ^β for every differential operator. Unlike SPH, PSE is symmetric, meaning that it exactly conserves the total amount of f in the simulation. This may be desired in physics simulations if f models the density of a conserved quantity, such as mass, energy, or charge.

A third alternative is Discretization-Corrected PSE (DC-PSE) [13]. This method achieves higher accuracy than PSE, but requires computing a separate kernel η for each particle at each simulation time step, hence incurring a higher computational cost per particle. The approximation formula of DC-PSE is identical with the above formula for PSE, but with these particle-specific kernels. Finite-difference methods and other collocation schemes are special cases of DC-PSE when the discretization points are on a grid [13].

2.2. OpenPME and the OpenFPM library

The OpenPME is a *Problem Solving Environment* [6] for particle-mesh simulations, which extends over the DSL in [7]. OpenPME provides a DSL as intermediate layer between application developers and the OpenFPM C++ library designed to implement scalable particle, mesh, and hybrid particle-mesh simulations on parallel computers. OpenFPM makes heavy use of C++ templates and template meta-programming (TMP) to provide arbitrary-dimensional and data-type-agnostic abstractions for domain decomposition, dynamic load balancing, inter-processor communication, GPU computing, and file I/O [14].

OpenPME allows users to focus on the numerical methods and the models to be simulated. It provides high-level domain-specific abstractions to express simulation applications. The DSL is based on a metamodel [6] covering domain-specific abstractions using particles, meshes, and hybrid (particles and meshes) methods to simulate both discrete and continuous models. An OpenPME program contains three phases: initialization, simulation, and visualization. The initialization phase defines all parameters for the simulation, such as the space dimension, the domain

size, the boundary conditions, the numerical methods to be used, the space and time resolution. The simulation phase contains the main *time loop*, where the spatially discretized fields are evolved according to the temporal discretization scheme chosen. In the visualization phase, the results are stored in *VTK* files for visualization.

Figure 2 exemplarily shows an excerpt of the OpenPME main time loop for the Gray-Scott reaction-diffusion simulation [15] used here as one of the test cases. As can be seen from the example, OpenPME affords a high level of abstraction and is therefore well suited for autotuning. In the code, the discretization methods are specified by the programmer, here `explicit_euler` and `DC-PSE`. The discretization parameters (here: δt, h, ϵ, and r_c) are not defined by the user in this example, indicating to the compiler that they shall be autotuned.

```
simulation
...
time loop
 start: 0  stop: 5000
 temporal method: explicit_euler
 spatial method: DC-PSE
  ∂u/∂t = Du * ∇²u - u * v² + F * (1 - u)
  ∂v/∂t = Dv * ∇²v + u * v² - v * (F + k)
```

Figure 2: Excerpt from the OpenPME code for the Gray-Scott simulation test case. Undefined parameters are autotuned.

3. Autotuning of Numerical Discretization

Building an autotuning system requires (i) defining the search space of possible configurations, (ii) ranking the configurations by a directly measurable or composed objective, (iii) designing measurements to evaluate the objectives for a given configuration, and (iv) devising a search algorithm that finds high-ranking configurations by measuring only a small subset of all configurations from the search space. These are described in the following subsections.

3.1. Search space

For simplicity, and to ensure consistency of all tested methods, we place the discretization points on a regular Cartesian mesh. The mesh spacing $h > 0$ then directly controls the spatial sampling. Other simulation parameters are the time step size $\delta t > 0$, smoothing kernel width $\epsilon > 0$, and the cutoff radius $r_c > 0$ (see Section 2.1). This defines a one-sided, infinite 4D continuous search space. For autotuning, this search space needs to be bounded and discretized. For the test cases presented here, this is done as shown in Table 2, yielding discrete search spaces containing 184 320 to 307 200 configurations.

3.2. Ranking of configurations

Simulation accuracy and runtime are two conflicting objectives. The optimization goal therefore has to be defined as a combination of both. This can be done by searching for tradeoff-optimal solutions, or by combining the objectives to a single one [16]. Tradeoff-optimal solutions, also called Pareto optimal, are configurations that are not outperformed by any other configuration in all objectives. This yields a *Pareto front* of multiple tradeoff-optimal configurations from which the user has to choose. Since we integrate our approach into the OpenPME compiler, we are interested in one single configuration and therefore have to define a preference function. In numerical simulations, this is usually done by letting the user define an error threshold. This is naturally understood by users. Among the configurations yielding a simulation accuracy below this threshold, one then aims to find the configuration with minimal runtime. Similarly, a runtime threshold would be possible, but more subjective to the execution environment. Other typical approaches, such as a combined score using a weighted sum, should be avoided since the choice of weights is not intuitive to the user.

3.3. Measurements

Given the optimization objectives, the autotuner needs to measure the runtime and the numerical accuracy of every configuration it tries. While the former is easy to measure, evaluating the accuracy would require knowing the exact solution of the PDEs, which is usually not available in simulations.

Instead of the unknown exact solution, we therefore use a highly accurate *reference simulation*. This reference simulation is computed once in the beginning using a higher space and time resolution than any configuration in the search space. Using this reference simulation, we measure accuracy of a given test configuration as the L_2 norm

$$L_2 = \sqrt{\frac{1}{N}\sum_{p=1}^{N}\left(f_{\text{reference}}(\mathbf{x}_p) - f_{\text{test}}(\mathbf{x}_p)\right)^2}.$$

The L_2 norm is the mean squared error over all N discretization points, where $f(\mathbf{x}_p)$ is the value of a field f at point $\mathbf{x}_p$, either in the reference simulation or the specific numerical configuration tested. Other error norms, such as L_∞, could be used instead.

The drawback of this approach is that evaluating the reference simulation requires considerable runtime. To evaluate the runtime and accuracy of a test configuration, however, it is sufficient to perform a few simulation time steps. This allows the use of reference simulations, that could not feasibly be executed for the entire simulation length. This is significant, since requiring the full execution of a more accurate simulation would render the tuning pointless.

3.4. Model-based search algorithm

It is challenging to effectively explore large search spaces by sampling only a few configurations. Moreover, the time it takes to measure a configuration varies greatly across the search space. Some configurations can be measured in seconds, whereas others require tens of minutes. For this reason it is desirable to avoid measuring slow configurations altogether. To achieve this, we leverage predictive data-driven performance models of the numerical methods.

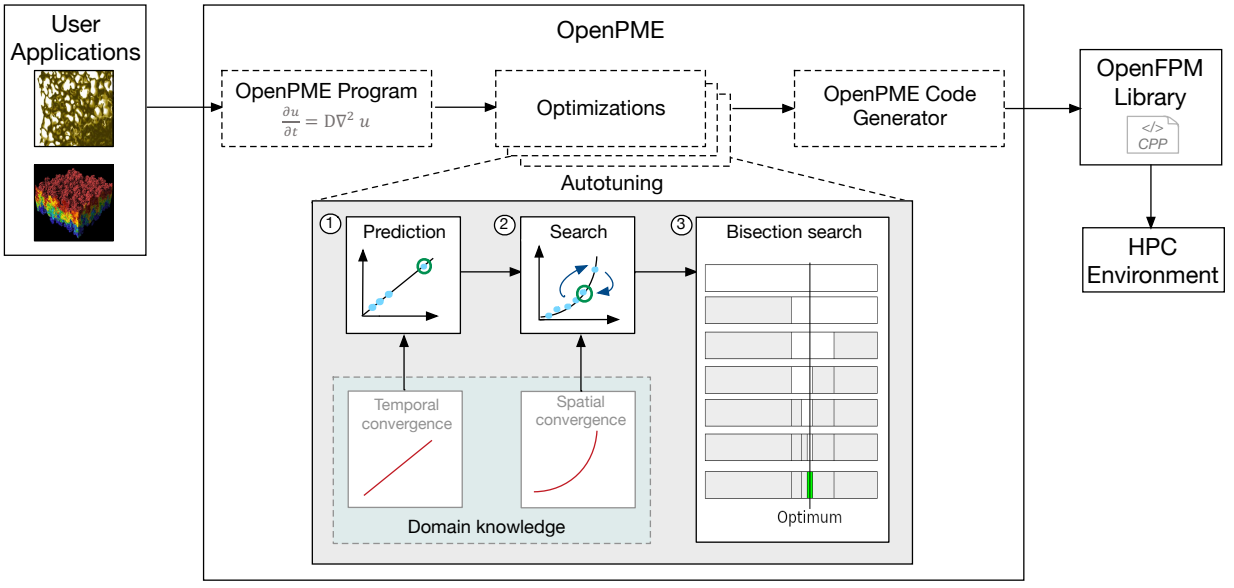


Figure 3: Autotuning based on regression over predictive performance models.

The overall strategy is illustrated in Fig. 3. Based on the domain knowledge of the theoretical convergence rates (here: linear for explicit Euler and quadratic for the space discretizations considered), the autotuner performs three steps (1) *Prediction* of the time step size δt using a calibrated performance model. (2) *Model-based search* for the space resolution h that achieves an accuracy above the user-provided threshold with minimal runtime for the given δt. This search is guided by iteratively re-fitting the performance model for the spatial discretization. (3) *Bisection search* over all remaining parameters, for which no performance models are available. These three steps yield the discretization parameters that the compiler then uses to translate the continuous simulation description in the OpenPME program to a discretized simulation code for OpenFPM. The translation from OpenPME to OpenFPM takes place in the "OpenPME Code Generator", where multiple model-to-model transformations are performed. A model is a graph-like

representation with a spanning tree equivalent to an abstract syntax tree (AST). In a model-to-model transformation, an input graph is mapped to an output graph in order to lower the program to an intermediate representation, followed by a final text-generation phase that produces the C++ output code [6].

For numerical discretization methods, the accuracy performance models required for steps (1) and (2) are naturally available, since the convergence rates of the numerical errors are theoretically known. This means that it is known ("domain knowledge") how simulation accuracy *scales* with space and time resolution h and δt, respectively.

This is illustrated in Fig. 4 for the example of the Gray-Scott reaction-diffusion simulation. For the second-order accurate space discretization methods considered here, we expect a slope of -2 on these doubly logarithmic plots, as indicated by the dashed black lines.

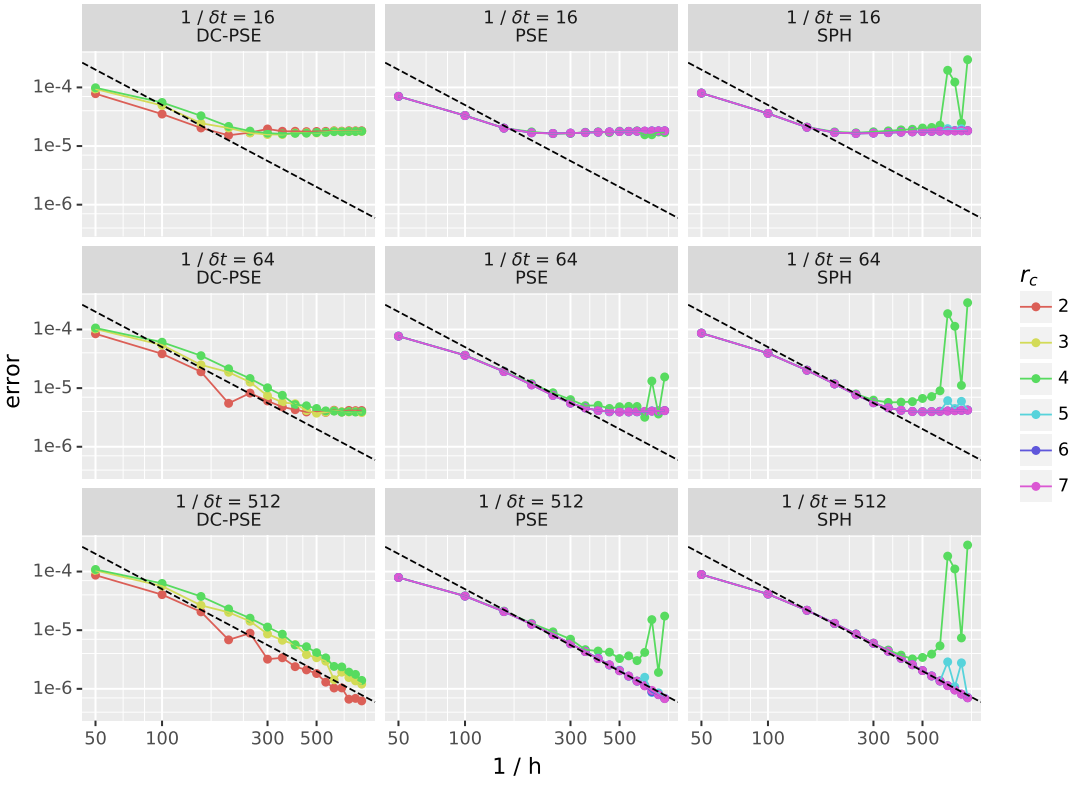


Figure 4: Measured error in Gray-Scott reaction-diffusion simulations of different spatial resolutions $1/h$ for $\epsilon = h$. The dashed line shows the theoretical slope of -2 of second-order accurate discretizations.

The actual measurements, however, deviate from the ideal theoretical line, because the theoretical rates assume otherwise perfect conditions, like infinite r_c, unbounded simulation domain, and arbitrarily high floating-point precision. In addition, the theoretical models predict the *slope*, but not the absolute values of the errors, which also depend on other simulation parameters and on the specific equation being simulated. If either $1/\delta t$ or r_c is not chosen sufficiently large, the error does not converge with the expected rate, irrespective of the value of h. This phenomenon is known as *numerical instability*. A precise threshold for when a simulation becomes numerically unstable is problem-specific and cannot be predicted in general. For these two reasons, a predictive performance model for accuracy can only be learned from measured data. Due to the theoretically known scaling laws, however, only few measured configurations are required to fit the regression model.

Our autotuner determines these data-driven performance models automatically from the configurations it tests. The theoretical convergence rates are known from the DSL, where the user specifies the numerical methods to be used. This knowledge is integrated in a three-step model-based search method as given in Algorithm 1:

Step 1: The autotuner fixes all parameters but δt (here: $h = 1/400$, $\epsilon = h$, and r_c the maximum allowed value). Then, it measures configurations for three different values of δt (here: $\delta t \in \{1, \frac{1}{16}, \frac{1}{32}\}$). The accuracy model for

time discretization (here: explicit Euler, error $= a\delta t$) is fitted through these three points using linear least-squares regression. This model is then used to predict the δt that just provides the target accuracy. This δt is then reduced by a factor of 1.5 in order to provide margin for optimizing the remaining parameters. The concrete numbers shown here are the hyperparameters used in this paper.

Step 2: Using the δt predicted in Step 1, the autotuner iteratively searches for h, while keeping all other parameters fixed. The search is assisted by the accuracy model for spatial discretization. This is illustrated in Fig. 5 for the first four iterations of Algorithm 1 on the quadratic regression model error $= ah^2$ for numerical methods with a spatial convergence order of 2. The algorithm starts from the largest possible value of h (Fig. 5a), as this will have the shortest runtime. Fitting the parameter a in the model through this point and the point $(1/h, 1/\text{error}) = (0, 0)$ results in a first prediction for the h required to reach the accuracy threshold (dashed line). This predicted configuration is measured subsequently, yielding a measured error that does not necessarily match the prediction. The model is thus re-fitted using all data points, yielding a new prediction (Fig. 5b). This process is iterated until a configuration slightly above the accuracy threshold is found (Fig. 5d) for which h matches the predicted h in the discrete search space.

Algorithm 1: Model-based search

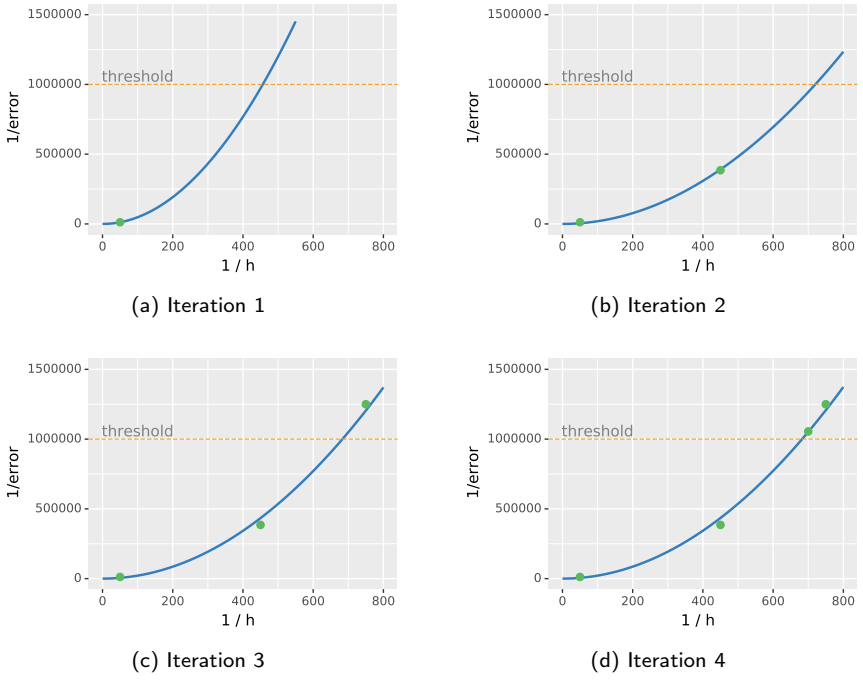
Result: Optimized configuration
`// Define initial configuration c`
Set $c.r_c$ to maximum value
Set $c.\delta t$ to predicted value
Set $c.\frac{\epsilon}{h}$ to 1
Set $c.h$ to maximum value in the search space $h_{\max}$
Set $C = \emptyset$
`// Optimize h`
repeat
  Measure configuration c and place in set C
  Use linear regression on C to estimate a in $e = a \cdot h^2$
  Use $h = \sqrt{\frac{e}{a}}$ to estimate h_{estimate} for $e_{\text{threshold}}$
  $c.h := \max(h_{\text{estimate}}, h_{\min})$
until $c \in C$ *and* $e(c) \leq e_{\text{threshold}}$;
`// Optimize` ϵ`,` r_c `, and` δt
Use bisection search to find the smallest ϵ that does not change e significantly
Use bisection search to find the largest δt that does not change e significantly
Use bisection search to find the smallest r_c that does not change e significantly

Step 3: Parameters for which no performance model is available, here r_c and ϵ, are optimized using bisection search with h fixed at the value found in Step 2. The time step size δt is also included again in bisection search to fine-tune it. Bisection search considers the part of the search space between the current value of each parameter and the search space boundary with minimal runtime (i.e., minimal r_c, maximal δt). Since the runtime behavior with ϵ is not monotone, the entire search space is considered in this dimension. The search stops when it has found the minimum-runtime configuration.

The two main advantages of this algorithm over general-purpose optimization are (i) the use of known convergence orders to assist the search and (ii) the awareness of configuration evaluation times avoiding measuring slow configurations (small h and δt). The approach also essentially optimizes the parameters individually. This has shown to be very effective when done as described. This simple approach, however, is still coarse and may miss potentially beneficial fine-tuning of the parameters and parameter correlations. This could potentially be improved by a subsequent local search. But as shown in the benchmarks below, the simple algorithm presented here already performs remarkably well.

4. Evaluation

We empirically evaluate and benchmark the proposed approach against general-purpose autotuners as implemented in the OpenTuner framework [17]. The available optimization techniques in OpenTuner do not exploit domain knowledge and consider the search space as a black-box. This allows us to quantify the effect of including model-based



(a) Iteration 1 (b) Iteration 2

(c) Iteration 3 (d) Iteration 4

Figure 5: Illustration of the first four model-based search iterations to find h for a second-order accurate discretization method.

information.

For the evaluation, we consider two test cases: the linear diffusion equation and the nonlinear Gray-Scott reaction-diffusion equation. Each of these PDEs is discretized with three discretization methods: SPH, PSE, and DC-PSE (see Section 2.1) all using explicit Euler time discretization. For each combination, different autotuners are used to find the parameters h, ϵ, r_c, and δt for two error levels: 10^{-5} and 10^{-6}. We tune the normalized ratios r_c/h and ϵ/h rather than r_c, h, and ϵ separately. This removes correlations with h in the search space, hence improving the performance of general-purpose tuners, which are otherwise unaware of this correlation. Each experiment is repeated 10 times. Additionally, we evaluate a subset of the experiments against hand-optimized configurations found by 11 users.

4.1. Experimental setup

We compare all methods listed in Table 1. For the comparison methods, we use three different maximum optimization timeouts: 1-fold, 8-fold, and 16-fold the time used by our model-based approach. The search space is bounded and discretized as given in Table 2. The value ranges of the tuning parameters are manually chosen based on prior knowledge. These hyperparameters are provided by the user to the tuner by means of descriptor files. For this evaluation, the parameter ranges are chosen wide enough to cover a large variety of behaviors. We use the same search space throughout all benchmarks. It contains 307 200 configurations for SPH and PSE and 184 320 for DC-PSE. The tuning runs could take as long as the final simulation itself and sometimes even longer. That is why only a fraction of the configurations are measured to keep the tuning time reasonable. The measurements of individual configurations were done at $T = 2$ and took between a few milliseconds and over 10 minutes depending on the parameters. The resulting final configurations were remeasured at $T = 100$ for more stable results [1]. As reference simulation for accuracy measurement (see Section 3.3), we use particle strength exchange (PSE) with a Gaussian kernel of width $\epsilon = 0.8h$ with $h = 1/1600$, $\delta t = 1/1024$, and $r_c = 9h$ in all cases (model-based search, generic optimization algorithms, manual tuning). This reference simulation is about four times more accurate than the most accurate configuration in the search

[1] Full simulations can be expected to use final simulation times of $T = 5000$. For Gray Scott, the characteristic patterns stabilize around that point in time. Such a high value of T for our experiments would require a prohibitively large computational time, without adding insight into our results analysis.

method	abbr.
model-based search (ours)	*ModS*
OpenTuner default	AUC
Differential Evolution	DE
Genetic Algorithm	GA
Uniform Greedy Mutation	UniGM
Gaussian Greedy Mutation	NormGM
Regular Nelder Mead	RegNM
Multi-Nelder Mead	MulNM
Particle Swarm Optimizer	PSO
Random search	Rand

Table 1
List of compared methods.

param.	values
$1/h$	$\{50, 100, 150, \dots, 800\}$
$1/\delta t$	$\{1, 2, 3, 4, \dots, 256\}$
ϵ/h	$\{0.6, 0.7.0.8, \dots, 2.0\}$
r_c/h	$\{5, 6, 7, 8, 9\}$ (SPH, PSE)
r_c/h	$\{2, 3, 4\}$ (DC-PSE)

Table 2
Search space for autotuning. All simulations use Gaussian smoothing kernels and explicit Euler time discretization.

space. Computing this reference simulation takes 2 and 5 hours for diffusion and Gray-Scott, respectively, and is not included in the search time since it is the same for all approaches and we mainly focus on the optimization time.

All measurements are conducted on the *taurus* HPC system of TU Dresden on Intel Haswell nodes equipped with two Intel Xeon E5-2680v3 CPUs, each with 12 cores at 2.50 GHz. The 24-core nodes have 64 GB of RAM available and are connected via an Infiniband network with 40 Gb/s bandwidth. Each benchmark was run in parallel on 22 cores using OpenFPM v.2.0.0 and OpenMPI v.3.1.1 on Red Hat Enterprise Linux 7.9. We use 22 cores instead of all 24 in order to reduce measurement standard deviation from background load fluctuations of the operating system, leaving 2 cores for the operating system.

The first test case considers numerical simulations of the diffusion of a continuous smooth field $u(\mathbf{x}, t)$ in space $\mathbf{x}$ and time t as governed by the isotropic homogeneous *diffusion equation*

$$\frac{\partial u}{\partial t} = D\nabla^2 u,$$

where D is the diffusion constant and ∇^2 is the Laplace operator, i.e., the sum of all second derivatives of u with respect to all space coordinates. We numerically simulate the space-time dynamics of u for $D = 10^{-4}$ by solving this equation on the 2D unit square with initial condition $u(\mathbf{x}, 0) = \frac{1}{0.16\pi} \exp\left(\frac{|\mathbf{x}-\mathbf{c}|}{0.16}\right)$ with $\mathbf{c}^\top = [0.5, 0.5]$ and periodic boundary conditions on all four sides.

The second test case considers the *Gray-Scott reaction-diffusion equation* [15], which models nonlinear spatio-temporal patterns emerging from the interaction of two diffusing and reacting chemicals. It is described by the PDEs (cf. Fig. 2)

$$\frac{\partial u}{\partial t} = D_u\nabla^2 u - uv^2 + F(1-u), \qquad \frac{\partial v}{\partial t} = D_v\nabla^2 v + uv^2 - (F+k)v,$$

where D_u and D_v are the diffusion constants of the two chemicals with concentration fields $u(\mathbf{x}, t)$ and $v(\mathbf{x}, t)$. The scalar constants F and k define the reaction rates. We numerically simulate the space-time dynamics of u and v for $D_u = 4 \cdot 10^{-5}$, $D_v = 2 \cdot 10^{-5}$, $K = 0.055$, $F = 0.03$ by solving these coupled equations on the 2D unit square with initial conditions $u(\mathbf{x}, 0) = 1 - 0.5/\left(\frac{(x_1-0.5)^4+(x_2-0.5)^4}{0.15^4} + 1\right)$, $v(\mathbf{x}, 0) = 0.25/\left(\frac{(x_1-0.5)^4+(x_2-0.5)^4}{0.15^4} + 1\right)$ and periodic boundary conditions on all four sides.

4.2. Autotuning results

Figure 6 shows the measured runtimes for the diffusion simulation after autotuning with a target error threshold of 10^{-5}. The present model-based autotuner finds configurations that generally outperform (in the median and reliability) those found by any comparison method, even when the general-purpose tuners were given 16 times as much time as required by our model-based method (*ModS*). When given the same amount of tuning time, no algorithm but ours managed to find a configuration below the error threshold in every trial. The quartiles for the model-based search

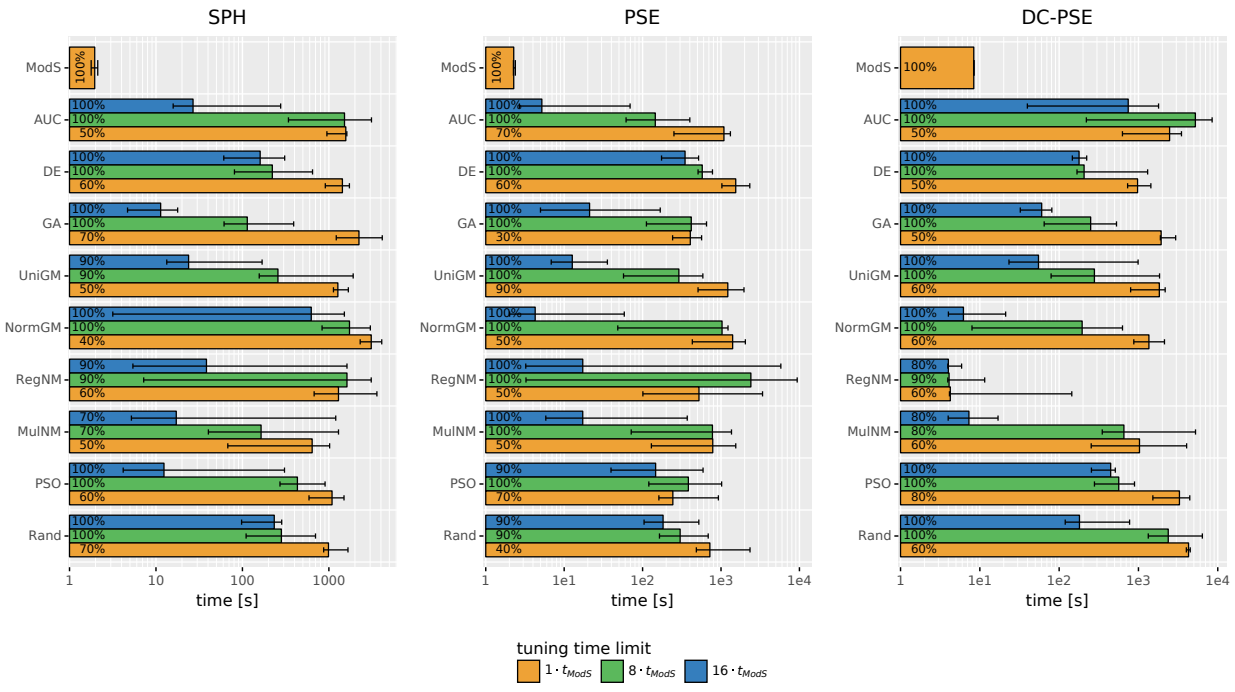


Figure 6: Autotuning results for the diffusion benchmark with an error threshold of 10^{-5}. Bars display the median runtime of the best configuration found over 10 tunings on a logarithmic scale. Error bars show the quartiles over the 10 runs. The percentage within each bar tells how often any configuration below the error threshold was found (success rate).

show the background runtime variations in the HPC system, since this autotuner is deterministic and always finds the same configuration in all of the 10 runs. The results for a target error threshold of 10^{-6} are shown in Fig. 7. While RegNM appears to perform well in the reduced diagram, it shall be noted, that its success rate for the longest tuning time only amounted to 10%, 50% and 70% for SPH, PSE and DC-PSE respectively.

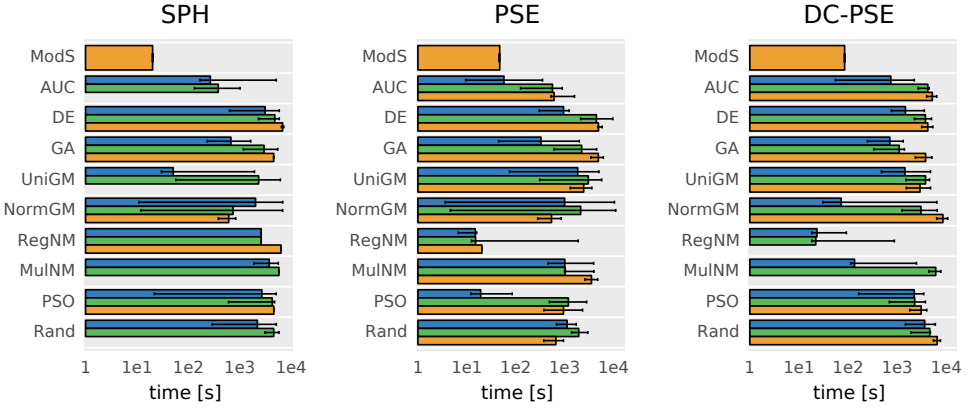


Figure 7: Autotuning results for the diffusion benchmark with an error threshold of 10^{-6}.

In all cases, the model-based search required few iterations and avoided measuring slow configurations. Our model-based autotuner required between 2 and 4 minutes of tuning time for each case. Predicting δt requires at least three measurements by construction. The actual search measured between 7 and 11 configurations, depending on the discretization method and target error.

Figure 8 shows the results for the Gray-Scott simulation with a target error of 10^{-5}. Our model-based autotuner outperforms all other methods within the same tuning time. When allowed 16-fold more tuning time, the GA and NormGM sometimes find equivalent or slightly better configurations, but especially in case of NormGM with less consistency. This is expected for the nonlinear Gray-Scott problem, where the computational cost of evaluating a single configuration can be high. While our approach takes approximately the same amount of optimization steps, the total tuning time increases. This also increases the available tuning time (16x higher) for the other algorithms, resulting in a larger number of measured configurations and better final results. This hypothesis is confirmed when lowering the error threshold to 10^{-6}, as shown in Fig. 9. Our model-based approach is then outperformed by the general-purpose black-box autotuners when they are given 16x the tuning time. For DC-PSE, no valid configurations have been found by our model-based approach, which is why that plot is omitted. Our model-based autotuner required approximately 8 minutes of tuning time to reach an error threshold of 10^{-5} and between 1 and 3 hours to reach 10^{-6}.

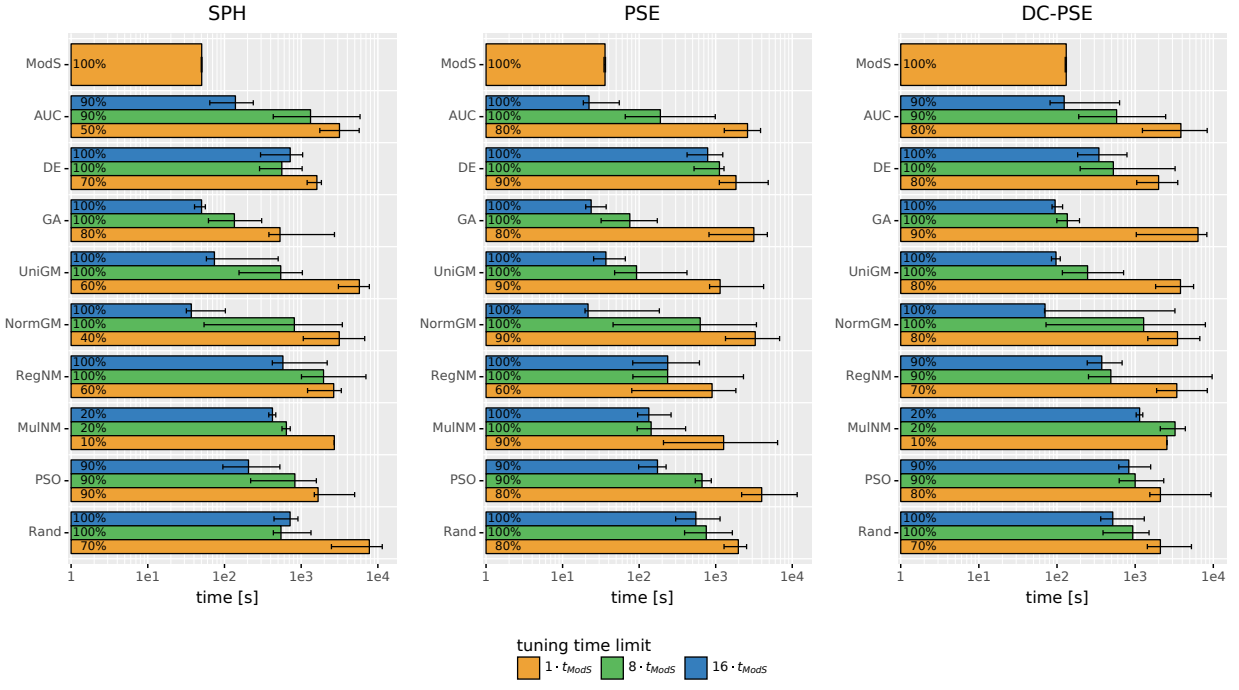


Figure 8: Autotuning results for the Gray-Scott benchmark with an error threshold of 10^{-5}.

Over all experiments, the simulations optimized by our model-based autotuner outperformed valid configurations found by the best comparison algorithm by a factor of 4.2 on average (geometric mean) if the comparison algorithms were given 16x the tuning time. To provide an indication of how much room is left for further optimization, we also compare our model-based autotuner with near-optimum results. To this end, we selected the three most performant techniques over all experiments, namely RegNM, GA, and NormGM and let them run for longer time to pick up the best found configurations. In the case of diffusion with error thresholds of 10^{-5} and 10^{-6} and Gray-Scott with error threshold of 10^{-5}, the best configurations found by the general-purpose techniques were on average 3.3x faster than our model-based tuner's when given 128x the tuning time. For Gray-Scott with an error threshold of 10^{-6}, the general-purpose techniques were given 32x our tuning time. The best configurations found performed on average 2.9x better than our results. These results indicate that there is still potential for future work on better heuristic searches.

4.3. Comparison with manual tuning

We also compare the performance of our model-based autotuner with manual optimizations done by users of varying expertise ranging from graduate students to professionals with over 20 years of experience. For this user study, 11 researchers from the field of computational science were provided with the same measurement interface, search space specification, and optimization goal as the autotuner. The measurement interface allowed them to specify and

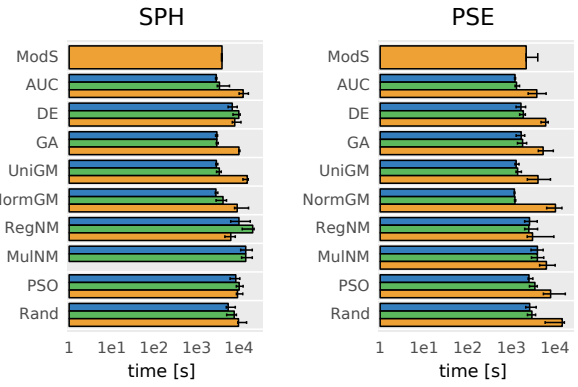


Figure 9: Autotuning results for the Gray-Scott benchmark with an error threshold of 10^{-6}. No valid configurations for DC-PSE were found, although they exist.

test configurations from a text-based user interface without manually editing, recompiling, and submitting the code. Due to time restrictions, the experiment was only done for PSE and an error threshold of 10^{-6}. The available time for the experiment, excluding setup and instructions, was limited to 90 minutes.

For the diffusion simulation, 6 users were able to find configurations that performed equally or up to a factor of 1.8x better than our model-based autotuner. The other 5 users were outperformed by the autotuner by factors between 5 and 1377 with one person unable to find any configuration fulfilling the accuracy requirement.

In the experiment with the Gray-Scott simulation, 5 users outperformed the tuner with up to 1.8x faster configurations. The tuner outperformed 3 users by factors between 2.8 and 13, and 3 users did not have enough time to find any configuration within the error tolerance.

Summarizing, this small-scale user study suggests that experienced users are able to outperform our autotuner by about 2-fold. Many users, however, could significantly benefit from our autotuner.

5. Related Work

To assist programmers in bypassing manual optimizations of their code, many successful autotuning systems have been proposed to automate the search for the best implementation and improve performance.

ATLAS [2] is one of the earliest an autotuning systems for linear algebra libraries. It applies optimization strategies for loop unrolling, latency hiding, blocking factors, etc, based on machine architecture specifics. ATLAS explores possible implementations in a generated search space, independently tuning each parameter while the others are fixed. Another well-known example is the FFTW [3, 4] C library for autotuning fast Fourier transforms. FFTW uses a planner that takes as input a description of the input data structure and the hardware features, decomposes the tuning problem into sub-problems, and selects the fastest variant.

OSKI [1] provides a collection of low-level primitives for libraries and applications to automatically tune sparse-matrix kernels for specific hardware. The tuning process in OSKI is done at runtime, as the matrix is unknown beforehand. Another autotuning framework for computational kernels is CHiLL [1]. It evaluates at compile time the interaction between multiple parameters in a search space to generate a set of kernel mappings to an architecture and choose the best-performing implementation.

Orio [18] is an annotation-based tuning system that takes as input an annotated code to generate multiple tuned versions and trigger low-level performance optimizations either specific to a certain hardware architecture, or independent of it. Orio provides different search heuristics to reduce the number of explored configurations in the search space. OpenABL [19] is a tuning framework assuring portability of generated code from DSLs across multiple large-scale systems. It provides a DSL for agent-based simulations and a source-to-source compiler that generates code through pluggable backends that leverage the AST-based intermediate representation exposing parallelism, locality, and synchronization at the agent level. In [20], the authors propose a portable autotuning framework for stencil computations. The autotuning approach uses a domain-specific transformation and code generation, combined with automated search, to transform an annotated sequential Fortran stencil expression into tuned parallel implementations for different archi-

tectures.

The authors of [21] integrate an autotuning methodology for numerical multigrid solvers into the *PetaBricks* language. The autotuner can tune algorithmic choices in multigrid solvers to optimize the depth and traversal strategy of the grid levels. This autotuner also exploits theoretically known convergence rates to produce a tuned multigrid solver that achieves high performance across a variety of platforms.

In a similar idea to our model-based search, the authors in [22] proposed an autotuning framework based on a performance model to improve parallel I/O operations in HPC applications. They explore the I/O parameters and use empirical predictive models to reduce the search space. The autotuning process starts with a prediction of tunable parameters, selects the best and refits the performance model with the newly collected write time data.

In [23], the autotuning potential of time-discretization methods is investigated. The authors explore whether offline or online autotuning is better for a specific method. Also targeting PDE discretization, two parameters are considered for autotuning: the grid size and the number of processors used.

In [24], the authors presented ATF (Auto-Tuning Framework), a generic framework for automatic program optimization offering a choice of interdependent tuning parameters in high-performance applications. The autotuning process in ATF starts by efficiently exploiting tuning parameter constraints in order to generate the search space, which is stored in memory using a chain-of-trees structure. A multi-dimensional search technique is then used on these tree-like representations to explore the search space. ATF demonstrated improvements in every autotuning phase by generating and exploring search spaces faster and requiring less memory to store them.

The authors of [25] introduced KTT (Kernel Tuning Toolkit), an autotuning framework for accelerator kernels implemented in OpenCL or CUDA. Besides offline autotuning, KTT supports dynamic tuning of code optimization parameters when the input data change. KTT has been demonstrated to generate kernels that reach peak performance with an acceptable overhead when searching the tuning space at runtime.

Another GPU kernel tuning framework is Kernel Tuner [26]. It offers several different search and optimization algorithms to accelerate the tuning process. Kernel Tuner was used in different application scenarios and showed considerable reductions in tuning time when compared with brute force search.

OpenTuner [17] is a framework for building domain-specific multi-objective autotuners. It provides multiple search techniques and predefined data types to support complex tuner representations. In OpenTuner, autotuning techniques share results through a database so that improvements found by one technique can benefit also other techniques. OpenTuner has been successfully used for building autotuners in a variety of distinct projects, demonstrating considerable speedups. In our work, we used a number of methods provided by OpenTuner as baselines in the benchmarks.

mARGOT [27] is a dynamic autotuning framework where the user specifies high-level goals while the application software-knobs are tuned accordingly to provide a suitable configuration. Moreover, mARGOT identifies optimization opportunities in a reactive and proactive way at runtime. The framework has been evaluated in applications ranging from embedded systems to HPC, demonstrating how changes in the execution environment can be leveraged to realize optimizations opportunities.

HyperMapper [28] introduced a new tuning methodology that uses guided search based on active learning for handling multi-objective optimizations, unknown feasibility constraints, and categorical variables. HyperMapper has been originally designed for tuning hardware accelerators in a compiler pass within the Spatial DSL [29]. There, it was able to find better Pareto fronts than state-of-art heuristic random search, and with significantly fewer samples.

To the best of our knowledge, there are no other model-based optimization techniques for the simulation parameters considered in our approach. We thus compare with general-purpose optimization techniques as provided by OpenTuner.

6. Conclusion

We presented an autotuning approach to determine parameters of numerical discretization schemes in simulations of PDEs, such that a user-provided accuracy threshold is met within a short exploration time. Our approach uses predictive performance models for the accuracy of discretization schemes, which are calibrated by linear regression using measurements gathered at compile time by the autotuner. This data-driven approach is more general than using fully analytical performance models. While analytical models would be faster to evaluate, they are only available for specific PDEs, such as the diffusion equation [30]. Our approach, in contrast, is problem-agnostic and works for any simulated PDE. The domain knowledge required for this can be directly extracted by the compiler from the simulation DSL for which the presented autotuner was developed. This provides a high level of abstraction, rendering the autotuner intuitive to users.

We proposed a novel optimization strategy to search the large design space for good configurations. Our search combines model-based prediction with iterative model-assisted search, and finally uses bisection search for parameters for which no performance model exists. Empirically, we showed that this algorithm is able to find valid configurations by performing around 10 measurements in a search space containing hundreds of thousands of possible configurations. Tuning times ranged from a minute for the fast linear diffusion simulation to 3 hours for the slow nonlinear Gray-Scott simulation. In all benchmarks, the performance of the simulations found by our autotuner were orders of magnitude faster than those found by a host of general-purpose autotuners when given the same tuning time. Even with a 16-fold shorter tuning time, our technique was superior by a factor of 4.2 on average over all experiments. In comparison with manual tuning by domain users, the presented model-based autotuner produced simulations that ran at most 2 times slower than those found by the best users and outperformed about half of them. Moreover, a deterministic autotuner makes simulations more reproducible by eliminating the subjective and often arbitrary step of manual parameter selection.

In future work, we will extend our approach to other numerical methods and investigate ways of accelerating the reference simulation for accuracy measurement. Currently, performing the reference simulation takes several hours (2 to 5 hours in the tests presented here), which is the bottleneck of our method. We will also explore how the principles of model-based autotuning transfer to simulations of discrete models, such as molecular dynamics or discrete element simulations, which can also be expressed in the OpenPME DSL. More benchmarks from different domains will help further evaluate the scalability of our autotuning approach with increasing search space size and dimensionality. Finally, it remains to be explored what the best syntax is for the user of the DSL to interact with the autotuner and to inspect and extract tuning results for later reuse. The present implementation will be made available as open source at the time of publication.

Acknowledgments

The authors are grateful to the Centre for Information Services and High Performance Computing (ZIH) of TU Dresden for providing their facilities for the benchmarks. This work was supported in parts by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under grant number 350008342 (project “OpenPME”) and by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under funding code 031L0160 (project “SPlaT-DM – computer simulation platform for topology-driven morphogenesis”).

References

[1] R. Vuduc, J. W. Demmel, K. A. Yelick, OSKI: A library of automatically tuned sparse matrix kernels, in: J. Phys.: Conf. Ser., Vol. 16, IOP Publishing, 2005, p. 521.

[2] R. C. Whaley, J. J. Dongarra, Automatically tuned linear algebra software, in: SC’98: Proc. ACM/IEEE Conf. Supercomputing, IEEE, 1998, pp. 38–38.

[3] M. Frigo, S. G. Johnson, FFTW: An adaptive software architecture for the FFT, in: Proc. IEEE Intl. Conf. Acoustics, Speech and Signal Processing, ICASSP’98 (Cat. No. 98CH36181), Vol. 3, IEEE, 1998, pp. 1381–1384.

[4] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, Proc. IEEE 93 (2) (2005) 216–231.

[5] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, M. Püschel, Spiral in Scala: Towards the systematic construction of generators for performance libraries, SIGPLAN Not. 49 (3) (2013) 125âĂŞ134.

[6] N. Khouzami, L. Schütze, P. Incardona, L. Kraatz, T. Subic, J. Castrillon, I. F. Sbalzarini, The OpenPME problem solving environment for numerical simulations, in: M. Paszynski, D. Kranzlmüller, V. V. Krzhizhanovskaya, J. J. Dongarra, P. M. A. Sloot (Eds.), International Conference on Computational Science (ICCS’21), Springer, Springer International Publishing, Cham, 2021, pp. 614–627. doi:10.1007/978-3-030-77961-0_49.

[7] S. Karol, T. Nett, J. Castrillon, I. F. Sbalzarini, A domain-specific language and editor for parallel particle methods, ACM Trans. Math. Softw. 44 (3) (2018) 34:1–34:32.

[8] S. Li, W. K. Liu, Meshfree Particle Methods, Springer, 2004.

[9] R. A. Gingold, J. J. Monaghan, Smoothed particle hydrodynamics - theory and application to non-spherical stars, R. Astronom. Soc., Montly Notices 181 (1977) 375–378.

[10] L. B. Lucy, A numerical approach to the testing of the fission hypothesis, Astronom. J. 82 (1977) 1013–1024.

[11] P. Degond, S. Mas-Gallic, The weighted particle method for convection-diffusion equations. Part 1: The case of an isotropic viscosity, Math. Comput. 53 (188) (1989) 485–507.

[12] J. D. Eldredge, A. Leonard, T. Colonius, A general deterministic treatment of derivatives in particle methods, J. Comput. Phys. 180 (2002) 686–709.

[13] B. Schrader, S. Reboux, I. F. Sbalzarini, Discretization correction of general integral PSE operators for particle methods, J. Comput. Phys. 229 (11) (2010) 4159–4182.

[14] P. Incardona, A. Leo, Y. Zaluzhnyi, R. Ramaswamy, I. F. Sbalzarini, OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers, Comput. Phys. Commun. 241 (2019) 155–177.

[15] P. Gray, S. Scott, Autocatalytic reactions in the isothermal, continuous stirred tank reactor: Isolas and other forms of multistability, Chem. Engrg. Sci. 38 (1) (1983) 29–43.

[16] K. Deb, Multi-objective optimization, in: Search Methodologies, Springer, 2014, pp. 403–449.

[17] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, S. Amarasinghe, OpenTuner: An extensible framework for program autotuning, in: Proc. 23rd Intl. Conf. Parallel Arch. & Compil., 2014, pp. 303–316.

[18] A. Hartono, B. Norris, P. Sadayappan, Annotation-based empirical performance tuning using Orio, in: IEEE Intl. Symp. on Parallel & Distr. Proc., IEEE, 2009, pp. 1–11.

[19] B. Cosenza, N. Popov, B. Juurlink, P. Richmond, M. K. Chimeh, C. Spagnuolo, G. Cordasco, V. Scarano, Easy and efficient agent-based simulations with the OpenABL language and compiler, Future Generation Computer Systems 116 (2021) 61 – 75.

[20] S. Kamil, C. Chan, L. Oliker, J. Shalf, S. Williams, An auto-tuning framework for parallel multicore stencil computations, in: Proc. IEEE Intl. Symp. Parallel & Distr. Process., IEEE, 2010, pp. 1–12.

[21] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, A. Edelman, Autotuning multigrid with PetaBricks, in: Proc. Conf. High Perf. Comp. Netw., Stor. & Anal., IEEE, 2009, pp. 1–12.

[22] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, M. Snir, Improving parallel I/O autotuning with performance modeling, in: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 253âĂŞ256.

[23] N. Kalinnik, R. Kiesel, T. Rauber, M. Richter, G. RÃijnger, On the autotuning potential of time-stepping methods from scientific computing, in: 2018 Federated Conference on Computer Science and Information Systems (FedCSIS), 2018, pp. 329–338.

[24] A. Rasch, R. Schulze, M. Steuwer, S. Gorlatch, Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF), ACM Transactions on Architecture and Code Optimization 18 (1) (2021) 1–26. doi:10.1145/3427093.

[25] F. Petrovič, D. StÅŹelák, J. Hozzová, J. Ol'ha, R. Trembecký, S. Benkner, J. Filipovič, A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit, Future Generation Computer Systems 108 (2020) 161–177. doi:10.1016/j.future.2020.02.069.

[26] B. van Werkhoven, Kernel Tuner: A search-optimizing GPU code auto-tuner, Future Generation Computer Systems 90 (2019) 347–358. doi:10.1016/j.future.2018.08.004.
URL https://doi.org/10.1016/j.future.2018.08.004

[27] D. Gadioli, E. Vitali, G. Palermo, C. Silvano, MARGOt: A dynamic autotuning framework for self-aware approximate computing, IEEE Transactions on Computers 68 (5) (2019) 713–728. doi:10.1109/TC.2018.2883597.

[28] L. Nardi, D. Koeplinger, K. Olukotun, Practical design space exploration, Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS 2019-Octob (2019) 347–358. doi:10.1109/MASCOTS.2019.00045.

[29] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, K. Olukotun, Spatial: A language and compiler for application accelerators, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 296âĂŞ311. doi:10.1145/3192366.3192379.
URL https://doi.org/10.1145/3192366.3192379

[30] B. Schrader, S. Reboux, I. F. Sbalzarini, Choosing the best kernel: Performance models for diffusion operators in particle methods, SIAM J. Sci. Comp. 34 (3) (2012) A1607–A1634.