MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures

Tobias Gysi
Dep. of Computer Science
ETH Zurich
tobias.gysi@inf.ethz.ch

Tobias Grosser
Dep. of Computer Science
ETH Zurich
tobias.grosser@inf.ethz.ch

Torsten Hoefler
Dep. of Computer Science
ETH Zurich
htor@inf.ethz.ch

ABSTRACT

Code transformations, such as loop tiling and loop fusion, are of key importance for the efficient implementation of stencil computations. However, their direct application to a large code base is costly and severely impacts program maintainability. While recently introduced domain-specific languages facilitate the application of such transformations, they typically still require manual tuning or auto-tuning techniques to select the transformations that yield optimal performance. In this paper, we introduce MODESTO, a model-driven stencil optimization framework, that for a stencil program suggests program transformations optimized for a given target architecture. Initially, we review and categorize data locality transformations for stencil programs and introduce a stencil algebra that allows the expression and enumeration of different stencil program implementation variants. Combining this algebra with a compile-time performance model, we show how to automatically tune stencil programs. We use our framework to model the STELLA library and optimize kernels used by the COSMO atmospheric model on multi-core and hybrid CPU-GPU architectures. Compared to naive and expert-tuned variants, the automatically tuned kernels attain a 2.0–3.1x and a 1.0–1.8x speedup respectively.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Optimization

Keywords

stencil; tiling; fusion; performance model; heterogeneous systems

1. INTRODUCTION

Stencil computations on regular domains are one of the most important algorithmic motifs in embedded, high-performance, and scientific computing. Applications range from climate modeling [4], seismic imaging [8], fluid dynamics, heat diffusion and electromagnetic simulations [14] through image processing [11] to machine learning. Given their importance, numerous optimization strategies [1, 5, 10] and domain-specific languages [2, 11, 15] exist. Yet, most of these schemes consider the optimization of a single stencil in isolation. Many applications, however, require nested stencils [18] that are applied in succession. The data dependencies of these nestings can form complex directed acyclic *stencil graphs* where multiple stencils need to be optimized in tandem to achieve highest performance.

Stencils programs perform element-wise computations on a fixed neighborhood called the stencil. Such stencils often have low arithmetic intensity because they have a fixed number of operations per loaded value. The biggest challenge is to map stencil programs to modern architectures with a growing gap between memory and compute bandwidth. Such architectures require *data-centric optimizations* that arrange data accesses to efficiently use the available memory bandwidth. Complex stencil graphs can be optimized using various techniques such as loop fusion, tiling, and various communication strategies on subgraphs. We model all possible combinations of optimizations for a particular stencil program (graph) as an *application algebra* and apply mathematical optimization techniques to find the best combination specific to an abstract hierarchical machine model.

Since typical stencil programs contain hundreds of stencils arranged in paths with dozens of stages and several input arrays, manual tuning of all options is infeasible. In fact, the number of stencil program variants is usually exponential in the number of stencils. In addition, the optimal stencil program variant is specific to each architecture. We show how to fully automate the optimization and implement it in our open-source[1] tool MODESTO, a **MO**del **D**riv**E**n **ST**encil **O**ptimization framework.

We demonstrate the efficacy of our method using the real-world application COSMO [4], a numerical weather prediction and regional climate model used by more than 10 national weather services and many other institutions. The dynamical core of COSMO, a central part of its implementation, applies more than 150 stencils, each operating on 13 arrays on average. This most performance-critical code has been rewritten using the STELLA library and was carefully tuned by experts for optimal performance. MODESTO-optimized stencil graphs match or improve upon the expert-tuned code by a factor of 1.0-1.8x. This demonstrates how our technique enables next generation stencil libraries that completely abstract optimizations from the library interface. Hence, we are able to improve usability as well as performance portability compared to state-of-the-art stencil libraries such as STELLA [6] or Halide [11]. In summary, we make the following contributions:

- We introduce a set of data-centric code transformations, an algebraic formulation of the transformation space, and a compile-time performance model that facilitate the automatic optimization of complex stencil programs.

[1] http://spcl.inf.ethz.ch/Research/Parallel_Programming/MODESTO/

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.

ACM 978-1-4503-3559-1/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2751205.2751223.

- We evaluate our approach by modeling the optimization of stencil codes written using the STELLA library and successfully tune kernels of a real-world application.

- We formulate the automatic tuning of stencil programs as a mathematical optimization problem and solve it using dynamic programming techniques.

2. STENCIL ALGEBRA

Although the stencil motif appears in a wide range of codes from various application domains, common patterns can be identified. Using them, we introduce a stencil algebra that formalizes stencil computations and facilitates their analysis and optimization.

2.1 Definition of a Stencil Program

The following core elements describe a stencil program:

A **field** F defines a dense, multi-dimensional and commonly hyperrectangular set of data values, which can be read and modified.

A **stencil** S is a computation that derives a value located in an output field from a set of input field values located within bounded distance to the output value. It is described by the triple (ops, out, in). The first element, ops, specifies the (possibly approximated) computational cost of executing this stencil. The second element, out, is the output field of the stencil. The third element, in, is a set that defines the input elements of the stencil. The elements of the input set are so-called "named vectors" that are named according to the field the input is read from and the vector itself describes the location of the input element as a relative offset to the position of the element the stencil computes. The set of input elements in is not allowed to contain elements of the output field. We define an example stencil s that computes the value $F_0(i,j)$ from the inputs $F_1(i,j)$, $F_1(i,j+1)$ and $F_2(i,j)$ with 5 computational operations using the following notation: $s := (5, F_0, \{F_1(0,0), F_1(0,1), F_2(0,0)\})$

A **stencil program** $P = T \cup O$ consists of a set of temporary stencils T as well as a set of output stencils O, where the results computed by the output stencils form the result of the stencil program, but the results of the temporary stencils are not made available outside of the stencil program. All stencils and fields have the same dimensionality.

The program definition just introduced is formulated minimalistically and with a strong focus on stencil graphs. As a consequence, it omits aspects that in the context of our article are of limited importance, e.g., boundary conditions, variable input field dimensionality, as well as complex dynamic control flow. However, programs which use such concepts can in many cases still be modeled. For example, stencils with varying input sets, due to the use of special boundary conditions, can often be modeled with an over-approximated input set and iterative stencil computations can be modeled by (partially) unrolling the relevant time loop.

2.2 Example

We now present an example stencil program which is derived from a horizontal diffusion kernel used by the COSMO atmospheric model [4]. We define the stencil program P_{hd} in terms of the temporary stencils s_{lap}, s_{fli}, and s_{flj} necessary to evaluate the output stencil s_{out}. A data dependency either refers to an input field loaded by the stencil program, such as in or wgt, or to a temporary field computed by the corresponding temporary stencil,

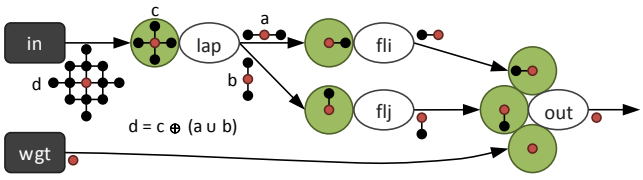


Figure 1: Horizontal diffusion dependency graph annotated with stencil (c) and stencil program (a, b, and d) access patterns

```
1  // Allocate temporary storage
2  Field[double] lap(ibegin,iend),
3    fli(ibegin,iend), flj(ibegin,iend);
4  // Apply the lap stencil
5  for(int i=ibegin-1; i<iend+1; ++i)
6    for(int j=jbegin-1; j<jend+1; ++j)
7      lap(i,j) = -4.0 * in(i,j) +
8        in(i-1,j) + in(i+1,j) + in(i,j-1) + in(i,j+1);
9  // Apply the fli stencil
10 for(int i=ibegin-1; i<iend; ++i)
11   for(int j=jbegin; j<jend; ++j)
12     fli(i,j) = lap(i+1,j) - lap(i,j);
13 // Apply the flj stencil
14 for(int i=ibegin; i<iend; ++i)
15   for(int j=jbegin-1; j<jend; ++j)
16     flj(i,j) = lap(i,j+1) - lap(i,j);
17 // Apply the out stencil
18 for(int i=ibegin; i<iend; ++i)
19   for(int j=jbegin; j<jend; ++j)
20     out(i,j) = wgt(i,j) *
21       (fli(i-1,j) - fli(i,j) + flj(i,j-1) - flj(i,j));
```

Figure 2: Naive implementation of the simplified horizontal diffusion example used by the COSMO [4] atmospheric model

such as fli, flj, or lap:

$$s_{lap} := (5, lap, \{in(-1,0), in(1,0), in(0,-1), in(0,1), in(0,0)\})$$
$$s_{fli} := (1, fli, \{lap(1,0), lap(0,0)\})$$
$$s_{flj} := (1, flj, \{lap(0,1), lap(0,0)\})$$
$$s_{out} := (5, out, \{fli(-1,0), fli(0,0), flj(0,-1), flj(0,0), wgt(0,0)\})$$
$$P_{hd} := \{s_{lap}, s_{fli}, s_{flj}\} \cup \{s_{out}\}$$

Figure 1 illustrates the data flow of the stencil program using a directed graph, whose black and white nodes represent input fields and stencils respectively. Arrows that do not point to a node and consequently exit the stencil graph model the outputs of the stencil program. A directed edge in the graph corresponds to a flow dependency between two nodes. We annotate each incoming edge of a stencil with the access pattern necessary for a single stencil evaluation. For instance, a single evaluation of the lap stencil accesses the in field at the five offsets shown by c. In addition, we annotate all outgoing edges of a stencil or an input field with the accumulated access pattern necessary to evaluate the out stencil at a single position. E.g., the lap stencil is evaluated at the positions defined by the union of the sets a and b. We compute the accumulated in field access pattern d as the Minkowski sum $d = (a \cup b) \oplus c$, with $a \oplus b = \{a' + b' \mid a' \in a, b' \in b\}$. Figure 2 shows a naive implementation of the horizontal diffusion kernel, which executes each stencil using a separate loop nest. While such an implementation may be straightforward to write, it is not efficient in terms of data locality, memory usage, or parallelism.

2.3 Data Locality Transformations

To improve the data locality of stencil programs, we discuss code transformations that combine loop tiling and loop fusion. While tiling sub-divides the loop domain into typically hyperrectangular tiles of limited size, fusion substitutes a sequence of loops by a single loop. Applied to stencil codes, we divide the stencil evaluation

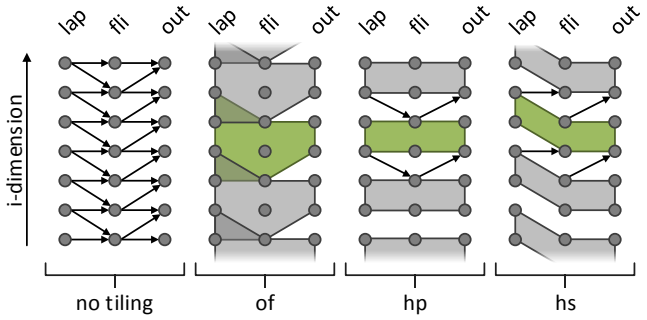


Figure 3: Tile shapes (shaded) for different tilings applied to a subset of the horizontal diffusion example projected to the i-dimension

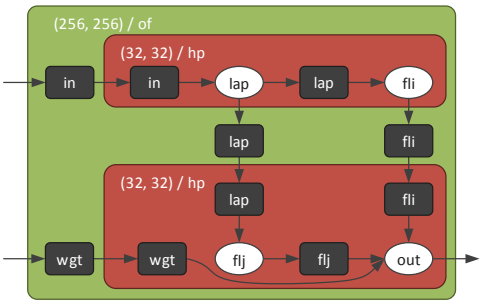


Figure 4: Stencil dependency graph of the horizontal diffusion example annotated with two tiling hierarchy levels.

domain into tiles and apply multiple stencils tile-by-tile. Consequently, we can store temporary values in smaller buffers that hold the working set of a single tile instead of the full evaluation domain.

While tiling increases the data locality, it causes additional synchronization efforts at the tile boundaries. As shown in Figure 1, a single stencil evaluation depends on one or more input or temporary fields accessed in a local neighborhood. When combining multiple stencils the neighborhoods grow depending on the stencil access patterns and the longest path in the dependency graph. We call all dependencies outside of the tile domain the **halo points** of a tile. In addition, we suggest three halo strategies that trade off parallelism against computation. Figure 3 shows the iteration space of one dependency path in the horizontal diffusion example, once without any tiling and then with different tiles as they result from the suggested halo strategies. Shaded regions mark the points that belong to a specific tile.

Computation on-the-fly (of) satisfies all halo point dependencies using redundant computation at the tile boundaries. Hence, we load input fields and evaluate temporary stencils in an extended domain covering the tile itself as well as its halo points. Using computation on-the-fly, we can update different tiles independently postponing synchronization at the cost of additional computation. As shown by Figure 3, computation on-the-fly results in overlapping tiles and is therefore often referred to as overlapped tiling [7, 21, 11].

Halo exchange parallel (hp) satisfies all halo point dependencies using communication with neighboring tiles. More precisely, we update all tiles in parallel and perform at least one halo exchange communication per edge in the longest dependency chain of the stencil dependency graph. Hence, halo exchange parallel avoids redundant computation at the cost of additional synchronizations.

Halo exchange sequential (hs) modifies the tile shape such that all unsatisfied halo point dependencies point in one direction. By iterating over the tiles in reverse dependency direction, we can update all tiles sequentially using a single sweep. While halo exchange sequential in general applies to one-dimensional tilings only, we can complement it with other halo strategies to support higher dimensional tilings. In summary, halo exchange sequential avoids redundant computation and synchronizations at the cost of being sequential.

As the surface to volume ratio decreases with increasing tile size, we preferably update small tiles using halo exchange communication and large tiles using computation on-the-fly. Depending on the hardware architecture high synchronization costs make computation on-the-fly attractive. Overall, choosing the optimal data locality transformations is not straight forward and motivates the use of a performance model.

2.4 Stencil Algebra Definition

Using the data locality transformations introduced in the previous section, we are able to generate a large number of stencil program implementation variants. In particular, we can repeatedly apply our tiling transformations to obtain a hierarchical tiling that leverages multiple levels of the memory hierarchy. By combining our data locality transformations, we are therefore able to cover most of the established stencil implementation techniques. Next, we formally define a stencil algebra whose elements express different stencil program implementation variants and show how to enumerate them. Figure 4 shows an implementation variant of the horizontal diffusion example, introduced in Section 2.2, annotated with two tiling hierarchy levels. Each white node corresponds to a stencil and each black node to a storage region that buffers either an input or a temporary field. We extend the dependency graph with boxes that represent the tiling hierarchy. More precisely, the boxes form a tiling tree where each box corresponds to a tiling that executes all contained boxes respectively stencils. Finally, we annotate each box with the tile size and the halo strategy of the tiling. In Figure 4 we employ an on-the-fly tiling at the bottom of the tiling hierarchy with two nested halo exchange parallel tilings.

In order to specify an element of our stencil algebra, we initially define a tiling hierarchy. More precisely, we define a tile size $t^l \in \mathbb{Z}^n$ for each level l of the tiling hierarchy. In case of the horizontal diffusion example we define two tiling hierarchy levels:

$$t^1_{hd} = (256, 256) \qquad t^2_{hd} = (32, 32)$$

Next, we specify a stencil program implementation variant as a bracket expression. We put all stencils that correspond to a specific tiling hierarchy level into brackets. Therefore, a hierarchical tiling results in a nested bracket expression with the outermost bracket term representing the bottom of the tiling hierarchy. We can define the horizontal diffusion implementation variant shown by Figure 4 using a twofold nested bracket expression.

$$[[s_{lap}, s_{fli}], [s_{flj}, s_{out}]]$$

In the following, we call each bracket term representing a tiling hierarchy a stencil group. A stencil group can be seen as a node of the tiling tree containing nested stencils or stencil groups that as a whole define the stencil program implementation variant.

Let g be a stencil group, then $g.child$ is the set of all children of the stencil group g, where a child is either a stencil or a nested stencil group. In addition, $g.sten$ is the set of all stencils in the subtree defined by the stencil group g. Finally, $g.in$ and $g.out$ define the input and output sets of a stencil group g, where an input and an output correspond to an incoming respectively to an outgoing data

dependency. As an example, we provide the stencil properties of the horizontal diffusion example shown in Figure 4.

$$g_0 = [g_1, g_2] \qquad g_1 = [s_{lap}, s_{fli}] \qquad g_2 = [s_{flj}, s_{out}]$$

First, we define the tree properties.

$$g_0.child = \{g_1, g_2\} \qquad g_0.sten = \{s_{lap}, s_{fli}, s_{flj}, s_{out}\}$$

Next, we define the external data dependencies.

$$g_0.in = \{in, wgt\} \qquad g_0.out = \{out\}$$

We enumerate all stencil program implementation variants using two operations: 1) shuffle the stencils respecting their topological order and 2) group stencils on different tiling hierarchy levels.

2.5 Performance Modeling

In order to understand the performance characteristic of a stencil program implementation variant, we next introduce a performance model. Similar to the Roofline model [19], we estimate the execution time based on the peak compute and communication throughput of the target hardware. In addition, we do not only distinguish between cached and global memory accesses but model additional memory hierarchy levels.

To model our target hardware we use an abstract machine that is built around a processing unit that performs computations on a limited set of local registers. All data is by default stored in a global memory (e.g., DRAM) with limited bandwidth to the processing unit. Data is transferred from global memory to local registers before any computation is performed and the results of a computation are transferred back to global memory before becoming externally visible. Between global memory and local registers there is a set of additional hierarchically organized memory levels, each with limited size, but increasing bandwidth to the processing unit.

When mapping a parallel hardware architecture to our model, the bandwidth of a given memory hierarchy level is the combined bandwidth of all (possibly multiple) memories at this level. The size of a memory hierarchy level is not the combined size, but the size of an individual memory at this level. E.g., assuming there are multiple L1 caches, we consider the size of a single L1 cache. Finally, assuming sufficient parallelism to simultaneously use all processing resources, the compute throughput of our model is the combined peak compute throughput of the hardware architecture.

We now consider again Figure 4, an illustration of a stencil program implementation variant with two tiling hierarchy levels that was introduced in the previous section. Each tiling hierarchy targets one specific level of the memory hierarchy, such as the DDR memory or the L1 cache of a CPU. We assume all input and temporary values of a stencil group are stored in the associated memory hierarchy level. Whenever a stencil program communicates data from one tiling hierarchy level to the next higher one, we model the communication time using the bandwidth of the associated memory hierarchy level. Therefore, we define a communication bandwidth $V^l \in \mathbb{R}$ as well as a memory capacity $M^l \in \mathbb{Z}$ for each level l of the tiling hierarchy. In addition to this vertical communication, a stencil code might also perform lateral halo exchange communication between neighboring tiles of the tiling hierarchy. Hence, we define a lateral communication bandwidth $L^l \in \mathbb{R}$ for each level l of the tiling hierarchy. Typical representatives of lateral communication links are interconnect networks or the scratch pad memory of a GPU. Finally, we define the compute throughput $C \in \mathbb{Z}$ of the target architecture. Thereby, we define storage sizes in terms of floating point values instead of bytes. In case two nested tiling hierarchy levels are associated to the same memory hierarchy level, we set the vertical communication bandwidth to infinity. Like the Roofline model, we assume that we can overlap communication and computation on all communication links respectively compute units of the system.

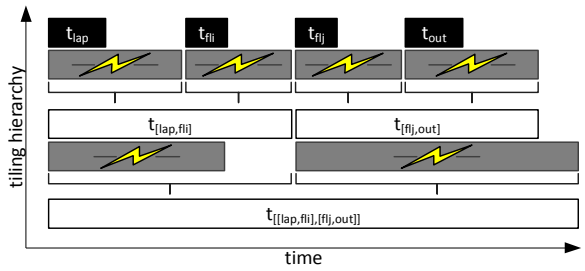


Figure 5: The time estimation for the horizontal diffusion example

When modeling the performance of a stencil program, we assume that the arithmetic intensity remains constant during the execution of a single stencil. On the other hand, the arithmetic intensities of different stencils might vary. Figure 5 illustrates the time estimation for the horizontal diffusion implementation variant shown by Figure 4. At the top of the tiling hierarchy, black boxes denote the stencil execution times. Below, gray boxes (with flashes) denote the communication times between parents and children in the tiling hierarchy. Furthermore, white boxes denote the stencil group execution times computed as the sum of the maximum between stencil execution times and communication times.

In particular, we estimate the execution time t_s of a stencil s that performs c_s floating point operations as the time needed to compute the stencil without considering any communication cost.

$$t_s = c_s / C$$

Using the child execution time t_c of a child stencil or stencil group c that causes v_c vertical and $l_c^1, \ldots, l_c^l$ lateral data movements, we compute the execution time t_g of a stencil group g that corresponds to level l of the tiling hierarchy as the sum of the maximum of the child execution times, the vertical communication between the stencil group and its children, and the lateral communication necessary to update the halo points of the temporary fields. We thereby optimistically assume the lateral communication overlaps with the child execution, which assumes the later communication is sufficiently balanced over the stencil group execution.

$$t_g = \sum_{c \in g.child} \max(t_c, v_c/V^l, l_c^1/L^1, \ldots, l_c^l/L^l)$$

We model the performance of an entire stencil program as the estimated execution time of the stencil group at the bottom of the tiling hierarchy. Furthermore, we complement the performance estimation with a feasibility check that compares the storage requirements of the stencil program to the available memory capacity on all tiling hierarchy levels.

2.6 Stencil Program Analysis

In order to evaluate our performance model, we analyze stencil programs using the mathematical concept of affine sets and affine maps. In particular, we show how to count the number of floating point operations, data movements, and storage locations required during the stencil program execution. Using the performance model introduced in Section 2.5, our analysis finally allows estimating execution time and feasibility of a stencil program.

2.6.1 Affine Sets and Maps

An affine set $S = \{\vec{i} \mid \vec{i} \in \mathbb{Z}^n \wedge \text{cons}(\vec{i})\}$ is a set of n-dimensional integer vectors, where the elements of the set are constrained by a Presburger formula $\text{cons}(\vec{i})$. Presburger formulas consist of comparisons ($<, \leq, =, \neq, \geq, >$) between expressions (quasi-)affine in vector dimensions and external parameters that are combined by Boolean operations ($\wedge, \vee, \neg$). For affine sets set operations such as union, intersection, subtraction, projection as well as cardinality are defined.

An affine map $M = \{\vec{i} \to \vec{j} \mid \vec{i} \in \mathbb{Z}^n, \vec{j} \in \mathbb{Z}^m \wedge \text{cons}(\vec{i}, \vec{j})\}$ is a relation, that relates n-dimensional input (domain) vectors with m-dimensional output (range) vectors. The elements are again constraint by a Presburger formula $\text{cons}(\vec{i}, \vec{j})$. Besides the normal set operations, there exist map-specific operations such as the application of a map m on a set s ($m(s)$), the composition of two maps ($m_0 \circ m_1$), or the inverse of a map (m^{-1}), which switches input and output of a map. We define the following set of important map operations in more detail.

The range product of two maps R_1 and R_2 is defined as:

$$R_1 \times_{\text{ran}} R_2 = \{\, \vec{i} \to (\vec{j}_1, \vec{j}_2) \mid \vec{i} \to \vec{j}_1 \in R_1 \wedge \vec{i} \to \vec{j}_2 \in R_2\}$$

The range intersection of a map R with a set S is:

$$R \cap_{\text{ran}} S = \{\vec{i} \to \vec{j} \mid \vec{i} \to \vec{j} \in R \wedge \vec{j} \in S\}$$

The range-projection of a map R projects the n output dimensions of a map onto the first $k+1$ output dimensions:

$$\mathcal{P}^{\text{ran}}_{[0-k]}(R) = \{\vec{i} \to (j_0, \ldots, j_k) \mid \exists x_{k+1}, \ldots, x_{n-1} \in \mathbb{Z} : \vec{i} \to (j_0, \ldots, j_k, x_{k+1}, \ldots, x_{n-1}) \in R\}$$

R^+ is the transitive closure of R:

$$R^+ = \{\vec{i} \to \vec{j} \mid \exists m \geq 0 : \vec{j} = (\underbrace{R \circ \cdots \circ R}_{m \text{ times}})(\vec{i})\}$$

We use $|S|$ to specify the cardinality of a set and $|R|$ to specify the cardinality of a map, where the cardinality of a map is defined as the number of related domain and range pairs.

We also define named sets and named maps as affine sets and maps that contain so-called "named vectors". The elements of these sets can either be written as tuples of a string and a vector, for example $\{(\text{“}A\text{”}, \vec{i}), (\text{“}B\text{”}, \vec{j}) \mid \vec{i} \in \mathbb{Z}^n, \vec{j} \in \mathbb{Z}^m\}$, or as named vectors $\{A(\vec{i}), B(\vec{j}) \mid \vec{i} \in \mathbb{Z}^n, \vec{j} \in \mathbb{Z}^m\}$. Named sets (maps) allow differently named elements to have vectors of different dimensionality. On named sets and maps the operations introduced above are applied individually to subsets or submaps that share the same name and dimensionality. To extract a set from a named set S, we define a bracket operator $S[\text{“}x\text{”}] = \{(\text{“}x\text{”}, \vec{i}) \mid (\text{“}x\text{”}, \vec{i}) \in S\}$. The bracket operator applied on a map, filters the maps according to the name of their domains $R[\text{“}x\text{”}] = \{(\text{“}x\text{”}, \vec{i}) \to (name, \vec{j}) \mid (\text{“}x\text{”}, \vec{i}) \to (name, \vec{j}) \in R\}$.

Computations on integer sets can be performed with isl [16] and counting of integer sets is possible with barvinok [17].

2.6.2 Data Dependencies

Given a stencil program P the set of flow dependencies in P can be derived from the stencil data dependencies. To obtain them, we define for each stencil $s \in P$ a map D_s that associates the stencil evaluations to the corresponding input data dependencies.

$$D_s = \{s.out(\vec{u}) \to d(\vec{u} + \vec{v}) \mid d(\vec{v}) \in s.in\}$$

Next, we define the union of all stencil data dependencies.

$$D = \bigcup_{s \in P} D_s$$

2.6.3 Stencil Tiling Maps

We model the tiling transformations discussed in Section 2.3 using affine maps that relate the stencil evaluation domain to the tile domain. More precisely, we define for each stencil a tiling map that maps each point in the n-dimensional stencil evaluation domain to an n-dimensional tile identifier, such that all points that belong to the same tile are associated with a common tile identifier. We initially consider only a single tiling level and later generalize the concept to nested tilings.

Given a multi-dimensional tile size vector $\vec{t} = (t_0, \ldots, t_{n-1}) \in \mathbb{Z}^n$, we define a hyperrectangular tiling of a single stencil s as a named map $T^{\square}_s$ that associates each point $\vec{i} = (i_0, \ldots, i_{n-1}) \in \mathbb{Z}^n$ of the stencil evaluation domain with exactly one tile identifier.

$$T^{\square}_s = \{(s, \vec{i}) \to (\lfloor i_0/t_0 \rfloor, \ldots, \lfloor i_{n-1}/t_{n-1} \rfloor)\}$$

Depending on size and alignment of tiles and stencil evaluation domains, such a tiling may yield truncated tiles at the stencil evaluation domain boundaries. In case a given dimension of the stencil evaluation domain should not be tiled (indicated by tile size ∞), the corresponding dimension of the tile identifiers is set to zero.

We represent the tiling of a stencil group g by computing a named map that contains a tiling map for each stencil of the stencil group. We distinguish here between the three halo strategies introduced in Section 2.3.

Computation on-the-fly satisfies halo point dependencies using redundant computation. The corresponding tiling map is therefore a relation which maps the halo point stencil evaluations at the tile boundaries to multiple overlapping tiles. Given a stencil group g, we construct a tile map T_g in two steps. First, all output stencils of g are tiled with a rectangular tiling map. This does not yet introduce any redundant computation. Next, we compute for each tile all stencil evaluations that are required to compute the output points already assign to this tile. We do this by first defining the set of data dependencies D_g that are local to g and then composing the inverse transitive hull of D_g with the tiling map already defined for the output stencils. The resulting map connects the temporary stencil evaluations via the dependent output stencil evaluation to the corresponding tile identifier. This map may now possibly relate one temporary stencil evaluation to multiple tiles and can consequently introduce redundant computation.

$$T_g = \bigcup_{s \in g.out} T^{\square}_s \circ (D^+_g)^{-1}$$

Halo exchange parallel satisfies halo point dependencies using communication. We therefore assign each point in the stencil evaluation domain to exactly one tile and use tiles of identical size, shape and alignment for all stencils in our stencil group. The tiling map T_g describes such a tiling for a stencil group g.

$$T_g = \bigcup_{s \in g.sten} T^{\square}_s$$

Halo exchange sequential is a variant of halo exchange parallel, whose tiling map is constructed accordingly. In contrast to halo exchange parallel, we shift the stencil tiling maps such that all unsatisfied halo point dependencies between tiles point in one direction. Figure 3 illustrates the tile shape of shifted stencil tiling maps and their halo point dependencies. We define a shifted tiling map by subtracting the shift offset form the stencil evaluation domain before computing the associated tile identifiers.

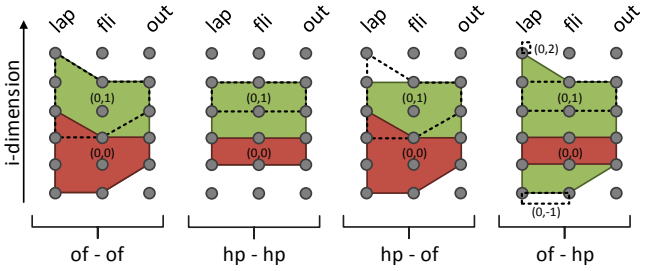


Figure 6: Tile shapes (shaded) for a nested tiling applied to a subset of the horizontal diffusion example projected to the i-dimension

Nested Tilings.

We now describe the construction of nested tilings, tilings that result from recursively applying the previously introduced tiling transformations. To give a first intuition of such tilings, Figure 6 shows the different nested tilings that can be constructed from combining on-the-fly and halo exchange parallel tiling on two tiling levels. It shows for each combination one full outer tile, one full inner tile, and, using dashed lines, the remaining inner tiles placed inside the outer tile. Most combinations are rather straightforward, but it is interesting to note, that in case of on-the-fly tiling being nested inside halo exchange parallel tiling, the redundant computation of the on-the-fly tiles may require the computation of points located outside of the surrounding tile.

As visible in the illustration just discussed, we identify each nested tile with a tile vector whose first and second entry correspond to the tile identifiers of the first and second tiling level respectively. Hence, we can model a nested tiling with l tiling hierarchy levels with a tiling map that relates each point in the n-dimensional stencil evaluation domain to a tile identifier with $n \cdot l$ dimensions. To construct such a map for a given stencil group g nested in another stencil group p we first define tiling maps for the output stencils of g. These tiling maps are formed by combining for each stencil the tiling map $T_p[s]$ that we derive for this stencil from p (not considering any nested groups) with an additional hyperrectangular tiling that uses the tile sizes specified for g. We define the tiling map $T_{g,s}$ of such a stencil s as the range product of the tiling map $T_s^{\square}$ with the recursively computed parent tiling map $T_p[s]$.

$$T_{g,s} = T_p[s] \times_{\text{ran}} T_s^{\square}$$

When computing the tiling map of a nested stencil group T_g, we adapt the previously introduced on-the-fly and halo exchange tiling maps to use $T_{g,s}$ instead of $T_s^{\square}$. The resulting tiling maps for halo exchange parallel and on-the-fly tiling are

$$T_g = \bigcup_{s \in g.sten} T_{g,s} \qquad \text{and} \qquad T_g = \bigcup_{s \in g.out} T_{g,s} \circ (D_g^+)^{-1}.$$

We can now define for each stencil a tiling map T_s that maps each evaluation of this stencil to a tile identifier with $l \cdot n$ dimensions, that identifies for all levels of the tiling hierarchy the tiles the stencil evaluation is assigned to. We obtain T_s by extracting the tile map that corresponds to s from the tile map of the stencil group g at the top of the tiling hierarchy that contains s.

$$T_s = T_g[s]$$

When constructing hierarchical tilings that involve halo exchange sequential, we inherit the shift offsets introduced by the sequential execution to all nested tiling hierarchy levels. Thereby, we align the nested tiles to the parent tile boundaries.

2.6.4 I/O Maps

While the tiling maps alone allow the analysis of computational aspects, we introduce auxiliary maps that support the analysis of data movements and storage usage.

First, we define for each stencil s an input map I_s that relates a set of inputs (stencil evaluations or input fields) used by a certain evaluation of s to the tile(s) this evaluation is assigned to. The construction of I_s is similar to the construction of the on-the-fly tiling. We compose the stencil tiling map T_s with the reversed stencil data dependencies D_s^{-1}. Furthermore, we define the input map of an entire stencil group g as the union of all nested stencil input maps.

$$I_s = T_s \circ D_s^{-1} \qquad I_g = \bigcup_{s \in g.sten} I_s$$

Second, we define for each child stencil or stencil group c an output map O_c that relates the set of outputs written by the child to the tiles they are assigned to. In case the parent stencil group applies halo exchange communication, we define the output map O_c as the union of the child output stencil tiling maps.

$$O_c = \bigcup_{s \in c.out} T_s$$

In case the parent stencil group applies computation on-the-fly, we compute the output map by following the data dependencies starting from the parent stencil group output stencils. While this construction is similar to the computation of the on-the-fly stencil evaluation tiling map, it differs by the fact that we only consider the data dependencies of the stencils executed after the child stencil or stencil group. Thereby, we make sure we do not consider internal dependencies between the output stencils of the child stencil group. Initially, we define the partial input map $I_{p,c}$ of a parent stencil group p and a child stencil or stencil group c considering all input dependencies of children executed after the child c.

$$I_{p,c} = \bigcup_{\substack{c_i \in p.child \\ c < c_i}} I_{c_i}$$

Then the output map O_c of a child stencil or stencil group is the union of all partial input and parent output dependencies.

$$O_c = \bigcup_{s \in c.out} \left(I_{p,c}[s] \cup \left(\bigcup_{o \in p.out} T_{p,o} \right)[s] \right)$$

2.6.5 Tile Selection

We analyze the characteristics of a stencil program by counting stencil evaluations, data movements, or storage requirements on a limited domain. As we are interested in the relative rather than the absolute performance and as our performance model does not consider low hardware utilization due to strong scaling, we can choose an arbitrary but limited domain size. We therefore perform our analysis on the origin tile of the lowest tiling hierarchy level. Assuming m tiling hierarchy levels, we select the origin tile of the lowest tiling hierarchy level using the tile selection set S that contains all tile identifiers with the first n-dimensions fixed to zero.

$$S = \{(x_0, \ldots, x_{n-1}, y_n, \ldots, y_{nm}) \mid x_i = 0 \wedge y_j \in \mathbb{Z}\}$$

When analyzing the storage requirements, we want to make sure a single tile fits the memory capacity of the corresponding memory hierarchy level. We therefore define an additional tile selection set S^* that selects the origin tile on all levels of the tiling hierarchy.

$$S^* = \{(x_0, \ldots, x_{nm}) \mid x_i = 0\}$$

In order to limit the domain of a tiling map, we finally intersect the range of the tiling map with a selection set.

2.6.6 Analysis

Relying on the previously introduced stencil program formulation, we now discuss the analyses we use to obtain the program properties needed for evaluating the performance model introduced in Section 2.5. Using the previously introduced maps, we count the points that correspond to the number of stencils evaluations, the amount of data moved, and the amount of storage used when evaluating a given stencil program on a limited domain.

Computation.

In order to analyze the amount of computation performed by a stencil program, we count the stencil evaluations associated to the origin tile of the lowest tiling hierarchy level. We obtain these evaluations by intersecting the range of the stencil evaluation tiling map with the origin tile selection set S. We then count all stencil evaluations associated to the remaining tile identifiers. Hence, we define the amount of computation c_s performed by a stencil s as the cardinality of the constraint tiling map times the number of floating point operations performed by a single stencil evaluation.

$$c_s = |T_s \cap_{\text{ran}} S| \cdot s.ops$$

Vertical Communication.

As discussed in Section 2.5, vertical communication refers to the data movements between a parent stencil group and its child stencils or stencil groups. We therefore analyze the number of loads and stores performed by a child stencil or stencil group when executed by a parent stencil group. We analyze the vertical communication on a restricted domain that corresponds to the origin tile of the lowest tiling hierarchy level.

In order to compute the number of loads performed by a stencil or stencil group c, we count the elements in the constraint input map of c. More precisely, we intersect the range with the origin tile selection set and project out any dimension above the parent stencil group tiling hierarchy level l. Due to the projection, the points in the resulting map describe all elements loaded by the child stencil or stencil group not considering redundant stencil evaluations on nested tiling hierarchy levels. Hence, we define the number of loads l_c performed by a child stencil or stencil group c as the cardinality of the constraint and projected child input map.

$$l_c = \sum_{s \in c.in} |\mathcal{P}^{\text{ran}}_{[0-nl]}(I_c[s] \cap_{\text{ran}} S)|$$

Accordingly, we define the number of stores s_c performed by a child stencil or stencil group c as the cardinality of the constraint and projected child output map.

$$s_c = \sum_{s \in c.out} |\mathcal{P}^{\text{ran}}_{[0-nl]}(O_c[s] \cap_{\text{ran}} S)|$$

Finally, we define the total amount of vertical communication of a child stencil or stencil group c as the sum of its loads and stores.

$$v_c = l_c + s_c$$

Lateral Communication.

Lateral communication refers to the halo exchange communication between neighboring tiles of the same tiling hierarchy level. We therefore compute the lateral communication performed by a stencil group as the difference between the amount of computed and the amount of consumed temporary values, which corresponds to the unsatisfied halo point dependencies between the children of the stencil group. We analyze the lateral communication on a restricted domain that corresponds to the origin tile of the lowest tiling hierarchy level.

We compute the amount of lateral communication necessary to update the outputs of a child stencil or stencil group, as the difference of the elements used by subsequent children and the elements written by the child itself. Therefore, we intersect the range of the partial input map and output maps with the origin tile selection set and project out any dimensions above the parent stencil group tiling hierarchy level l. Hence, we define the amount of halo points l_c communicated by a child stencil or stencil group c as the cardinality of the difference between the projected and constraint partial input and output maps.

$$l_c = \sum_{s \in c.out} |\mathcal{P}^{\text{ran}}_{[0-nl]}((I_{p,c}[s] \setminus O_c[s]) \cap_{\text{ran}} S)|$$

In case multiple nested tiling hierarchy levels employ halo exchange communication, we possibly run lateral communication on all these levels. By projecting out one level after the other, we assign the lateral communication to the different levels of the tiling hierarchy. Thereby, we get the sum of the lateral communication on the remaining tiling hierarchy levels not yet projected out. By computing the difference of adjacent levels, we finally get the lateral communication assigned to exactly one level.

Storage Requirements.

We analyze the feasibility of a stencil program by computing an upper bound for the storage necessary in order to execute a single tile on each level of the tiling hierarchy. We therefore analyze the storage requirements on a restricted domain that corresponds to the origin tile on all levels of the tiling hierarchy. In case the upper bound exceeds the capacity of one memory hierarchy level, we say a stencil program is infeasible.

We compute the storage requirement of a stencil group as the amount of storage necessary to evaluate the stencil group on a single tile. As shown by Figure 4, we reserve storage for each input and temporary field used during the evaluation of the stencil group. In contrast, output fields are immediately written to storage managed outside of the stencil group. Thereby, we overestimate the storage requirement as the limited life time of some fields might allow sharing a common buffer. We evaluate the storage requirements using the input map intersected with the tile selection set S^*. Furthermore, we project out any dimension above the parent stencil group tiling hierarchy level l. Hence, we define the amount of storage m_p required by a parent stencil group p as the cardinality of the constraint and projected input maps.

$$m_p = \sum_{c \in p.child} \sum_{s \in c.in} |\mathcal{P}^{\text{ran}}_{[0-nl]}(I_p[s] \cap_{\text{ran}} S^*)|$$

In order to determine the feasibility of a stencil program, we compare the memory requirements of each stencil group to the available memory capacity.

3. CASE STUDY

We evaluate our approach using the real-world application COSMO. Its dynamical core was recently rewritten using the STELLA [6] stencil library, which exposes the possibility to manually fuse or split stencils on multiple tiling hierarchy levels. In this case study we show how to automatically tune STELLA programs.

3.1 STELLA

STELLA is a domain specific embedded language for finite difference methods that is designed to separate the stencil specification from the hardware architecture specific implementation strat-

Hierarchy	Vertical	Tile Size	Strategy
1	DDR	$(256, 256, 64)$	of
2	L2	$(8, 8, 64)$	of

Table 1: CPU tiling hierarchy

Hierarchy	Vertical/Lateral	Tile Size	Strategy
1	GDDR/-	$(256, 256, 64)$	of
2	GDDR/-	$(64, 4, 64)$	of
3	Register/Register	$(\infty, \infty, 1)$	hs
4	Register/Shared	$(1, 1, 1)$	hp

Table 2: GPU tiling hierarchy

egy. When executing a stencil program STELLA uses two levels of parallelism: 1) coarse grained parallelization that decomposes the stencil evaluation domain into blocks executed on different processing units and 2) fine grained parallelization that executes the individual blocks on a single processing unit possibly using vectorization and hardware threads. STELLA supports stencil fusion on three different tiling hierarchy levels. We can apply consecutive stencils using a single loop over a block, using multiple separate loops over a block, or using multiple separate loops over the full domain.

At compile-time, STELLA generates target architecture specific loop code using C++ template meta-programming. With two available backends, STELLA can currently target CPU and GPU architectures using the OpenMP and CUDA programming models respectively. Thereby, STELLA employs a fixed but platform specific tiling hierarchy, which we will model using our stencil algebra.

We model the CPU backend of STELLA using the two tiling hierarchy levels shown by Table 1. As discussed in Section 2.6, we compute all stencil program performance characteristics for the origin tile of the base tiling hierarchy level. Therefore, we introduce a first tiling hierarchy level that represents the stencil program evaluation domain. A second tiling hierarchy level models the coarse grained parallelism of STELLA. Currently, the CPU backend does not implement fine grained parallelism. Hence, there is no need to model the third tiling hierarchy level of STELLA.

We model the GPU backend of STELLA using the four tiling hierarchy levels shown by Table 2. Just as in case of the CPU backend, we introduce two tiling hierarchy levels to model the stencil program evaluation domain and the coarse grained parallelism. Furthermore, we introduce two additional tiling hierarchy levels that represent the fine grained parallelism. The GPU backend allocates one thread per ij-position (tiling hierarchy level 4) that iterates over all points in the k-dimension (tiling hierarchy level 3). Thereby, different threads communicate via shared memory, while different loop iterations communicate via registers. Moreover, tile size infinity denotes no tiling in the corresponding dimension.

3.2 Stencil Program Optimization

When implementing a stencil program using STELLA, we have multiple degrees of freedom. As discussed in Section 2.4, we can change the stencil evaluation order and fuse or split the execution of successive stencils on multiple levels of the tiling hierarchy. We therefore split the optimization in two steps and apply different optimization methods: 1) we optimize the stencil evaluation order using brute force search 2) we optimize the tiling for a given stencil evaluation order using dynamic programming. During our optimization we do not consider tile size choices, but rely on the tile sizes that are used by COSMO and have proven robust for a wide range of stencil programs and their implementation variants.

In order to optimize the stencil evaluation order, we enumerate all topological sorts of the stencil dependency graph using brute force search. In general, a graph may have up to $\mathcal{O}(n!)$ valid topological orders. However, due to its data dependency chains a typical stencil dependency graph has less topological orders resulting in a much smaller search space.

In a second step, we search the optimal tiling given a stencil evaluation order. Using a tiling hierarchy and an abstract machine model, we search for a tiling with minimal estimated execution time and a storage requirement that fits all levels of the memory hierarchy. We estimate execution time and storage requirements using the analysis introduced in Section 2.6. In order to enumerate the search space, we fuse all pairs of subsequent stencils on all levels of the tiling hierarchy. Thereby, we assume the subsequent stencils are executed by nested stencil groups that represent the full tiling hierarchy. Given m tiling hierarchy levels and n stencils, up to m tiling hierarchies can be split between each pair of neighboring stencils. Overall, this means there are $\mathcal{O}(m^n)$ ways to split the stencil program. Assuming we have a set of all stencil program implementation variants I and the functions $t(x)$ and $m^l(x)$ that estimate the execution time and the maximal storage requirement at the level l of the tiling hierarchy respectively, we define the following optimization problem:

$$\begin{aligned} &\underset{x \in I}{\text{minimize}} && t(x) \\ &\text{subject to} && m^l(x) \leq M^l \qquad l = 1, \ldots, m \end{aligned}$$

We can either solve the optimization problem using brute force search or employ our dynamic programming approach reducing the search space from $\mathcal{O}(m^n)$ to $\mathcal{O}(mn^4)$ elements. We can apply dynamic programming as the problem has optimal substructure. In particular, we compute for each tiling hierarchy level an n^2 matrix that contains the optimal stencil group executing a continuous subset of the stencil program. Thereby, one matrix dimension corresponds to the start index and the other matrix dimension to the stop index of the subset. We compute a matrix entry using a second dynamic programming algorithm[2] that constructs the optimal stencil group using a combination of the previously computed optimal child stencil groups. More precisely, we compute the optimum for a given start and stop index either using the optimal child stencil group containing all stencils or using a child stencil group containing all stencils from an intermediate index to the stop index plus the recursively computed optimum from the start index to the intermediate index. By increasing the intermediate index step-by-step and storing partial solutions, we compute a single entry of our n^2 matrix using $\mathcal{O}(n^2)$ steps.

4. EVALUATION

We evaluated our framework using three example kernels from the COSMO atmospheric model. In addition to the horizontal diffusion kernel "hd" introduced in Section 2.2, we use two kernels that are part of the most time-consuming component in COSMO, the sound wave forward integration. More precisely, the "uv" kernel updates the horizontal wind velocity components by computing the horizontal pressure gradient, whereas the "div" kernel computes the divergence of the three-dimensional wind field. Figure 7 illustrates all kernels used during the evaluation including a combination of the "uv" and "div" kernels.

We perform our experiments using adapted standalone kernels: 1) we replace divisions by multiplications to increase the numerical stability on random input data and 2) we replace one-dimensional

[2] Our nested dynamic programming step is not guaranteed to find the optimal solution. For all four example kernels discussed in Section 4, exhaustive search based tests confirmed the optimality of the dynamic programming results for several stencil evaluation orders.

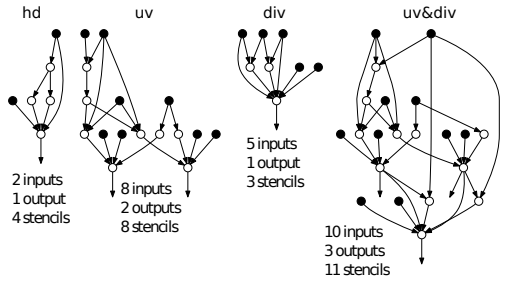


Figure 7: Example kernel stencil dependency graphs

Hierarchy	Vertical (V)	Memory (M)
1	26 GB/s	∞
2	768 GB/s	512 KB

Table 3: Intel Core i5-3330

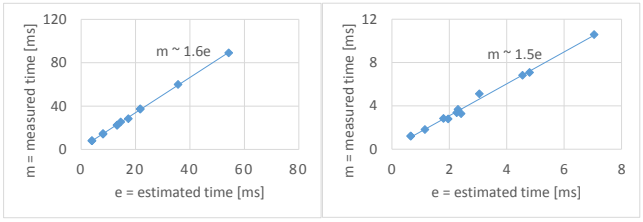
constant fields by scalar constants as our framework does only support n-dimensional fields. We implement for each kernel three different variants: 1) "no fusion" refers to a naive implementation without loop fusion, 2) "hand-tuned" refers to a manually tuned implementation as used in production by COSMO, and 3) "optimized" refers to an automatically tuned version using MODESTO. All kernel variants are written using STELLA and therefore are parallel and employ tiling. Similar to the production configuration, we run our experiments using a $(256, 256, 64)$ point domain that provides sufficient parallelism to fully utilize the hardware.

We measure the performance of our example kernels using two target architectures: 1) an Intel Core i5-3330 CPU with a dual channel DDR3-1600 memory interface and 2) a Nvidia Tesla K20c GPU. Table 3 and Table 4 define the machine model of the target architectures for the STELLA tiling hierarchy discussed in Section 3.1. Thereby, we use the peak bandwidth of the individual memory hierarchy levels, except for the register file and shared memory where we assume infinite bandwidth and one half of the peak bandwidth respectively. More precisely, we divide the peak bandwidth of memories used for lateral communication by two as each communication corresponds to a write and a read access. Furthermore, we underestimate the capacity of the GPU register file as it is used for additional tasks such as index computations. Finally, we set the peak performance C of the target architectures to 48 Gflops and 585 Gflops respectively (without fused multiply add).

To evaluate the accuracy of our performance model, we compare the measured execution time of our example kernels to the modeled execution time. Figure 8 shows the accuracy of the model for both target architectures. Using linear regression, we fit trend lines that show a close correlation of modeled and measured performance. Hence, the relative performance of modeled and measured execution times for different kernels are in accordance, which is of key importance for our approach. However, we consistently overestimate the absolute performance as the kernels can not leverage the peak performance of both target architectures. Our performance

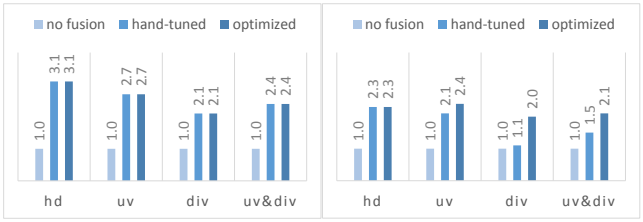
Hierarchy	Vertical (V)	Lateral (L)	Memory (M)
1	208 GB/s	-	∞
2	208 GB/s	-	∞
3	∞	1174 GB/s	4096 Registers
4	∞	∞	32 Registers

Table 4: Nvidia Tesla K20c



(a) Accuracy CPU (b) Accuracy GPU

Figure 8: Comparison of measured and estimated execution time



(a) Speedup CPU (b) Speedup GPU

Figure 9: Speedup of hand-tuned and optimized kernels

model shows that our kernels, like most stencil computations, are heavily memory bandwidth limited. Consequently, the correlation factors of 1.5x respectively 1.6x can be attributed to the fact that the kernels attain only a fraction of the peak main memory bandwidth.

Figure 9 shows the speedup of hand-tuned and automatically tuned implementation variants for both target architectures. As discussed in Section 3.2, MODESTO optimizes topological order and stencil fusion. Overall, MODESTO achieves the same or better performance compared to the hand-tuned kernels used by COSMO. Starting from a naive STELLA implementation, we are able to improve the performance by a factor 2.0x–3.1x. The first three experiments achieve optimal performance by fusing all stencils on the highest level of the tiling hierarchy. In contrast, for the last experiment fusing all stencils exceeds the memory capacity. Hence, the optimization splits the stencils in two separate groups. To verify this decision, we implemented an additional variant of the last experiment that fuses all stencils. On CPU and GPU fusing all stencils results in a 10% and 8% performance reduction respectively.

5. RELATED WORK

Optimal and close-to-optimal stencil arrangements have been investigated for several decades. Many approaches rely on empirical methods to derive efficient implementations. Datta et al. [3] optimize an example stencil for a wide range of hardware architectures using autotuning. Patus [2] is a DSL autotuning framework for single stencil computations on multi-core CPUs and single GPUs. Zhang et al. [20] present an iterative compilation approach for single stencil computations on single and multi GPU systems which focuses on deriving optimal block sizes.

Overtile [7] is a DSL code generator for iterative stencils that uses overlap tiling to generate efficient GPU code also relying on iterative compilation. There is also a cache-oblivious tiling strategy for iterative stencil computations [5] for which the number of expected cache misses has been analytically computed and empirically evaluated for single CPU systems and one caching level.

For stencil graphs, there is Halide [11], a DSL based approach focused on image processing. Halide uses again compilation based autotuning to choose stencil program implementation variants considering a set of tiling strategies and further optimizations. PolyMage [9] is an image processing DSL that guides the optimization using a model-driven heuristic. Basu et. al [1] perform loop fu-

sion, overlapped tiling and wave front execution for optimizing a geometric multigrid stencil graph. They do not consider hierarchical tiling and do not use any analytical model. Olschanowsky et al. [10] optimize an iterative, but multi-kernel stencil computation resulting from solving partial differential equations and study different inter-loop optimizations using empirically evaluation on multi-core CPUs.

There has also been work that discusses analytical performance models. There is work not limited to stencil computations that provides lower bounds for tile sizes selection [13]. Renganarayana et al. [12] use geometric optimization to model tiling and related problems on one and multiple levels and to derive optimal tile sizes. Zhou et al. [21] present work on hierarchical overlapped tiling and optimize OpenCL programs for multi-core CPUs. They provide basic performance models for the number of stencils to fuse into one tile focusing on (possibly unrolled) kernels that process only one stencil repeatedly and do not consider varying tiling and fusion strategies. Finally, Wahib et al. [18] take arbitrary stencil graphs from larger scientific applications and present an analytical performance model for choosing an optimal execution strategy. Even though closely related, they limit themselves to kernel fusion using computation on-the-fly only considering shared memory and apply their work on NVIDIA GPUs only.

6. CONCLUSION

With MODESTO we have presented an approach for modeling and automatically selecting efficient implementation strategies for stencil programs. Focusing not only on single, possibly iterative applications of stencils, but on directed acyclic graphs of stencils we consider the effects of three different tiling strategies in combination with different fusion choices, all applied on possibly multiple hierarchy levels. We model the effects of these implementation strategies on the use of both lateral and vertical memory bandwidth, and estimate the cost of possibly redundant computation by using a analytical model that allows to predict the amount of data transfer and computation for a given stencil program implementation variant. In combination with a given CPU or GPU model we estimate the relative performance of the different implementation variants and show using a combination of exhaustive search and dynamic programming how to choose the best implementation variant.

We evaluated MODESTO by means of the STELLA stencil library that implements different stencil program transformations for CPU and GPU architectures. In particular, we successfully model the tiling hierarchy of STELLA and automatically tune kernels of the COSMO atmospheric model. Thereby, we achieve speedups of 2.0–3.1x against naive and speedups of 1.0–1.8x against expert-tuned implementation variants.

Acknowledgments

This publication has been funded by Swissuniversities through the Platform for Advanced Computing Initiative (PASC). We thank Oliver Fuhrer (MeteoSwiss) and Carlos Osuna Escamilla (ETH) for helpful discussions, Armin Größlinger (University of Passau) for providing isl bindings for Java, as well as the Swiss National Supercomputing Center (CSCS) for their continuous support.

7. REFERENCES

[1] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *High Performance Computing (HiPC), Int. Conf. on*, pages 452–461. IEEE, 2013.

[2] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE Int.*, pages 676–687, May 2011.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proc. of the 2008 ACM/IEEE Conf. on Supercomputing*, SC '08, pages 4:1–4:12. IEEE Press, 2008.

[4] G. Doms and U. Schättler. The nonhydrostatic limited-area model LM (lokal-model) of the DWD. Part I: Scientific documentation. Technical report, German Weather Service (DWD), Germany, 1999.

[5] M. Frigo and V. Strumpen. The memory behavior of cache oblivious stencil computations. *The J. of Supercomputing*, 39(2):93–112, 2007.

[6] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, and T. Schulthess. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomputing frontiers and innovations*, 2014.

[7] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Int. Conf. on Supercomputing, Proc. of*, pages 311–320. ACM, 2012.

[8] G. A. McMECHAN. Migration by extrapolation of time-dependent boundary values*. *Geophysical Prospecting*, 31(3):413–420, 1983.

[9] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proc. of the Twentieth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[10] C. Olschanowsky, M. M. Strout, S. Guzik, J. Loffeld, and J. Hittinger. A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 793–804, NJ, USA, 2014. IEEE Press.

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. of the ACM Conf. on Programming Language Design and Implementation*, PLDI '13, pages 519–530. ACM, 2013.

[12] L. Renganarayana and S. Rajopadhye. Positivity, posynomials and tile size selection. In *ACM/IEEE Conf. on Supercomputing, Proc. of*, SC '08, pages 55:1–55:12, NJ, USA, 2008. IEEE Press.

[13] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. Analytical bounds for optimal tile size selection. In *Compiler Construction*, pages 101–121. Springer, 2012.

[14] A. Taflove. Review of the formulation and applications of the finite-difference time-domain method for numerical modeling of electromagnetic wave interactions with arbitrary structures. *Wave Motion*, 10(6):547 – 582, 1988. Special Issue on Numerical Methods for Electromagnetic Wave Interactions.

[15] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *Proc. of the Twenty-third Annual ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[16] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS'10*, pages 299–302. Springer, 2010.

[17] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.

[18] M. Wahib and N. Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 191–202, Piscataway, NJ, USA, 2014. IEEE Press.

[19] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[20] Y. Zhang and F. Mueller. Autogeneration and autotuning of 3d stencil codes on homogeneous and heterogeneous gpu clusters. *Parallel and Distributed Systems, IEEE Transactions on*, 24(3):417–427, 2013.

[21] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proc. of the Inter. Symp. on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.