FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Automated Performance Prediction for Generated PDE Solvers

Meike Blöcher

Bachelorarbeit

Automated Performance Prediction for Generated PDE Solvers

Meike Blöcher

Bachelorarbeit

Aufgabensteller:	Prof.Dr.Harald Koestler
Betreuer:	Sebastian Kuckuk
Bearbeitungszeitraum:	1.2.2019 – 1.7.2019

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30. Juni 2019 . . . .

Contents

Terms and Acronyms

DSL	domain-specific language
FD	finite differences
FLOPs	Floting Point Operations per second
ghost layer	additional layer around the discretized grid for the update of the new stencil
multigrid method	class of iterative solvers
PDE	Partial Differential Equation
RRZE	Regionales Rechenzentrum Erlangen

1 Introduction

In the time of fast computing where new technology and hardware is produced persistently it is important to find software solutions to be portable to different hardware and tackle problems on a more abstract level, meaning not writing specific code for one problem but thinking bigger and getting solutions for similar problems. Instead of spending a lot of time on deeply optimizing code for one specific platform, it might be better to spend it on finding rules for different hardware groups and implementing a solution that is working on similar platforms.

This can be done by building a code generator for the common method to solve the problem which can vary in the implementation for the specific requirements. Such a code generator is developed in the ExaStencils project [2], which is implementing the multigrid method. This method is an iterative solver for Partial Differential Equations (PDEs). It uses grid coarsening, refinement, and a smoothing algorithm to compute the solution. The optimal number of refinement and coarsening steps for the specific PDE can vary for each problem as well as the usage of different smoothing algorithms. These can be defined by the user through input parameters, but the overall structure stays the same which is optimal for using code generation. The project is not only trying to implement the algorithm but also reach highest possible performance on different hardware. Therefore the generated code should be optimized for the specific hardware. The information about the hardware is also given by the user so the generator needs to adapt to the different challenges by having different models for varying hardware.

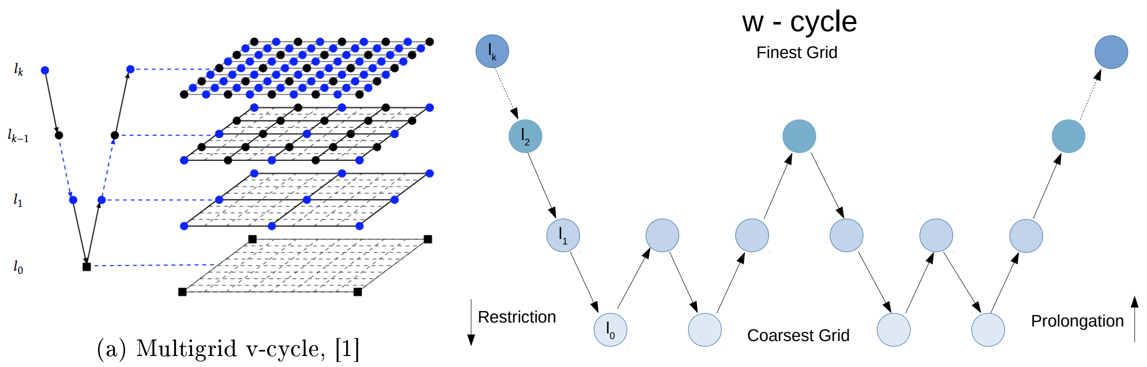
Knowledge about the performance of a software project is always a benefit especially for highly optimized code. Performance prediction not only gives information about how long the code is possibly running which is interesting for scheduling jobs, but can also provide information whether the code is already fully optimized or where it can be improved. This is done when comparing the run times to the estimated times of a perfectly optimized code on that platform.
In the case of the ExaStencils code generation project which is trying to achieve best performance on any hardware leading to exascale performance on the compute clusters in the future, a performance prediction is important to be able to optimize the code up to the last detail. The generation should be able to predict the performance for every hardware type, when given the necessary parameters and information. With this prediction the optimization can be implemented accordingly. For this reason the project needs an accurate performance prediction model to work with the different hardware requirements, and the model and implementation for estimating CPU performance is shown in this work. This can then be adapted for the use on GPU platforms or other hardware in the future. A good performance prediction can give clues for code optimization, by comparing actual runtime or throughput, indicated by Floting Point Operations per second (FLOPs), to the maximal performance of the hardware. Looking at memory usage and improving it or including communication times in the performance prediction in case of parallel programs on multiple nodes can highlight where bottlenecks are and how to improve them.

Following the implementation of the performance prediction, the next step is to improve the actual performance to be matching the predicted times. This can be done in many different ways. There is the option of running the code in parallel with multiple threads on one node using OMP or running the code on multiple nodes. But it is also important to write the code as efficient as possible on the given hardware which can not be neglected when trying to reach the best performance. Small changes in the code structure can have big influences on the overall performance and build the base for a good performance as the mentioned methods are building on the written code. For example the blocking of loops can achieve optimal usage of memory, which means that memory bottlenecks are minimized. This leads to better overall performance on the specific hardware. As there are already parallelization techniques like OMP and MPI implemented in this project, this work is focusing more on the optimal usage of the available memory by looking at cache usage. The general model of layer conditions [5] is used to block the field sizes of the multigrid method.

The structure of this thesis is as follows: In section 2, the ExaStencils project is described in further detail to give a better overview of the project and explain the backgrounds including the multigrid method. A short introduction to similar projects is also given. Afterwards the performance prediction model is introduced in section 3. First the previously implemented model is explained and then the improvements and modifications made to it are shown. In the same section the implementation is explained and a comparison to the existing implementation is given. The next step, improving the performance, is discussed in section 4. This is done with the concept of layer conditions which is ported from an existing model for two-dimensional grids to a three-dimensional one. The section also includes an evaluation of the achieved performance improvement. Afterwards the results of the new implementations are discussed in section 5 and an outlook to further work in this field is given in section 6.

2 Related Work

This section presents the background of the topics investigated in this thesis. After an introduction to the multigrid method and the concept of code generators, the implementation of both in the ExaStencils project is shown. Finally the concept of layer conditions as proposed by a research group at the Regionales Rechenzentrum Erlangen (RRZE) is explained.



(a) Multigrid v-cycle, [1]

(b) Multigrid w-cycle

Figure 1: Illustration of the multigrid method

The term multigrid describes the general class of iterative solvers for PDEs. They are usually working on a discretized domain. To get the most exact solution the domain has to be discretized as fine as possible which leads to very fine grids resulting in higher computational effort. To reduce the needed computation power the grids are coarsened from one level to the next. This is shown in figure 1a, where the number of grid points is bisected between levels l_k and l_{k-1}. The emerging solution on each gird level is smoothed by another iterative solver, for example a Jacobi [10] or Gauss-Seidel [7] algorithm. After having a solution for the coarsest grid the grids are refined again and the solution is interpolated onto the new level. The number of coarsening steps can be varied to get an optimal result.

The multigrid variants implemented in the ExaStencils project are focusing on stencil based smoothing algorithms. Here the grid is structured and the given stencil is used to update the grid points from only neighbor points. This method is often used to get solutions for PDEs describing physical problems, for example heat dispersal or motion of fluids. The most common version of the multigrid algorithm is the implementation of a v-cycle as shown in figure 1a. In this version the solving function for the next coarser grid level l_{k-1} is only called once from l_k. In comparison to that, the w-cycle shown in figure 1b is calling the next coarser grid level twice. This results in a more complex structure which is smoothing out errors better, but increases the computational effort. Which version is best to use depends on the specific problem.

Code generators are often used when a class of problems can be solved in a similar way, but not efficiently by one specific fixed implementation. Driven by input parameters provided by the user, the generator produces code that is tailored to the problem. Because the multigrid methods have the same overall structure but differ in certain details they are ideal for the use of code generators. For example in the ExaStencils project the user can choose smoothing functions and number of refinement steps, among others.

Code generation mostly translates to writing more code than needed for a specific problem solver, as the generator itself has to be implemented. But the benefit is that it can easily be transformed for similar use cases, and putting in a lot of work for a specific example means spending about the same time again for the next solver with a similar problem, as the code optimizations used are only working for this specific example. This means code generation can have a time advantage if similar code is needed. It also gives the opportunity (also used in the ExaStencils project) that user groups with less programming experience can get their solver in already optimized code form. In different configuration levels more or less computer science experience is needed, as input can be given through knowledge files, specific flags or variables for different optimization for example.

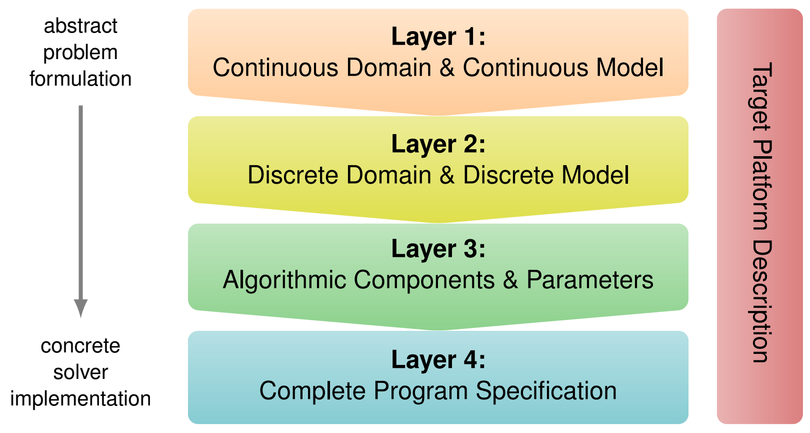

abstract problem formulation
concrete solver implementation
Layer 1: Continuous Domain & Continuous Model
Layer 2: Discrete Domain & Discrete Model
Layer 3: Algorithmic Components & Parameters
Layer 4: Complete Program Specification
Target Platform Description

Figure 2: DSL hierarchy of ExaStencils[2]

ExaStencils is a project of several universities to build a code generator for stencil code, more specific multigrid methods. The goal is to reach exascale performance in the future and be able to run optimal on different types of hardware. Its different abstraction levels help different user groups to get the optimal performance no matter how much knowledge they have about performance optimization. During the project a domain-specific language (DSL) was developed called ExaSlang [3]. DSLs are languages developed for a specific problem in contrast to general purpose languages, like Java or C. ExaSlang is organized in 4 layers, every one special for its use cases. These can be seen in figure 2.

One is dedicated for engineers, one to mathematicians and one to computer scientists. The first layer is handling continuous domains whereas the second layer is working with discrete approximations. Both of these layers are thought to be used by engineers and scientists who have little experience with coding. These layers are giving less options for optimizing the code by hand and are generating a suitable solution automatically. Only in the third layer the implementation of the multigrid method can be modified, which means the user can implement the algorithm as they want it assuming that the user group already knows about the optimal implementation for the specific example. Layer 4 is also allowing adjustments in the code optimization in regard of the performance. This layer is dedicated to computer scientists. [3]

The implementation of the multigrid algorithm in the ExaStenils project needs some explanation for understanding the terms used in the following sections: In the ExaStencils project the smallest level stands for the coarsest grid level and the highest level is the finest one. If an example is running with min 1 and max 7 level it has 6 restriction steps starting at 7 and ending at 1. [2]

Another note for better understanding: The different data structures, like vectors for the right hand side or the array with the grid points itself are each stored in a separate field. This leads to multiple fields which are later referred to in the performance prediction model. Each field has a different size and can have different data types. For example the grid is stored as whole numbers, e.g. `int` or `long`, but throughout the computation the results are in `double` values. Then the size of these elements is different.

A good implementation for performance prediction of stencil code is the software Kerncraft of a research group from the RRZE. They are also working on stencil code and estimating its performance. But the integration of their code into the generator language ExaSlang is quite difficult so a similar approach written in the ExaStencils project was the easier approach. Looking at their work was a good opportunity to get inspiration for optimization and estimation strategies that will be explained in this work. [4]

The blocking concept implemented in this work is based on the layer condition model introduced in the Kerncraft project at the RRZE. It works by computing the needed memory space for the iteration depending on the dimension as this varies when blocking the loop. Their model is only developed for one and two dimensions, so in order to use it for the code generator a cast into 3D has to be made. This new model and the implementations are explained further in section 4. [5]

3 Performance Prediction

As already explained the performance prediction is the first step to find possible optimization areas so it needs to be as accurate as possible. This prediction model is focusing on the memory access prediction since that is the predominant part of the overall performance.

In this section, first the existing prediction model is explained and evaluated, followed by presenting new ideas and the implementation of the new model. Finally the new performance prediction model is evaluated.

3.1 Existing Performance Prediction

The performance prediction is trying to estimate the execution time of the program. The execution time is made up of the time to execute all the computations needed and the time to access the data in the memory.

$$t_{\text{execution}} = max(t_{\text{memory}}, t_{\text{computation}}) * n_{\text{iterations}} \tag{1}$$

This existing performance prediction counts up the time used for the mathematical operations already considering the internal pipeline handling, which means it calculates the number of fused multiply add operations in the function evaluated. This time is compared to the time needed to access the data in the main memory and the maximum is the final execution time, as shown in equation (1). In this implementation it is assumed that the data in the main memory only needs one memory access per field per iteration and that the data is available in the cache after that. This is a very optimistic approach and assumes that the loops are perfectly blocked.

In the multigrid method, for every step only a couple of operations have to be executed, but the data of all the stencil points needs to be accessed. This leads to a longer memory access time compared to the computational time in most cases and as the maximum of both times is evaluated the memory time is the more important one. This means the method is memory bound and thus the focus of the prediction should be on estimating the memory time. The prediction is also only calculating the performance of one grid iteration in the algorithm as that is characteristic. Evaluating every different type of loop in the multigrid method is much more work and would not improve the estimated time significantly. Also the effects of the red-black implementation as in the 3D finite differences (FD) Poisson problem are not considered, in this example the first x dimension loop is only iterating over the even grid points and the second one over the odd ones.

To be able to rate this prediction model and also to have a comparison for the following improvements measurements were made. The runtime of the examples were compared to the predicted times. The basic 3D FD Poisson model was run on the "Woody" cluster at the RRZE with eight threads. A single node with a Skylake v5 processor was used, see [6] for details of this processor. This platform was chosen as it delivers reproducible results.[1]

The graph in figure 3 shows the factor between the measured run times on the system and the predicted times for the mutigrid cycles of these examples. To measure the execution times two timers are implemented. One is measuring the initialization steps only. The other one is measuring the computation steps after the initialization is done. This is including different parts of the algorithm, not just the multigrid cycle function. In contrast, the overall prediction time is computed by the number of iterations times the predicted time per multigrid cycle function. In the example of the 3D Poisson model the multigrid cycle of the regarded level is executed six times. When comparing this time with the runtime, there is always going to be a small error, as the main method is running a little longer than the multigrid cycle itself.

[1] The parameters needed to reproduce the measured results are: On cluster "Woody" a node with 4 cores was used and the program was run with 8 threads. The bandwidth was measured with the STREAM benchmark and set to 32.1 MB/s in the platform file. The cache size is 32 GB and there is only one L3 cache. The variables in the knowledge files are: `poly_optLevel_fine = 3` to block the field sizes, `opt_loopBlocked = true` for getting the prediction as the old model, estimating one memory access per field. `experimental_addPerformanceEstimate = true` was used to generate a file with all the estimated times to have a better overview.

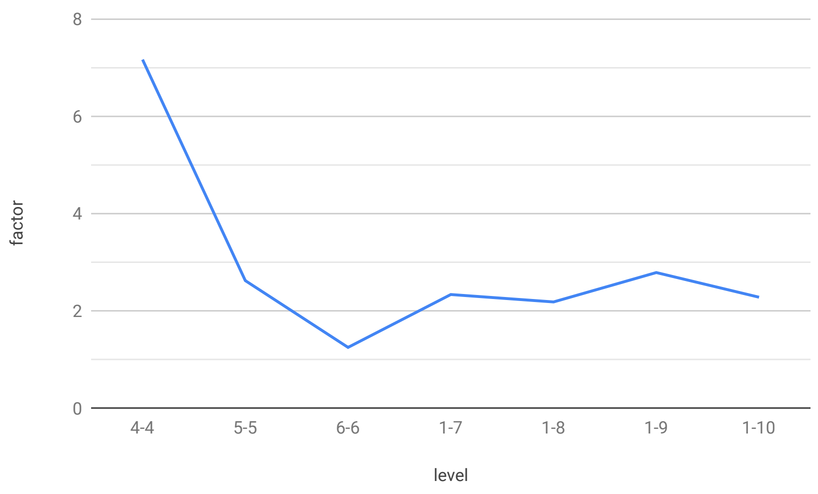


Figure 3: Factor between prediction times of old model and non blocked execution times of different min and max levels, Example 3D FD Poisson

The x axis in figure 3 is showing the min and max grid level of the example version the multigrid method is working on. Examples with the same min and max level are working only on the given grid. The other evaluated versions have the minimum level 1, which is the coarsest grid, and 7, 8, 9, or 10 as the maximum level, corresponding to the finest grid. The higher the maximum level is the more refinement steps have to be made and the more complex the example gets.

The y axis is giving the factor between the runtime of the program and the predicted time. If the estimated time is two times shorter than the executed time the factor is two. For the variants which are computing a multigrid method with grid refinement and coarsening all predicted times are more than two times faster than the actual execution time. This factor even gets worse for more complex problems. On the chosen platform the level 10 multigrid version is the highest computable for this example, because bigger problems run out of memory during the computation. Level 11 would have to store $(2^{11})^3$ data values for the 3D Poisson example.

Ideally for every level combination the error should be 1 but this prediction is too optimistic and the predicted time is mostly shorter than the measured one. This is due to the fact that the prediction is assuming that the code is blocked optimally and only one data value per field needs to be accessed. Additionally the write function is only counted as one memory access, but would more accurately be counted as two. One access is needed for the reading and one for the writing. These problems are adressed in the new model explained next.

3.2 New Model Ideas

For the muligrid method the memory access time is always going to be longer than the time needed for the computations. Therefore the computation has to wait for the data to be accessible.

Hence, the focus lies on improving the prediction of memory access time. This is done by finding the actual number of memory accesses per iteration which is influenced by data being evicted from the cache. In the following different attempts are explained that were considered in the process of developing the new model.

The first attempt was to compute the memory offsets between the stencil points, meaning the amount of data located between the two points in the memory, to see how much cache space is going to be needed for one iteration. Luckily the indices of the data arrays in the generated code are already the memory indices so computing the offset is quite simple. After looking at the generated code of different levels of the 3D FD Poisson example a pattern was emerging of how the offsets are depending on the size of the field. The factor for the amount of data between points of different dimensions in the memory are:

$$d_x = 1 \quad (2)$$
$$d_y = 2^{level} \quad (3)$$
$$d_z = (2^{level})^2 \quad (4)$$

The offsets show, that the data in x direction is stored consecutively in memory whereas in the y dimension, values are spaced according to the size of the x dimension as shown in equation 3. In the z direction the data offsets depend on the x and y dimensions' length.
Additional points might be included for ghost layers depending on user input. Ghost layers refer to the additional needed data around the actual domain, to be able to update the boundary points.

Assuming that each stencil value is loaded into a separated cache line and always a whole cache line is fetched, then the rest of the cache line would be the data needed for the next iterations. Assuming it is a 5 point stencil, this means that there are 5 cache lines needed to store the elements for one iteration, but the data for the next x iterations is already loaded. X would be the number of data fitting into the rest of the cache line. Also assuming that `double` values are regarded, this means the one data point needs 8 bytes and with an average cache line size of 32 bytes this means 4 data points can be stored so in total 4 iterations are available in the cache line after accessing the 5 stencil points needed for this computation. Regarding the whole cache now the number of cache lines in the cache divided by the number of stencil points for one iteration step is the number of iterations possible with one cache and the number of memory access times would be one for every cache line. This represented in the following equation:

$$n_{\text{iterations}} = n_{\text{cacheLines}}/n_{\text{stencilPoints}} \cdot n_{\text{dataperCacheLine}} \cdot 1/n_{\text{fields}} \cdot 1/n_{\text{threads}} \quad (5)$$

Looking at an example of 10 cache lines and the 4 data points per cache line from before would result in a total of 8 possible iterations before the data is overwritten in the cache. This is not including that there are multiple fields and multiple threads sharing the cache, which can be added with the last two variables as shown in the equation (5). Additional computations on what could be reused of the data that is still in the cache were too complicated, especially when not knowing the evicting strategy exactly. This is making the idea not stable enough for the prediction model.

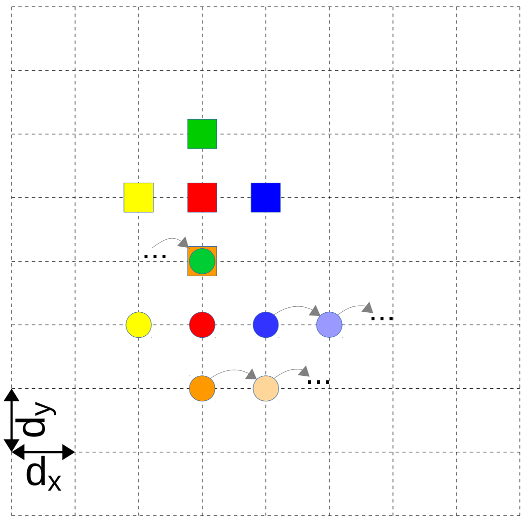


Figure 4: Example 2D stencil with circles as old and squares as current iteration point. The arrows are showing the stencil movement from one to the next iteration

Another point that is not regarded is that if the cache is big enough, data that is needed again might still be in the cache. In case of a symmetric 5 point stencil, as seen in figure 4 the next iteration has the left stencil point, in the figure marked as yellow, already in the cache as the point was the center from the previous iteration step and also the y direction stencil point might still be in the cache when performing d_y iteration steps. To actually compute these factors the evicting strategy is important and the properties of the stencil need to be known. But in general the approach of estimating at cache line level is too specific and looks into too much detail, so another option needs to be found.

The next idea was to normalize the offsets, meaning computing the factors between the stencil points and the center of the stencil, and to find out how much of the data fits into the cache. Then find general rules for when one, two or more data points need to be loaded during the iteration step. The 2D stencil example in figure 4 is showing the current stencil (squares) updating the red center square and the second stencil of n iterations ago (circles). If only one data point should be accessed per iteration the green point has to be accessed, as this is the one furthest away that has never been accessed before and all the other points still have to be in the cache. The orange square is the point lying furthest away from the center point that needs to be still in the cache. Accordingly the point where the orange one was accessed the first time has to be found. The data from there on have to fit into the cache that only one access is necessary. As illustrated in figure 4 the iteration in which the orange square was loaded for the first and only time was in the iteration where the circle stencil was evaluated. The data between these iterations is two times the y offset for this stencil, which is the same as two times the x dimension's length.

Similarly, for two data accesses the cache has to be bigger than the sum of the x and y offset and smaller than the factor just explained for the example with one access. In this case the green and blue point have to be loaded, which means the orange point was last loaded d_x+d_y iterations ago. This is referring to the point left of the orange and beneath the yellow point. But two data points needed could also mean the green and orange one need to be accessed or really any combination of the stencil points. Computing these specific options is already quite complicated for only a simple stencil as in this example. When looking at a 3D stencil it is obvious that these computations get even more complicated as a new dimension has to be thought of and more combinations of data to be accessed together are possible. Furthermore, these rules only work for symmetric stencils and would have to be implemented for every possible stencil by hand in order to generate code for arbitrary stencils. Thus this attempt turned out to be not an option either.

3.3 Implementation

The final approach, which is also computing linearized stencil offsets, is parting the available cache size into blocks, where only the first data point has to be accessed and the rest of the block can be read directly. The size of these blocks varies, depending on how many blocks have to be made and how many threads are working on the cache. For example when the first data point is accessed, this should be the point with the biggest offset, as much data as fits into the cache is accessed. If this means the whole stencil is available, one block and with that one data access per iteration is enough. If it does not fit the space has to be parted into more than one block. The next try would be two. Here the first data point that is not fitting into the first block needs to be accessed again and builds the beginning of the new window. Once the right number of blocks or rather the number of memory accesses needed per iteration is found the data size is computed and returned.

Showing with an example: The available cache block can contain 50 elements and the stencil offsets are 5, 10 and 60. Now the stencil points with offsets 5 and 10 can be got from memory in one go but the one at offset 60 needs to be accessed separately. So there is one window containing elements from 1 to 25 and the next one 60 to 85.

As explained earlier each vector or array needed for the algorithm executed is stored in a different field. Every one of them has to have at least one access per iteration. To take that into account the model has to compute the number of data accesses needed for every field. The solution is straight forward: The cache size is divided into as many parts as there are fields. The different size of the field is not considered, as the computation of that is too much work to be investigated in this work. But for further improvement this can be regarded later on.

Another point that needs to be considered is that the program is working in parallel, so the number of threads working on one node influences the cache usage. To implement that the available cache size is parted into the number of threads working on one cache.

The approach is going to be explained with the help of the code in figure 5.

```
def dataPerIteration(fieldAccesses : HashMap[String, IR_Datatype], offsets : HashMap[String, ListBuffer[Long]]) : Int = {
 if(fieldAccesses.values.map(_.typicalByteSize).toList.distinct.length > 1)
  Logger.error("Unsupported: Not the same Datatype")
 val dataTypeSize = fieldAccesses.values.head.typicalByteSize
 //remodeling old implementation / for optimal blocking factor
 if ( Knowledge.opt_loopBlocked == true){
  return  dataTypeSize * fieldAccesses.keys.size
 }
 // multi thread
 var cacheSize = Platform.hw_cacheSize
 val numberOfThreadsUsed = Knowledge.omp_numThreads
 val numberOfCaches = Platform.hw_numCacheSharingThreads/Platform.hw_cpu_numCoresPerCPU
 if (numberOfThreadsUsed > numberOfCaches){
  cacheSize = (cacheSize/numberOfThreadsUsed)* numberOfCaches
 }

 val maxWindowCount : Int = offsets.map(_._2.length).sum

 for (windowCount <- 1 to maxWindowCount) {
  val windowSize = cacheSize / windowCount / dataTypeSize
  var empty : ListBuffer[Boolean] = ListBuffer.empty[Boolean]
  var windowsUsed = 0

  fieldAccesses.keys.foreach(ident => {
   var sortedOffsets = offsets(ident).sorted.reverse
   val length = sortedOffsets.length
   var i = 1
   do {
    val maxOffset = sortedOffsets.head
    sortedOffsets = sortedOffsets.drop(1).filter(offset => math.abs(maxOffset - offset) > windowSize)
    windowsUsed += 1
    i = i + 1
   } while (i < length && i <= windowCount && !sortedOffsets.isEmpty)
   empty += sortedOffsets.isEmpty
 })
 if (windowsUsed <= windowCount && empty.filter(x => x == false).isEmpty)
  return windowsUsed * dataTypeSize
 }
 return maxWindowCount * dataTypeSize
}
```

Figure 5: Function `dataPerIteration`

The implementation is working with the `HashMap` `fieldAccesses`. This gives information about the data needed for the computation in general: In the `HashMap` the IDs of the different fields are stored as keys and their size and data type are given through the mapped values.

The second input parameter is a list of stencil offsets, which is already given in a linearized form, meaning that it is the actual number of elements between the stencil point in the memory and the center point. The center point has the value 0, the point on the right the value 1, and the one to the left the value -1 for example.

This implementation is only working for problems where the data type in every field is the same. If that is not the case the number of accesses computed per field needs to be already stored with the data size in each loop. This change has to be done in the future if that is needed.

In lines 10 to 15 in figure 5 the available cache size per thread is calculated. This is mainly done with a variable set in the platform configuration. The information on the hardware has to be given by the user, as the code should run well on different tyes of hardware. This size is then divided by the threads working on the same cache.

In line 17 the maximal number of windows, as the blocks are referred to in this section, is identified. This is the number of all offsets, form each field, added together. Next the needed number of windows is computed by increasing the number of windows in each loop by one. As the size of the windows is depending on the number of windows needed, this is computed new in every loop. The `empty` list is needed to check if every field has no offsets left which do not fit into windows yet. The variable `windowsUsed` is counting the windows needed.

After the initialization of all the needed variables the following is done for every field: The linearized offsets are sorted from biggest to smallest, so that the first window is starting at the furthest stencil point. Then, until the list of offsets is empty or until the maximal number of windows is reached, the stencil points which fit into the current window are dropped out of the list, by filtering if the distance between the first point in the window to the current stencil point is smaller than the window size, in line 30. The number of windows used is incremented accordingly. The `empty` list is updated, so that in the next step it is assured that no offset is left and the right number is found. If the loop over the `maxWindowCount` ends without finding a window number, the maximal number is returned, which means that no stencil point has still been in the cache.

Note that the code block in lines 6 to 8 in figure 5 allows to switch the prediction back to the old model. If the flag `opt_loopBlocked` is set to `true`, only one data access per field and iteration is returned, which corresponds to the old implementation. This leaves the option to compare the new version with the old version, and allows the user to choose either model. Currently the flag also switches on the use of layer conditions explained in section 4. This can easily be changed to give the user more options to generate specific parts.

3.4 Comparison to old Model

The new prediction model should estimate the runtimes of the current implementation of the multigrid method better since the memory accesses are calculated more accurately now.

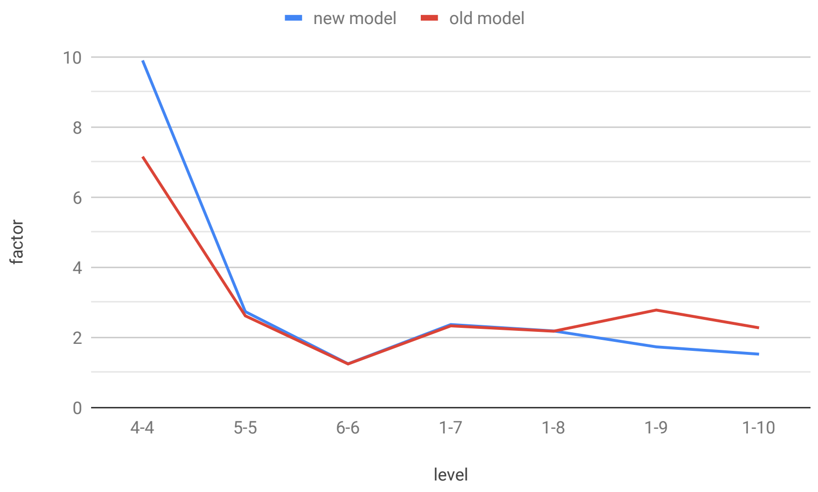


Figure 6: Comparison of the error factors of the execution time compared to old and new prediction times for the 3D FD Poisson example

To evaluate the new model the measured times are compared to both the old and new prediction times. The factors between these times are shown in figure 6.

For the small examples (4-4, 5-5, 6-6, 1-7, and 1-8) no improvements are visible. These examples are so small that they have all the data needed in the cache, like the old prediction model is supposing. For the bigger examples the new model reduces the factor significantly.

Even after this improvement, factors of around 1.8 are still too high for a good prediction. To find the reason where the model is not working good enough more measurements of parts of the multigrid method, in this example the smoothing algorithms, were made.

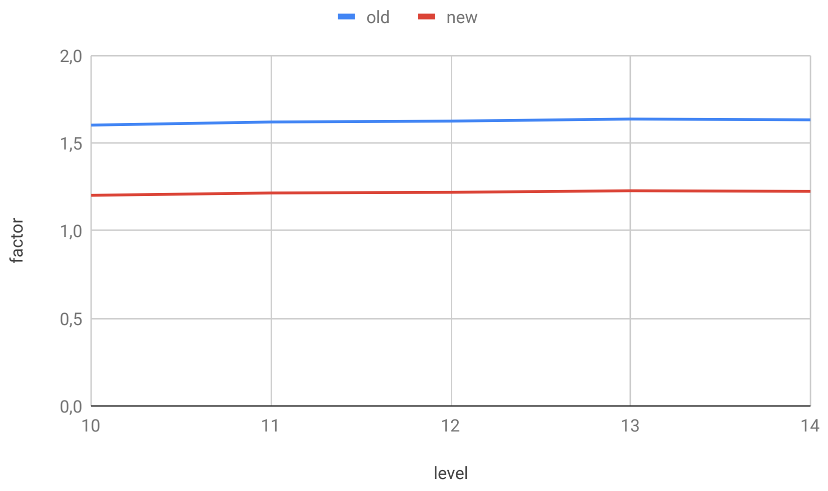


Figure 7: Comparison between the error factors of the old and new prediction times of the implementation of 2D Jacobi method

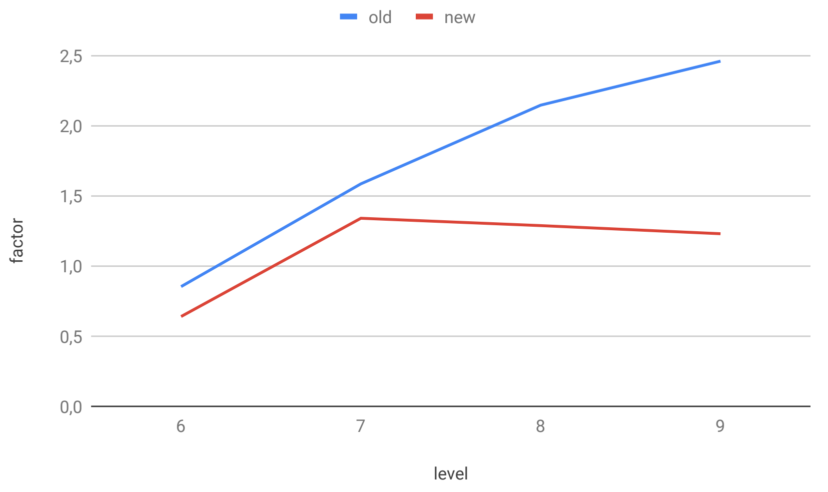


Figure 8: Comparison between the error factors of the old and new prediction times of the implementation of 3D Jacobi method

In figures 7 and 8 results of measurements performed separately for the Jacobi smoothing algorithm are shown. The Jacobi method was chosen since the implementation is simple. Note that in these figures only one level number is given which is referring to both the minimum and the maximum level. In figure 7 measurements for the 2D version of the algorithm are shown. For this example the error factor is reduced equally for all the executed examples. The remaining factor of 1.2 is acceptable, because there are still overhead effects that have to be taken into account. It shows that the model can be accurate but other effects in the more complicated examples are diluting the results.

The measurements in figure 8 show the corresponding prediction error factors for the execution of the 3D Jacobi smoothing algorithm. This example is a more complicated version as an additional dimension has to be regarded. These measurements are also showing an improvement, which is getting even better for bigger problems since they are more strongly affected by the caching effects now considered by the prediction model. Here also the remaining factor of about 1.3 comparing the runtimes with the predicted times is satisfying.

When taking the results from the smoothing algorithm function into consideration, it is clear that the new model is improving the prediction results, but the surrounding multigrid functionality for the examples with the same min and max levels are diminishing the overall improvement. With only one smoother call and no recursive steps the overhead of the surroundings are bigger than the improvement of the prediction.

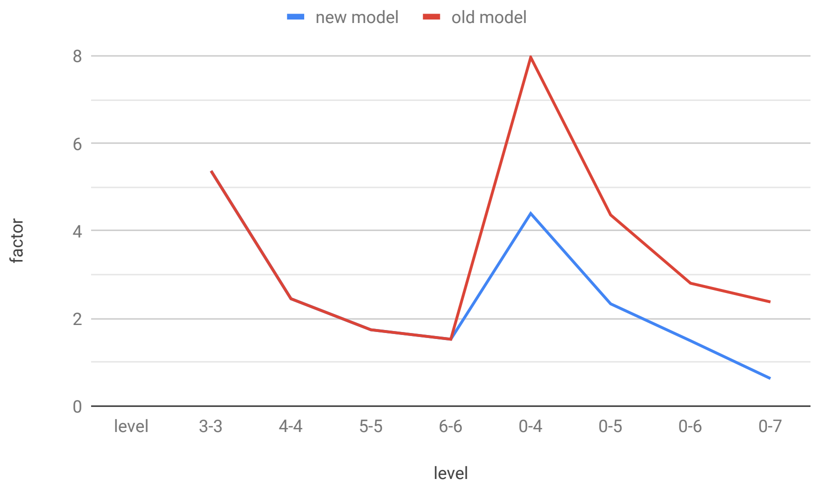


Figure 9: Comparison of the factor between the old prediction times and the runtimes and the new prediction with the runtimes of the 3D FD Stokes example

To verify the results additional measurements with a different example were made. These are shown in figure 9. This implementation is solving the 3D FD Stokes problem. The results from these measurements confirm the explanations from the previous example.

In this example the small problems(3-3 to 6-6) are not showing any improvement, but for the bigger ones an improvement is visible. Even though here is an error occurring for the 0-7 problem, as the predicted time is slower than the measured one.

Overall the new implementation is improving the prediction times for different problems. For further improvement the overall structure of the model has to be changed.

4 Performance Optimization using Layer Conditions

This section is covering the topic of layer conditions which is trying to find the optimal blocking factor for the given problem. The need for blocking the problem size is resulting from the prediction in the last section, which evaluated that more than one data point per iteration had to be accessed. As soon as more than one memory access per iteration is necessary, the algorithm is not working as fast as the hardware is allowing for. This means the goal of the layer conditions is to only need one memory access per iteration per field.

4.1 Idea

Loop blocking is a popular method to optimize memory accesses of a traversal of a field. The concept is that it might be better to not traverse the grid from the bottom left to the top right point by going through the rows step by step, but by splitting the grid into blocks and iterating over the block dimensions. This improves the performance by fitting the necessary data access better to the way data is stored in memory. When working with stencil codes, this is more difficult, as the stencil points might reach out of the block size. This leads to a phenomenon similar to ghost layers, where the data needed for the boundary points of the blocks are needing points from the block next to it. But this phenomenon is not regarded in this implementation.

Layer conditions as the research group from the RRZE [5] is implementing it is taking this concept to compute optimal blocking factors for stencil code solvers. The overall concept is to compute the number of data points that need to be in the cache for the iteration step depending on the size in x direction for a one-dimensional layer condition and additionally in y direction for the 2D concept. The equation (6) to compute this is the following:

$$C_{\text{req.}} = (\sum L_{\text{rel.offsets}} + max(L_{\text{rel.offsets}}) * n_{\text{slices}}) \cdot s \tag{6}$$

$L_{\text{rel.offsets}}$ lists offsets in between all points in all slices. A slice in 1D is a row and a slice in 2D is an array as seen in figure 10.

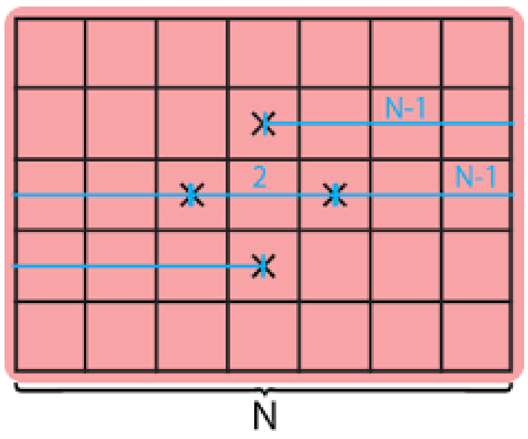


Figure 10: Example of a 2D layer condition slice [5]

In the describing figure the offset list would contain two times the offset $N-1$ plus once the offset 2, leading to a sum of $2N$ and the maximal offset $N-1$. The s in the formula stands for the size of the element, for example 8 bytes. If there would be two slices like the shown one, the list would contain each of the offsets twice. The slices in 1D are computed by the number of rows the stencil is spread over and in 2D every vector or array is one slice. With the example of the multigrid algorithm, one slice is needed for every field. There are fields for every data structure, like the right hand side or the data grid itself.

While for a 2D model only one dimension of the model needs to be blocked, this gets more difficult for a 3D model where there need to be more slices for more dimensions and also necessary blocking factors for more than one dimension. In the 3D model developed in this work one 2D slice for every stencil group that is further than the y dimension size away from the center stencil point is projected to a new slice. For each of those the relative offsets are computed so the previous formula can still be used.

During the modeling the problem came up that equation 6 needs to be solved for the depending x dimension. This would need the implementation of an additional equation solver. Also one equation for two dimensions which need to be blocked is not enough. To avoid the computation of the equation's solution, the concept was remodeled. The needed data for one stencil was computed and the factor of how often it fits into the cache is computed. If the factor is bigger than one it means that no blocking has to be done. If it is smaller, this is the blocking factor overall and has to be mapped onto the two dimensions that are going to be blocked. The blocking dimensions are the two inner loops, so x and y direction for the 3D version. The outer loop does not need to be blocked, as the calculations would follow each other directly anyway.

As the generator should be working for two- and three-dimensional problems the model has to work for both and check which one is needed in each case.

4.2 Implementation

For the implementation at first the structure around the actual blocking function had to be set up. Already included in the ExaStencils generator is an opportunity to set the tile sizes manually. The tiles are the blocks that are looped over after the blocking. These have to be activated with setting the variable `poly_optLevel_fine` in the knowledge file to the highest level 3. Afterwards the tile sizes can be set manually or in the code generation structure. In this work the tile sizes are set to the returned value of the function that finds the blocking factors. This also generates additional loops iterating over the predicted blocks.

```
/* min loop dimension: 513 ,64 ,513 elements */
/* Scop ID: 22;  Statements: S0021 */
for (int _i1 = 0; _i1<=(iterationOffsetEnd_y+8); _i1 += 1) {
#pragma omp parallel for schedule(static) num_threads(8)
for (int _i3 = iterationOffsetBegin_z; _i3<=(iterationOffsetEnd_z
    +512); _i3 += 1) {
for (int _i4 = std::max({iterationOffsetBegin_y,(64*_i1)}); _i4<=
    std::min({(iterationOffsetEnd_y+512),((64*_i1)+63)}); _i4 += 1)
     {
for (int _i5 = ((iterationOffsetBegin_x+1)-((_i4+
    iterationOffsetBegin_x+_i3+1)%2)); _i5<=(iterationOffsetEnd_x
    +512); _i5 += 2) {
fieldData_Solution[8][((265225*_i3)+(515*_i4)+_i5+265741)] =
```

Figure 11: Example Code for blocked loop of 3D FD Poisson level 9

```
/* min loop dimentsion: 513 ,513 ,513 elements */
#pragma omp parallel for schedule(static) num_threads(4)
for (int i2 = iterationOffsetBegin_z; i2<(iterationOffsetEnd_z+513)
    ; i2 += 1) {
for (int i1 = iterationOffsetBegin_y; i1<(iterationOffsetEnd_y+513)
    ; i1 += 1) {
for (int i0 = iterationOffsetBegin_x; i0<(iterationOffsetEnd_x+513)
    ; i0 += 1) {
if ((((i0+i1+i2)%2)==0)) {
```

Figure 12: Example Code for unblocked loop in 3D FD Poisson Level 9

The generated blocked loops can be seen in figure 11. In this example the 3D FD Poisson problem is solved with the multigrid method from level 1 to 9. The code shows that the new blocked tile size in y direction is 64 elements. This is an 8th of the actual grid size of 512. The outer loop in line 3 is looping over 8 blocks accordingly.

Compared to the original generated code, the structure of the inner loops stays the same but some adjustments to the start and end values of the iteration variables are made. Additionally the red-black implementation is moved from an if statement in the unblocked versions to the inner loops. The opportunity to block all three loops is given, so the first counter for the iteration variable name starts at 0 for the loop over the blocks generated in z direction. In this case only the y direction is blocked, so the iteration variables are i1 for the blocks in y direction, i3 for the z direction, i4 for the y direction and i5 for the x direction. To compare between the non blocked and blocked version the code of the identical example without blocking is shown in figure 12.

The implementation of the layer conditions is done in the function `findBlockingFactor`. This is shown in the figures from 13 to 16. These blocks are following each other directly in the given order within the actual program.

```
1  def findBlockingFactor(fieldAcesses: HashMap[String, IR_Datatype],
       fieldSize: Array[Long], stencilOffsets: HashMap[String, ListBuffer[
       Long]]): Array[Long] = {
2  //Multi thread:
3   var cacheSize : Double = (Platform.hw_cacheSize * Platform.hw_usableCache
        )/stencilOffsets.size //Bereich fuer jedes Feld
4   val numberOfThreadsUsed = Knowledge.omp_numThreads
5   val numberOfCaches = Platform.hw_numCacheSharingThreads/Platform.
        hw_cpu_numCoresPerCPU
6   if (numberOfThreadsUsed > numberOfCaches){
7    cacheSize = (cacheSize/numberOfThreadsUsed)* numberOfCaches
8   }
9   var factors  = mutable.HashMap.empty[String, Array[Long]]
10  stencilOffsets.keys.foreach(ident => {
11   var numberOfSlices = 1
12   var stencil = stencilOffsets(ident).sorted.reverse
13   if ( !stencil.isEmpty) {
14    var factorsA : ListBuffer[Long] = ListBuffer(0, 0)
15    val rel0 : Array[Long] = Array(0, 0)
16    var biggest3DOffset : Long = 0
17    var biggest2DOffset : Long = 0
```

Figure 13: Function `FindBlockingFactor`: Initialization

For the given function the input parameters are similar to the prediction function from section 3, but an additional vector of the field sizes is needed. This is for computing the actual number of elements for each tile size. The syntax can be seen in line 1 in figure 13. For the computation of the optimal blocking factors, first the available cache size for sharing threads and fields has to be computed. This is done the same way as explained in section 3. Next the map `factors` is initialized which is later containing the element size of the blocked tiles. The name is resulting from the first implementation idea which was to return only the blocking factors, which is not possible and unfortunately was not changed for this implementation. Line 10 in figure 13 is iterating over every field, so that the following code is executed for every one. The number of slices, which is the counter of how many slices are computed, is set to one. This is the minimum as one data per field has to be accessed per iteration every time. `FactorsA` is designed as a list buffer with 2 elements, to be able to react to the 3D or 2D option. The 3D option can then simply append another element. The factors of each dimension in one field are stored here. Additionally the array `rel0` is initialized here. This structure is needed to store the relative offsets between consecutive stencil points. The variables to store the biggest offset in each dimension are also initialized.

```
1  //3D
2     if (fieldSize.length == 3) {
3      factorsA += 1 * fieldSize(2)
4      // 3D slice in +y direction
5      var stencil_biggery = stencil.filter(_ > fieldSize(1))
6      if (!stencil_biggery.isEmpty) {
7       stencil = stencil.filter(_ < fieldSize(1))
8       stencil_biggery = computeRelativeStencilOffsets(stencil_biggery)
9       relO(1) = stencil_biggery.sum
10      if (stencil_biggery.length > 1) {
11       biggest3DOffset = stencil_biggery.reduceLeft(_ max _)
12      }else{
13       biggest3DOffset = stencil_biggery.head
14      }
15      numberOfSlices += 1
16     }
17     //3D slice in -y direction
18     var stencil_smallery = stencil.filter(_ < -fieldSize(1))
19     if (!stencil_smallery.isEmpty) {
20      stencil = stencil.filter(_ > -fieldSize(1))
21      stencil_smallery = computeRelativeStencilOffsets(stencil_smallery)
22      relO(0) = stencil_smallery.sum
23      if (stencil_smallery.length > 1) {
24       val tmp = stencil_smallery.reduceLeft(_ max _)
25       if (tmp > biggest3DOffset) {
26        biggest3DOffset = tmp
27       }
28      }
29      numberOfSlices += 1
30     }
31    }
```

Figure 14: Function `FindBlockingFactor`: Computation of 3D offsets/ slices

In block 14 of the function the 3D offsets are computed. After checking if this example is three-dimensional the z dimension is set to the size of the field in that dimension. The third dimension is never blocked because the loops would be run through after each other anyways. In the lines from 4 to 16 the stencil points of the slice that is in positive z direction away from the center point are taken out of the list of all the stencils. If there are none this slice does not exist in this example and nothing happens. Then the relative offsets between the stencil points are computed in the function `computeRelativeStencilOffsets`. Next the biggest stencil offset is stored as the `biggest3DOffset` which is needed for the equation to compute the required cache space. The same procedure is done for the stencil points in negative z direction. In this case if the biggest relative offset of this stencil part is bigger than the one from the other 3D slice this is the new biggest 3D offset. Additionally the sum of all the 3D offsets is computed, by storing the results in the `relO` array. Finally, after computing the 3D parts necessary for the formula, a maximum of three slices can be needed.

However, if the example is only two-dimensional the code in figure 14 is not executed and the block in figure 15 is executed next. This code snippet is searching for the biggest 2D offset with the same method as the 3D version. The relative offsets between two consecutive stencil points are calculated, and the biggest one is chosen. In the 2D version the offsets are not taken out of the stencil list. This was only necessary for the 3D version, as these offsets should not be in the list when the 2D version is searching for the biggest offset. This allows reusing the code for the 2D version for the 2D part of the 3D version.

```
//2D
   var rel_stencil = computeRelativeStencilOffsets(stencil)
   if (!rel_stencil.isEmpty) {
    biggest2DOffset = rel_stencil.head
    if (rel_stencil.length > 1) {
     biggest2DOffset = rel_stencil.reduceLeft(_ max _)
    }else {
     biggest2DOffset = rel_stencil.head
    }
   }
   else {
    if (!stencil.isEmpty) {
     biggest2DOffset = stencil.head
    }
    rel_stencil = stencil
   }
```

Figure 15: Function`FindBlockingFactor`: Computation of 2D slices

```
1     val cacheRequired = ( rel0.sum + rel_stencil.sum   + Math.max(biggest2DOffset, biggest3DOffset) * numberOfSlices ) * fieldAcesses(ident).typicalByteSize
2     var ny = (cacheSize / cacheRequired)
3     val nx : Double = 1
4     if (ny > 1){
5      ny = 1
6     }
7     else{
8      nx = Math.sqrt(ny)
9      ny = nx
10    }
11    factorsA(0) = (nx * fieldSize(0)).toLong
12    factorsA(1) = (ny * fieldSize(1)).toLong
13    factors(ident) = factorsA.toArray
14   }
15   else
16    factors(ident) = Array(fieldSize(0), fieldSize(1), fieldSize(2))
17  })
18  return factors(factors.minBy(value => value._2(1))._1)
19 }
```

Figure 16: Function `FindBlockingFactor`: Compute blocking factors

In this last block (figure 16), the formula for getting the needed cache is applied in line 1. Then the factor of how often this data fits into the cache is computed. If the factor is bigger than one, meaning everything fits into the cache at least once, the field is not blocked. Otherwise the factor is parted into the x and y direction, equally in this case by taking the square root of the computed factor on both dimensions, and the field sizes multiplied so that the number of elements is computed. This is then stored into the map `factors`.

After this is computed for every field, the array with the smallest values in y direction is chosen and returned. The result of the implemented function is written into the variable `maxIterations` and the tile sizes are set by the caller of the `FindBlockingFactor` function.

4.3 Evaluation

The initial idea was to find blocking factors for two dimensions, one in x and one in y direction. For the z direction there is no need for blocking, as already explained in the section before. As the model is based on equation (6), the resulting blocking factor n had to be mapped to two dimensions, represented by the variables nx and ny. Different methods for distributing the computed factor onto the two dimensions are evaluated in the following and the best version is explained in further detail.

To realize the different distribution methods, the code in lines 7 to 10 in figure 16 has to be changed. The example code snippet 17 is showing the implementation of a minimum factor of 25% of the original field size, which is only set if the computed factor is smaller than that.

```
else{
if (nx < 0.25){
nx = 0.25
}
ny = nx
}
```

Figure 17: Implementation of different versions to find optimal blocking factor

This particular example can then be varied to the versions introduced in the following. In the cases where only one dimension is blocked, either nx or ny is set to the blocking factor n while the other one is fixed at 1 which corresponds to no blocking at all.

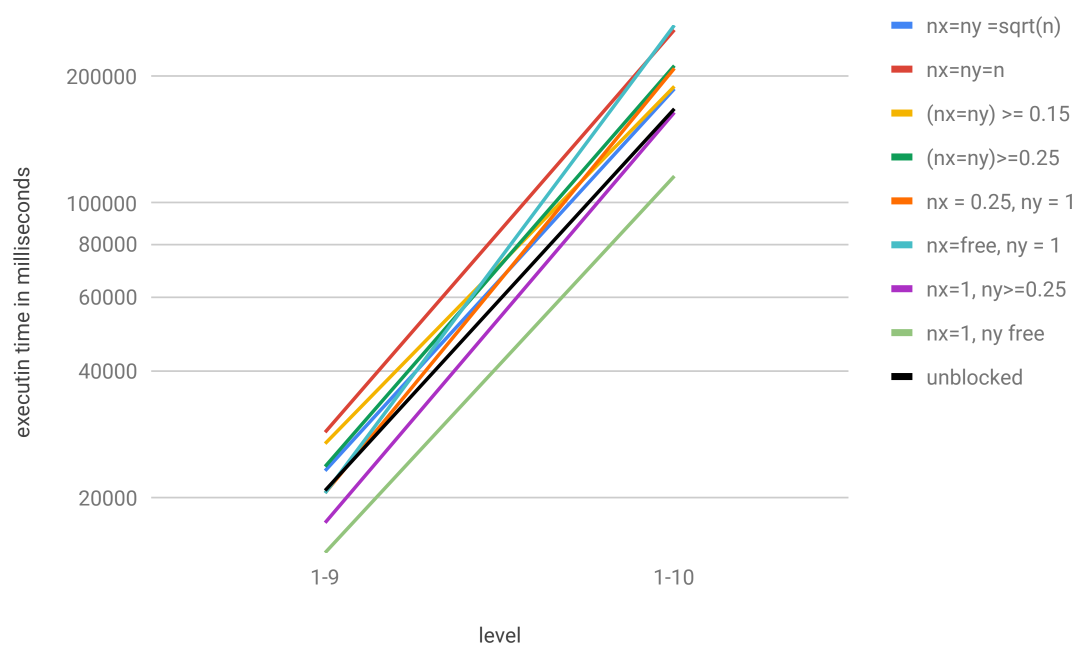


Figure 18: Measured runtimes with different blocking strategies, presented in logarithmic scale for 1-9 and 1-10

In the first attempt nx and ny were both set to the computed factor n. The results of this version can be seen as the red line (nx=ny=n) in figure 18. This attempt increased the runtime instead of improving it.

When loop blocks are very small, the outer loop to iterate over those blocks get larger and increases the runtime. This phenomenon could explain the previous performance loss. To test this assumption the new idea is to set a minimum size for the blocking factor, if the computed factor is too small. This attempt is accepting that for the blocked fields the optimal version of only needing one memory access per iteration is lost in favor of avoiding overhead in the outer loops.

In the next example the minimum factor is set to 15% . This example resulted in a still worse runtime than the unblocked version, but not as slow as the previous attempt. It is illustrated as the yellow line ((nx=ny)>=0.15) in figure 18. Accordingly getting the minimum higher, to 25%, should lead to even more improvement compared to the last results, meaning no worsening. The runtime of this implementation is shown as the dark green plot ((nx=ny)>=0.25). Instead of the suggested improvement this version is slower than the previous 15% one. This can be explained by the fact that the chunks are so big that now a lot of data has to be accessed from the memory plus still some increase through the additional loops.

After evaluating the previous attempts, it was obvious that the resulting overall blocking factor was always different from the original value n. In the first example if both dimensions are set to n the actual blocks are about half as big as intended. The next step was to measure the results for distributing the computed factor equally to both directions, by setting nx and ny to the square root of n. This is shown as the blue plot (nx=ny=sqrt(n)) in figure 18.

As this example was still worse than the unblocked runtime but better than previous versions, the next test was to leave out the blocking in one dimension. This should lead to an improvement because only one additional loop is generated instead of two. First only the x direction is blocked. For this example ny was set to 1, no blocking, and nx to a minimum of 25%. The results of this, seen in 18 as the orange line (nx=0.25,ny=1), were even worse than the previous attempts in the level 10 example. For the level 9 version the factor computed is still a quarter, so the results show, that for that level the computation is as good as the non blocked version.

To test the impact of the computed blocking factor in only one direction, without limitation, nx was set to n and ny is not blocked at all. The results are the light blue line (nx=free,ny=1) in the plot (figure 18). Here the runtime is the same as the non blocked code for level 9 but the worst in the level ten example. This is not a satisfying result, so a further test in only the y directions has to be made.

The same attempt as for blocking the x direction was implemented. ny was set to 0.25 if the computed results were smaller and nx is always 1. The results are shown in figure 18 as the purple (nx=1,ny>=0.25) line. This attempt lead to a slight improvement in runtime compared to the non blocked version, which is shown by the black line.

To underline the results from the last two attempts, that the x dimension blocking is leading to worse results, another run with only y blocking was started. This time ny was set to the computed blocking factor with no restrictions. This lead to even better results than the ny = 0.25 version. The plot is the light green line (nx=1,ny=free) in figure 18. The runtime is almost 30% faster than the original measurements with no blocking. Therefore the results for blocking in y direction are the best ones and the factor computed seems to work well after all.

The comparison between the best and worse runtime of some of the previous versions can be seen in more detail in figure 19 for the 3D FD Poisson model with min level 1 and max level 9 or 10. Here the runtimes are given in milliseconds on the y axis.

The results show that blocking in both x and y direction is leading to a performance loss most likely because executing two more loops is having a bad influence on the runtime. Also a blocking in x direction does not improve the performance. This can be explained through the memory alignment of the data points in the x loop. When blocking that direction this alignment is broken and bigger "jumps" between memory points have to be made. Whereas in y direction the memory is not aligned anyways and the traversal in z direction is helping that effect.

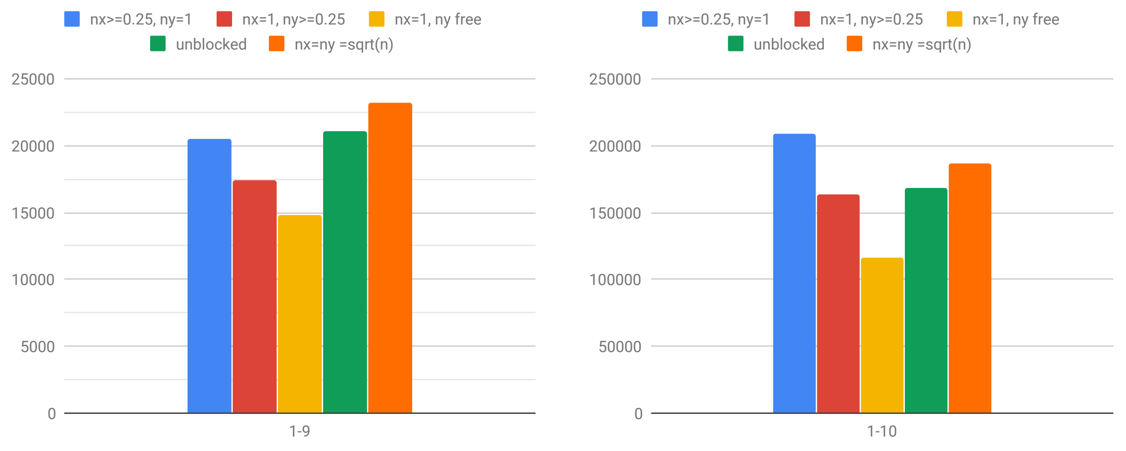


Figure 19: Detailed plot of execution times of best and worst blocking examples compared to the unblocked implementation for level 9 on the left and level 10 on the right

To evaluate the results of the blocking a comparison to the old predicted times, which have been assumed to be optimal, are made. Furthermore the cooperation between the prediction and the blocking model are tested.

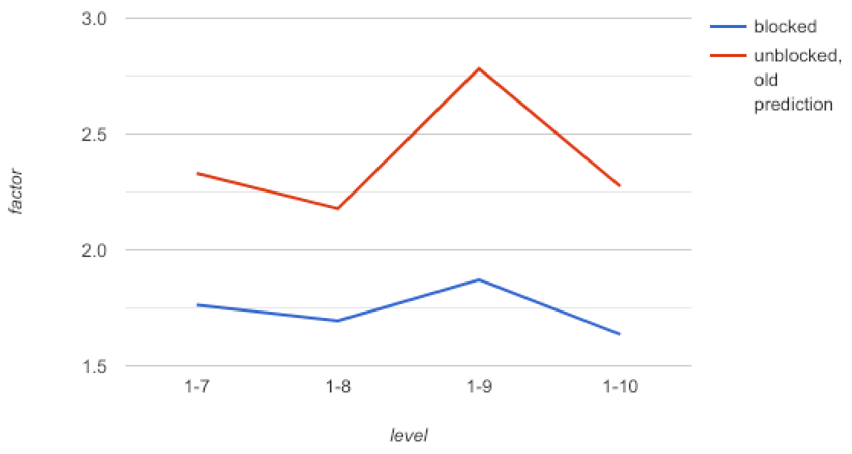


Figure 20: Comparison of factors between unblocked or blocked implementation and performance prediction for 3D FD Poisson

The implementation of the layer conditions was aiming to find the optimal blocking factor to have only one data accessed per iteration per field. This time was predicted by the old implementation of the performance prediction. In order to evaluate the new performance the times of the blocking model with only the y direction blocked with no restrictions is compared to the old non blocked implementation and prediction and can be seen in figure 20. The factor between the old prediction and the runtimes of the blocked model are around 1.8 to 1.7. These factors are an improvement to the ones with the old implementation where they are at a factor of 2.3. That the new runtimes are still not reaching the predicted times can be explained the same way as the results in section 3.4. The prediction times and measured runtimes are not evaluating the same functions and this leads to a divergence. To understand whether the factor between the predicted time and the measured execution time is because of the measuring problems or because the blocking model is not working perfectly, additional measurements would have to be made. Here the different parts of the multigrid method have to be executed individually to see where the prediction is having errors. This has to be assumed now on the base of the results in section 3.4.

Additionally the blocking is only used in one specific part of the overall multigrid method, while the remaining implementation is still unblocked. Thus the overall improvement is limited by the share of the computational load of those parts.

For the evaluation of the prediction model and the blocking implementation together, the prediction has to be done after the blocking factor is included in the loop size. Then the prediction model should evaluate the times of the blocked model.

However, the predicted times of the blocked code as computed after the blocking is applied are a lot smaller than the actual runtimes. For the small problem examples no difference can be seen, but for the bigger problems, where the blocking is actually changing the computation, the predicted times are up to a factor of 14 away from the actual runtimes for the optimal blocking example in y direction. For this example the emerging factors are analyzed further.

For the level 9 example the factor between the estimated time and the blocked execution time is 10 and for the level 10 example even 14. These numbers all refer to the best example of blocking only in y direction with the factor computed. In the level 9 generated code 8 blocks are generated and in the level 10 example 32 blocks. That the factor between the predicted and measured times rises with the number of levels as well as the number of blocks generated indicates that the prediction model is not taking the outer loops over the generated blocks into account. For this reason the prediction model should be changed to also work correctly with the blocking model.

For now the prediction model does not work together with the blocking model, but the old prediction is estimating the performance for the blocked loop satisfyingly.

5 Conclusion

The goal of this work was to evaluate the existing performance prediction model of the multigrid method in the ExaStencils project and improve it to provide a better estimate of the overall runtime. In the second step the performance of the generated code for the stencil evaluation should be improved to get closer to the estimated runtimes of the optimistic prediction model.

For the prediction model improvements, the evaluation of the existing model showed that the memory access behavior had to be modeled in more detail. This was done by considering the cache usage during applying the stencil. As a result the improved model is reducing the error between the predicted and measured runtime from a factor of 2.3 to 1.5 in the best case, as shown in section 3.4. The origin of the remaining factor has to be evaluated in future work in order to optimize the model further. This significant improvement applies to the complex examples with large grids. For the small examples however, the new model does not change the prediction, as the needed data fits already into the cache. Even for the smaller examples the investigations on the smoothing algorithms showed that the model is predicting the times more accurately, but the overhead of the surrounding multigrid implementation is masking these improvements. To avoid this, the overall structure of the model needs to be adapted.

In the second half of this thesis, the performance was optimized by using loop blocking. This included implementing a model to find the optimal blocking factor, which is based on the concept of layer conditions. As shown in section 4.3, this blocking implementation achieved a significant improvement since the overall runtime was reduced by about 30% compared to the unblocked version. Before the implementation in the code generator the user had to manually specify blocking factors for the different levels, which is a tedious task. In contrast the blocking model is automatically finding the blocking factor for each needed level and instructs the code generator to produce blocked loops. Even though the approach of using the blocking factor on only one dimension leads to this significant improvement, it is possible that a better mapping of the factor onto the given dimensions could be found, leading to further performance gains.

Due to the introduction of the performance optimization, neither the optimistic nor the improved performance prediction model are estimating the resulting runtimes accurately. The optimization can not fully reach the optimistic prediction executed before blocking, while running the prediction after blocking does not take the blocking loops into account. To avoid this the performance prediction model should be adapted in the future to ensure blocking is considered. Both cases individually improve the existing prediction times, but a collaboration of both would lead to even better results, as the performance improvement would be visible in the prediction.

6 Future Work

The work presented in this thesis is the first step into a wide area of possible performance optimizations. Some improvements have been made, other topics have been discovered where work on the implemented models still has to be done. Additionally for the project to work optimally on any given hardware, new models for the use on GPUs and multiple nodes, which is using MPI, could be implemented.

After the new implementations of the prediction model and the layer conditions, there is still room for more improvement. First of all the performance prediction model is not yet implemented in a way that it works for examples where the fields have different data types. To be able to run these examples in the future the implementation has to be added. For this the sizes of the available cache space for each field need to be adapted.

There are also still some options left for the performance improvement on one node. As seen in section 3.4 the model works well for the overall multigrid algorithm but for the prediction of the single level versions a better model needs to be implemented that is also counting the time for the structure around the smoother algorithm. To find the exact parts of the multigrid method where the error is occurring new measurements with only specific parts of the multigrid method need to be done. These results can lead to further improvement of the prediction accuracy. These problems of not predicting the time right for small problems already existed for the old version of the prediction model but have not been noticed because the factor between the runtime and the predicted times have been so big.

Analyzing the effects of predicting the performance for the different cache levels can also help understanding the problems of the presented model better.

The implementation of the layer condition was successful, but there are still more improvements possible. As already explained in section 4.3 a different model had to be implemented, if the factors between the predicted time and the runtime are only due to more than one memory access. As this error factor can also be due to the fact that the model itself is not predicting perfectly, the measurements made in that concern should give clues for the blocking model as well.

Furthermore, the given implementation with blocking in only the y direction was the best version found in this work. But a better implementation could still be found through additional detailed measurements to explain where the factor is actually coming from. The explanations given in section 4.3 are mostly based on assumptions.

Another point that can help improve the performance in the future would be an implementation that is also regarding the additional memory accesses that are necessary for the boundary points of the generated blocks, similar to ghost layers. This is not considered at all at the moment but could lead to additional improvement for problems that need to be parted into many blocks.

For the use on multiple nodes the parallelization with MPI provides a big speedup. But it comes with a price. The communication overhead can get very high and take a lot of time. For this a function to evaluate the time to send and receive messages and the additional needed memory for the ghost layers has to be added to the model. This should be considered in the performance prediction. Some ideas for possible models in this topics are explained:

The implementation of the layer conditions could be used to find out the optimal blocks which can be distributed over the nodes. The number of nodes used could be chosen as the optimal number of blocks computed by the layer condition function. This distribution can lead to a better usage of memory and provides a proposal on how many nodes or a multiple of this number should be used.

Furthermore the result can lead to better usage of memory as the field is separated into chunks more suited for the communication. For this the model could be extended into a model for an optimal blocking for the communication between the nodes. The consideration of the separated main memory makes that more difficult as the chunks need to be computed on that specific node. A lot more plans have to be done before computing. When executed on multiple nodes, also a blocking in x direction for the separation on the different nodes could have positive effects. Then the implementation on each node has to check for optimal block sizes in y direction as well. For that the implemented function can be reused.

GPUs have a different hardware and memory structure compared to CPUs. This means that the prediction model would need to be adapted to work with the memory structure of the GPU. Its memory might be considered a kind of L3 cache as the GPU gets the data from the main memory as well. The memory usage would need to be analyzed and instead of parting the cache into chunks for every thread and every field, it could be parted into the number of computing cores.

References

[1] W. Feng, "Projects for Fun: 6. Geometric and Algebraic Multigrid Solver." URL: http://web.utk.edu/~wfeng1/research.html. Last accessed on 2019-06-28.

[2] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt, "Exastencils: Advanced stencil-code engineering," in *Euro-Par 2014: Parallel Processing Workshops* (L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, eds.), (Cham), pp. 553–564, Springer International Publishing, 2014.

[3] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich, "Exaslang: A domain-specific language for highly scalable multigrid solvers," in *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), New Orleans, LA, USA*, November 2014.

[4] J. Hammer, J. Eitzinger, G. Hager, and G. Wellein, "Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels," in *Tools for High Performance Computing 2016* (C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M. Resch, and W. Nagel, eds.), (Cham), pp. 1–22, Springer International Publishing, 2017.

[5] J. Hammer, "Layer Conditions." Website. URL: http://rrze-hpc.github.io/layer-condition/, last accessed 2019-06-28.

[6] Intel, "Intel® Xeon® Processor E3-1240 v5." Website. URL: https://ark.intel.com/content/www/de/de/ark/products/88176/intel-xeon-processor-e3-1240-v5-8m-cache-3-50-ghz.html, last accessed on 2019-06-28.

[7] M. Stüzekarn, "Jacobi- und Gauß-Seidel-Verfahren, Jacobi-Relaxationsverfahren." Presentation in numerical proseminar, 2015. URL: https://www.math.uni-hamburg.de/home/hofmann/lehrveranstaltungen/sommer05/prosem/Vortrag5.pdf, last accessed on 2019-06-28.

[8] B. Barney, "Message Passing Interface (MPI)." Website, May 2019. URL: https://computing.llnl.gov/tutorials/mpi/, last accessed on 2019-06-28.

[9] S. Mittal, "A Survey of Techniques for Managingand Leveraging Caches in GPUs." Website. URL: https://www.academia.edu/6940614/A_Survey_of_Techniques_for_Managing_and_Leveraging_Caches_in_GPUs, last accessed on 2019-06-28.

[10] Encyclopedia of Mathematics, "Jacobi method." Website, October 2014. URL: https://www.encyclopediaofmath.org/index.php/Jacobi_method, last accessed on 2019-06-28.