AUTOTUNING BLOCKING FOR WEATHER AND CLIMATE MODELS

Fabrice Dal Farra, Matthias Busenhart, Yannick Niedermayr

ETH Zurich, Switzerland

ABSTRACT

We present an autotuning infrastructure for weather and climate models, implemented in C++. Weather circulation often takes place inside a single horizontal layer. Each layer can therefore be integrated over time by itself, according to the governing equations. We show that the use of blocking on such a layer makes a huge difference in speeding up the computation of the diffusion equation. The autotuner infrastructure allows to search for the best block size which then can be used as a fixed size for the final developed climate model. We show the impact of having such block size fixed during compile-time instead of passing it at runtime.

1. INTRODUCTION

Weather takes place hour by hour and its forecasts use the current atmospheric and oceanic conditions to predict the future meteorological conditions. Climate models on the other side, are an extension to weather forecasting, but analyze long timespans. Their main goal is to predict the average condition changes in a region over a period of time. The climate models are based on mathematical equations using thousands of data points [1]. Computing and solving these equations on a grid is computationally expensive and hence lots of different approaches exist. Storing the full grid (all 3 dimensions) requires a huge memory footprint. However, the weather is evolving in layers around the globe, meaning there is only little exchange between different height levels. This allows computing each layer individually, which on modern computers can be done concurrently on individual cores. In today's climate models, often Fortran code is used (for example in [2] where Python code is built on top of existing Fortran code) and k-blocking (3^{rd} dimension, corresponding to height) is widely used thanks to its low disruption and ease. Since weather model computations are often stencil-based (meaning they act only on a certain region around a specific data point), storing the full plane (ij-direction) is often not needed. This means, one can introduce blocking in either i or j direction, or in both at the same time. This is already implemented in certain domain specific languages, such as GT4Py [3], a Python library which can use GPUs or FPGAs in the background. The benefit of ij-blocking is, that the data used during a single time iteration is kept inside of the cache. Therefore, memory accesses are optimized, decreasing the total runtime of the computation.

Hence, one of the main goals of the project regards the general question if blocking is advantageous compared to non-blocking approaches. Moreover, the intention is to develop an autotuning infrastructure to find the best block size for a given climate model and simulation machine. In C++, having data structure sizes fixed at compile-time versus fixed at runtime can lead to a huge difference in execution time. Therefore, the project will explore the difference between those two approaches and their effect on blocking.

2. MATHEMATICAL MODEL

The transport of energy and matter in fluids is determined by advection and diffusion. An example concerns the transport of salt in the ocean through advective and diffusive processes. These change the density and are thus exerting a strong influence on the large-scale circulation in the ocean. The mathematical descriptions of these transport models is therefore fundamental in climate modelling science [4]. The model that is used throughout this project is the 2-dimensional diffusion equation

$$\frac{\partial u}{\partial t} = \alpha \left(\Delta \left(\Delta u \right) \right) \tag{1}$$

where α is the diffusion coefficient and Δu represents the Laplacian of u. We use a 2-dimensional equation since, as mentioned before, there is only little vertical exchange.

Numerical Approximations. In order to solve Equation 1 numerically, one uses a discretization in both space and time. The time derivative can be approximated with

$$\frac{\partial u^n}{\partial t} \approx \frac{u^{n+1} - u^n}{\Delta t} \tag{2}$$

which is a simple first-order forward Euler scheme. u^n refers to the value of u at timestep n. The discretization of space is done with the following approximation:

$$\Delta u^n_{i,j} \approx \frac{-4u^n_{i,j} + u^n_{i-1,j} + u^n_{i+1,j} + u^n_{i,j-1} + u^n_{i,j+1}}{\Delta x \Delta y} \tag{3}$$

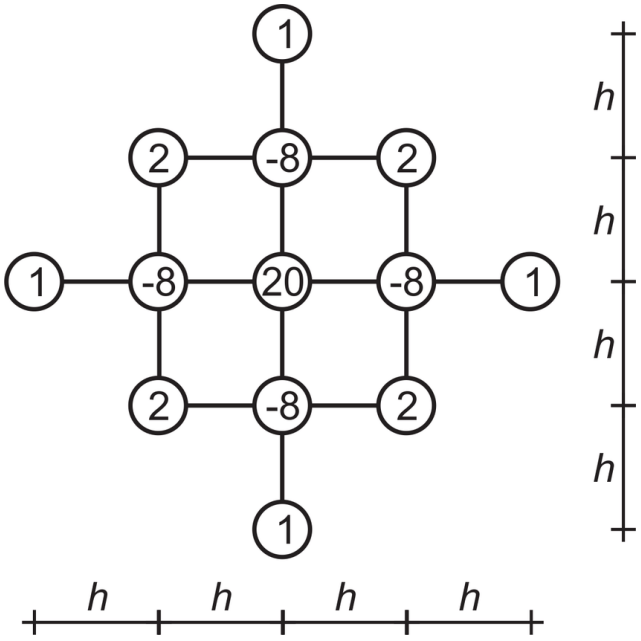

Fig. 1. Resulting 13 point stencil for 2D diffusion equation based on the discretization introduced in Equation 3, for an equally spaced grid with $h = \Delta x = \Delta y$. The numbers represent the coefficients for the terms on the right hand side of Equation 3.

which is an approximation for the Laplacian in space, where i and j denote the current grid point. The right hand side of Equation 1 is simply computing the Laplacian of the Laplacian of u, which means that Equation 3 is applied twice, resulting in a larger stencil, as shown in Figure 1.

If we choose the space discretization in both x and y direction to be the same, Equation 3 simplifies and the full discretized equation results in:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = \alpha \left(\frac{-4u_{i,j}^{n} + u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n}}{\Delta x^2} \right) \quad (4)$$

This equation can now be solved for u^{n+1} iteratively, until the final time t is reached.

In Figure 2, an initial data field is shown, where the whole field is set to zero except in the middle where it is one. Figure 3 shows the diffusion equation applied to the initial data.

Blocking. In order to reduce cache misses, which results from too many data points accessed during a computation, blocking is introduced. During the computation of the 2D-diffusion equation, every ij-layer is independent of all others. This means, that each k-level does only depend on its own. If one wants to compute the Laplacian of a single ij-

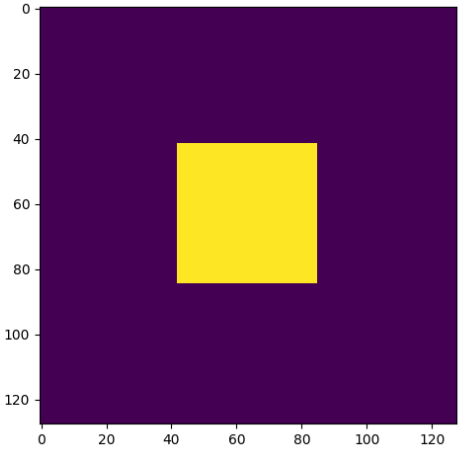

Fig. 2. Initial data on a 128×128 grid.

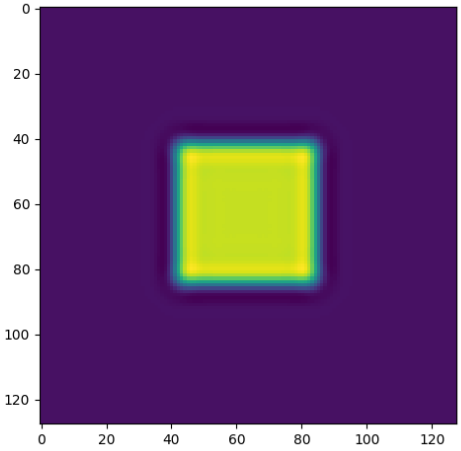

Fig. 3. Final data after 100 time iterations based on the initial data shown in Figure 2.

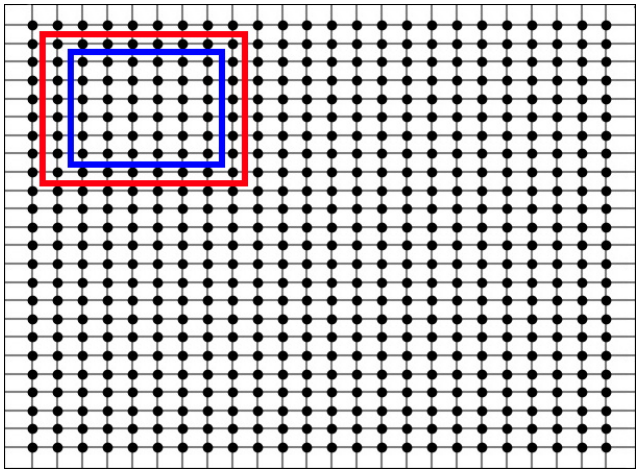

Fig. 4. Illustration of a 6x6 grid block (blue rectangle) with the associated 8x8 temporary field (red rectangle) that contains all data needed for the 6x6 block's stencil computations. The red temporary field requires a halo-ring around it (not shown).

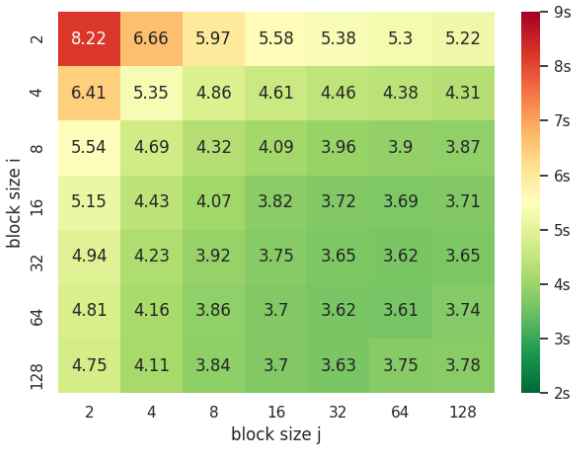

Fig. 5. Runtime measurements for Equation 1, with domain size $128 \times 128 \times 32$, 64 timesteps, average of 4 runs. Blocking is enabled and the block size is defined during runtime.

layer, all values of the layer are used once. However, if the diffusion equation should be solved for a specific amount of time, many iterations of applying the Laplacian operator to the layer are needed. This means, that the full layer must be read and written many times. In the diffusion equation case, for a single timestep, the Laplacian must be computed twice. If the layer is small, this is fine, since the full layer can fit into a fast memory cache and be read again for the second Laplacian shortly after. However, if the layer is bigger and can no longer fully fit into the cache, the second Laplacian must load all data again from a higher level cache (or even from main memory). If one now introduces blocking in the layer, as visible in Figure 4, temporal locality can be achieved. This means, that the subsequent call to the Laplacian can still benefit from the data residing in the fast cache. Special thought must be given to the size of the temporary fields and the boundary values, to get correct results.

3. AUTOTUNER

To find the perfect block size for a specific weather model, this project introduces a so-called autotuner. The autotuner has a multi-dimensional search-space that contains values for i- and j-block sizes. This allows to perform a grid-search over all possible parameters and find the parameters which yield the fastest computation.

Huang, Li and Yao survey in their work [5] different heuristics in order to efficiently find the best parameters in the parameter space. For example, having 3 parameters with 10 possible values each results in a search space of already 1000 parameter configurations. Extensively searching for the best block size for a given problem requires the full equation to be solved (for a specified number of iterations) for each configuration requires a lot of computing power. Therefore, as mentioned in [5], the use of heuristics can drastically reduce computing needs and therefore search time.

4. RESULTS

Figure 5 shows the runtime in seconds for a simple 2D-diffusion execution with blocking enabled. A block size which is too small results in bad performance due to overhead, represented by the top-left corner of the heatmap. The optimal block size for this example lies somewhere around 64 by 64 grid points. This means that the temporary field has a size of 66 by 66 plus additional halo points. This corresponds to an overall size for the temporary field of just over 36 KB. We ran our experiments on Piz Daint, which is an Intel Xeon E5-2690 v3 which has a L1 data cache of 32KB and L2 data cache of 256KB. This means, that our temporary field cannot fully fit into L1. If we consider that we only measured 4 runs and averaged, it is possible that slightly smaller block sizes (for example 32 by 64 grid points) could result in slightly faster code. They would fit fully into L1 cache.

In contrast to Figure 5 is Figure 6. Here, the block size was defined during compile-time. This allows the compiler (GCC 9.3.0 20200312 (Cray Inc.) in our case) to perform more and better optimizations (vectorizations, loop unrolling, etc) which results in faster code. Also, here, the block size 64 by 64 grid points is best.

Figure 7 shows the runtime in seconds for a 2D-diffusion

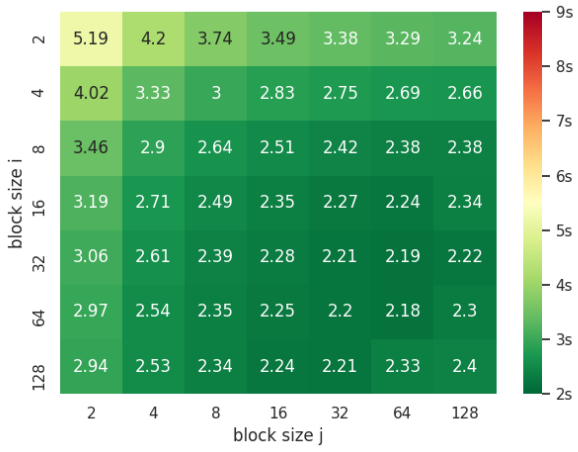

Fig. 6. Runtime measurements for Equation 1, with domain size $128 \times 128 \times 32$, 64 timesteps, average of 4 runs. Blocking is enabled and the block size is defined during compile-time.

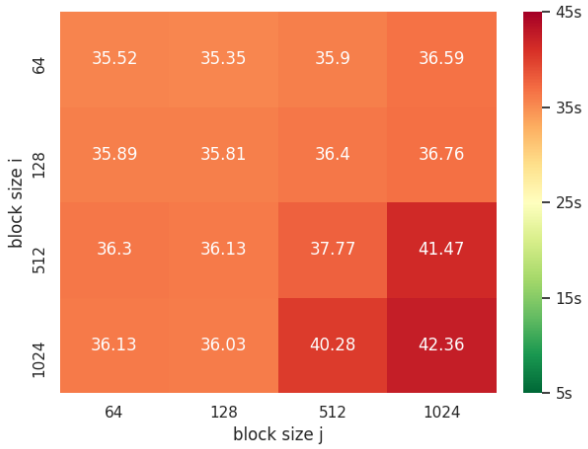

Fig. 7. Runtime measurements for Equation 1, with domain size $1024 \times 1024 \times 4$, 16 timesteps, single run. Blocking is enabled and the block size is specified during runtime.

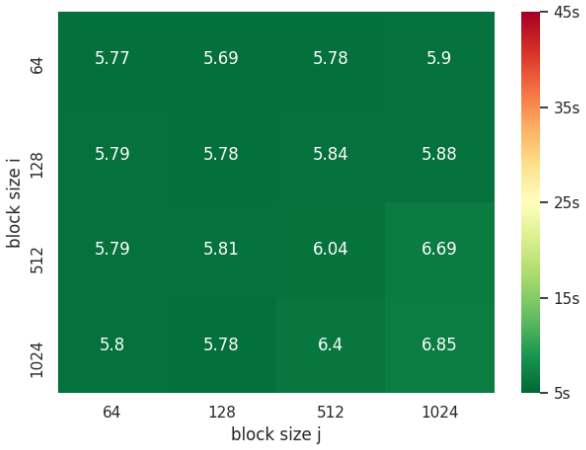

Fig. 8. Runtime measurements for Equation 1, with domain size $1024 \times 1024 \times 4$, 16 timesteps, single run. Blocking is enabled and the block size is specified during compile-time.

	Small Domain	Big Domain
Runtime-Blocking	3.61s	35.35s
compile-time-Blocking	2.18s	5.69s
Double-Stencil	1.74s	4.56s

Table 1. Comparison between blocking for the best parameters presented in Figure 5 to Figure 8. Double-Stencil refers to an implementation using the bigger stencil presented in Figure 1

execution with a larger domain size. This means, that the data is too big to fit into any cache level and therefore resides in main memory. In this case, finding a set of optimal parameters is crucial and can reduce memory transfer by a lot. The optimal block size is no longer in the bottom right, but rather in the upper left corner of the heatmap. This means that large block sizes at some point become suboptimal since they do not fit into a fast cache and therefore introduce more memory traffic. The best block size according to this Figure is 64 by 128, however, smaller block sizes have not been tried.

Also here in contrast is Figure 8, which shows the runtime for compile-time specified block sizes. However, the effect of having compile-time block sizes has a much more noticeable impact on runtime if the domain is bigger. Comparing Figures 7 and 8 yields a speedup of 1.66 for the fastest time, while here, the speedup is 6.2.

We also implemented the 13-point stencil shown in Figure 1 and compared it to the blocking-approach. The results are shown in Table 1. The table shows that avoiding any temporary field wherever possible is even better and can further increase the speed. Our 13-point stencil implementation uses no further optimizations and thus could poten-

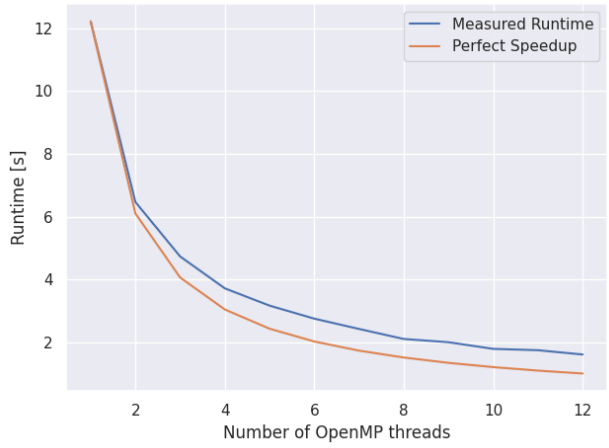

Fig. 9. Strong scaling with OpenMP threads on the k-index. Domain size is $128 \times 128 \times 128$, 1024 timesteps, average of 10 runs. The orange line shows perfect speedup using n cores.

tially be improved even more.

In order to show that also the k-loop can be optimized, Figure 9 is introduced. It shows a strong-scaling plot where OpenMP threads act on the k-loop used for the diffusion equation. This shows that the use of multi-core systems for this simple 2d-diffusion equation is a good choice, since nearly perfect speedup can be achieved.

5. CONCLUSIONS

We have designed a program that can automatically find a set of optimal blocking parameters for a given climate simulation model. Our infrastructure allows us to easy create parameter searches over various value ranges.

The grid search performed by the autotuner program analyzes pairs of i- and j-block sizes for their execution speed of the simulation. The optimal (fastest) parameter pair is typically chosen such that one block fits into the L1 data cache. This result seems relatively independent of the total problem size, as it is adhering to a hardware constant. However, finding this constant is a matter of running the autotuner for every unique hardware setup the weather model should be run on. Similar approaches have been tried in numerical algebra libraries, such as ATLAS [6].

While our autotuner can find the best set of parameters for a huge configuration space, it does require time. It is possible to apply heuristics to the search to find the global optimum faster, however this is out of scope of this project and not incorporated here.

In all experiments that compare result times for precompiled block sizes with those determined during runtime, the precompiled runs are much faster, as is expected. The compiler is can optimizations when compiling with a fixed block size, which results in faster code.

6. REFERENCES

[1] "What are climate models and how accurate are they?," `https://news.climate.columbia.edu/2018/05/18/climate-models-accuracy/`, Accessed: 2021-08-16.

[2] J. McGibbon, N. D. Brenowitz, M. Cheeseman, S. K. Clark, J. P. S. Dahm, E. C. Davis, O. D. Elbert, R. C. George, L. M. Harris, B. Henn, A. Kwa, W. A. Perkins, O. Watt-Meyer, T. F. Wicky, C. S. Bretherton, and O. Fuhrer, "fv3gfs-wrapper: a python wrapper of the fv3gfs atmospheric model," *Geoscientific Model Development*, vol. 14, no. 7, pp. 4401–4409, 2021.

[3] Langwen Huang, "Improving the performance of a domain-specific language compiler," 2021.

[4] Thomas Stocker, *Introduction to Climate Modelling*, 01 2011.

[5] Changwu Huang, Yuanxiang Li, and Xin Yao, "A survey of automatic parameter tuning methods for metaheuristics," *IEEE Transactions on Evolutionary Computation*, vol. 24, pp. 201–216, 06 2019.

[6] R. Clint Whaley and Jack Dongarra, "Automatically Tuned Linear Algebra Software," in *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999, CD-ROM Proceedings.