Modeling Optimization of Stencil Computations Via Domain-level Properties

Brandon Nesterenko
Qing Yi
Brandon Runnels
University of Colorado Colorado Springs
Colorado Springs, CO, USA

Pei-Hung Lin
Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, CA, USA

Abstract

Stencil computations are widely used in the scientific simulation domain, and their performance is critical to the overall efficiency of many large-scale numerical applications. Many optimization techniques, most of them varying strategies of tiling and parallelization, exist to systematically enhance the efficiency of stencil computations. However, the effectiveness of these optimizations vary significantly depending on the wide range of properties demonstrated by the different stencils. This paper studies several well-known optimization strategies for stencils and presents a new approach to effectively guide the composition of these optimizations, by modeling their interactions with four domain-level properties of stencils: spatial dimensionality, temporal order, order of accuracy, and directional dependence. When using our prediction model to guide optimizations for five real-world stencil applications, we were able to identify optimization strategies that outperformed two highly optimized stencil libraries by an average of 2.4x.

1 Introduction

In the scientific simulation domain, stencils are widely used in a variety of applications[20]. Often, the efficiency of the overarching application is critically linked to the performance of its stencils, where a large effort of work exists to optimize their performance. More specifically, many approaches exist to optimize their performance, including both automatic optimization frameworks by using compilers[9, 29] and domain-specific languages (DSLs)[26, 30]; and manual optimization by using libraries such as AMReX[13] or modifying the source code implementations to directly manage hardware resources. Where a majority of these optimizations are mostly variations of tiling[8, 28] and parallelization[18] and can be systematically applied to a wide variety of stencils, their effectiveness can be inconsistent due to the vast range of behaviors that their underlying computations can exhibit[21].

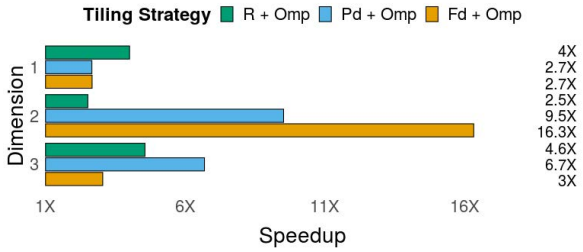


Figure 1. Impact of dimensionality on the effectiveness of rectangular (R), partial diamond (Pd), and full diamond (Fd) tiling with thread-level parallelism with 16 threads.

To illustrate the wide variation of optimization effectiveness, figure 1 compares the impact of applying three different optimization strategies to three implementations of a *heat equation* ($\frac{\partial u}{\partial t} = \alpha \Delta u$), which simulates heat propagation for an isotropic and homogeneous medium [14]. Here the three different implementations respectively instantiate the equation to be one-dimensional, two-dimensional, and three-dimensional. Each variation combines thread-level parallelism (using sixteen threads) with one of three different memory tiling strategies: (1) *rectangular tiling*, which strip-mines data accessed by a loop nest into rectangular blocks, with outer loops enumerating distinct blocks of data, and each block updated by an equal number of inner loops [28]; (2) *full diamond loop tiling*, which alternatively uses diamonds as the block shapes to allow for more concurrency and better load balancing among tasks that update distinct blocks of data[8]; and (3) *partial diamond loop tiling*, which works similarly as full-diamond loop tiling but places additional restrictions on the number of dimensions that can be tiled[8]. Rectangular tiling provides the highest speedup (4x) for the one-dimensional implementation, as it introduces the least amount of runtime overhead compared to the diamond tiling strategies. As the dimensions of the stencil increase, full-diamond tiling provides the highest speedup (16x) for the two-dimensional implementation, where more data are

PMAM'22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9339-3/22/04…$15.00
https://doi.org/10.1145/3528425.3529103

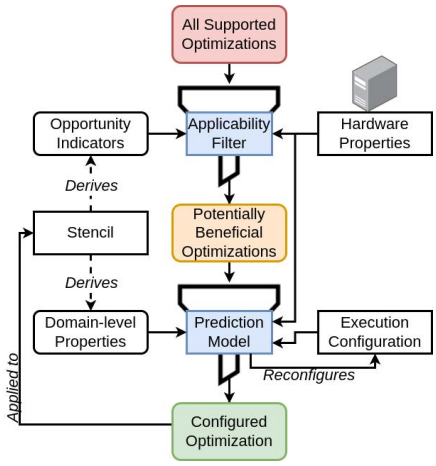


Figure 2. Workflow showing the process to iteratively navigate the optimization space for a stencil

shared across thread boundaries. Partial-diamond tiling provides the best speedup (6.7x) for the three-dimensional implementation, by lowering the runtime overhead of tiling compared to full-diamond tiling.

This paper aims to study the relationship between domain-level properties of stencils and optimization effectiveness, in order to model the interactions among the various optimizations and suggest the most effective optimization strategy for an arbitrary stencil. We have identified four higher-order properties of stencils: spatial dimensionality, temporal order, order of accuracy, and directional dependence, which have shown the most prominent influences to the data-access patterns of stencils. This paper focuses on correlating these properties with five well-known optimizations for stencils: OpenMP parallelization[11], rectangular loop tiling[28], partial and full diamond loop tiling[8], and multiplicative inversion[17]. There are three main challenges that we address:

1. Addressing safety complications from applying an optimization.
2. Selecting among multiple alternative code transformations that serve similar purposes.
3. Effectively configuring the transformations and execution configurations for a given architecture.

Figure 2 presents our workflow. The *applicability filter* provides a set of optimizations that are both 1) safe to apply to the source code, i.e. those that won't interfere with previously applied optimizations, and 2) supported by the underlying hardware platform. This component takes two sets of input into consideration: *opportunity indicators* and *hardware properties*, which respectively expose potential performance improvement opportunities in the input source code (e.g. whether the code can be parallelized) and the hardware platform (e.g., whether multi-threading is supported by the hardware). As some of the selected optimizations may serve similar purposes, the *prediction model* predicts the relative effectiveness of each, and outputs the one which should be most effective, and additionally predicts a set of tuning parameter values to configure it. This decision is made by considering a set of pre-selected *domain-level properties* of the stencil computation, its current *execution configuration*, and the properties of the hardware platform on which it will run. After applying the optimization recommended by our prediction model, either manually by developers or automatically through some compiler-based tools, the optimized code can be again optimized via our workflow, until all the applicable optimizations have been applied.

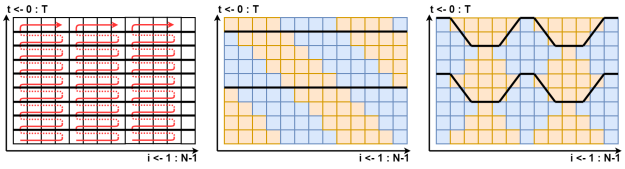
For the prediction model to predict the most beneficial optimization and configuration, it performs multi-variate polynomial regression analysis on a data set of performance metrics from profiling runs of 456,516 automatically generated and optimized stencil variants. To create these variants, we developed a special-purpose stencil code generator and profiler that parameterizes domain-level properties, optimization strategies, optimization configurations, and execution configurations. The regression model itself functions as a black box which uses a stencil's domain-level properties and execution configuration as its inputs, and predicts the speedup of each supported optimization to determine which optimization will be most profitable. We test the accuracy of this model by splitting the generated data set into training and testing sets, where 66% of the optimization strategies suggested by our model resulted in the optimal strategy, and 89% of the suggestions had performance values which fell within 10% of the performance of the optimal strategy. Our optimizer is then used to guide the optimization of five stencil kernels which are prominent in the scientific computing domain. Specifically, we use the Allen-Cahn[1], heat[5], Laplacian[7], Poisson for electrostatics[12], and wave[6] stencil kernels. We then compare the stencil implementations guided by using our optimization process against those optimized via AMReX[13] and Pochoir[26]. We show that in most cases, our optimization process outperforms these libraries (average of 2.4x). The case where our process did not result in highest performance is a result of our current prototype, which does not yet consider the breadth of all available stencil optimization types. Our future work entails extending our model with additional types of optimizations, e.g. dynamic programming, to cover these cases.

Our main technical contributions are as follows:

- We introduce a code optimization process that is unique in its internal representation and iteration of the optimization space using four different aspects of stencils: domain-, hardware-, code-, and execution-level. Our prototype focuses on properties that are tightly tied to multi-core architectures and memory performance.
- We show our approach is highly effective on five distinct stencil computations in comparison to two state-of-the-art grid computing libraries.

```
for(int t = 0; t < T; t++)
    for(int i = 1; i < N - 1; i++)
        A[(t+1)%2][i] = 0.125 * (
            - 4.0*A[t%2][i]
            + A[t%2][i+1] + A[t%2][i-1]);
```

Listing 1. One-dimensional, second-order accurate heat equation



(a) OpenMP Parallelization **(b)** Rectangular tiling **(c)** Diamond tiling

Figure 3. Impact of different optimization techniques on stencils (the x axis is the spatial dimension (from i=1 to N-1), and the y axis is the temporal loop iteration (from t=0 to T))

The remainder of this paper is organized as follows. Section 2 provides a background on the supporting optimizations of stencils. Section 3 provides a background on stencil codes and their domain-level properties. Section 4 describes our approach. Section 5 provides our experimental results. Section 6 discusses related works, and section 7 concludes.

2 Background: Stencil Optimization

For the purpose of this paper, we define stencils as computations that operate on multi-dimensional array representations of structured grids. Listing 1 implements the one-dimensional heat equation; the i loop updates each point in a structured grid as a weighted contribution from its neighboring points[22], and the t loop repeats the computation until convergence. Specifically, at each time-step t, the simulation computes a new value for each point (i) in the grid for the next time step $t+1$ by using values of the neighboring points ($i+1$ and $i-1$) from the current time-step t. Within the scope of this paper, we study five well-known optimizations for stencil computations.

OpenMP Parallelization[11]. Illustrated in figure 3(a), this optimization partitions the spatial dimensions of a stencil to be evaluated on multiple threads, each thread receiving an equal portion of the stencil to compute. The optimization is profitable when the data size of a stencil is sufficiently large.

Rectangular tiling[28]. illustrated in figure 3(b), this optimization strip-mines all the spatial dimensions of a stencil into rectangular blocks, with a set of outer loops enumerating all the blocks that comprise the overall stencil, and the inner loops traversing each block of data to perform needed computation[28]. When combined with OpenMP parallelization, each thread is assigned multiple blocks of data to compute, and as the time step loop progresses, the blocks assigned to each thread (colored yellow and blue in Figure 3(b)) shift to the left to accommodate dependence relations among neighboring blocks, so that tasks assigned to different threads can start in a pipelined fashion.

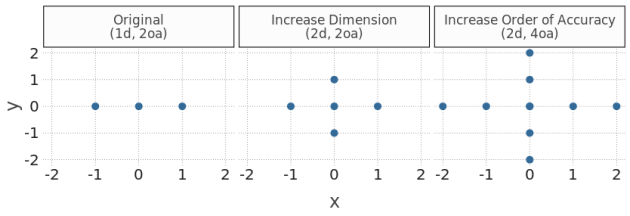


Figure 4. The impact spatial dimensionality and order of accuracy have on stencil data-access pattern

Diamond tiling[8]. as an extension to rectangular tiling, this optimization has two variants: full and partial. In essence, it divides all the rectangular blocks into two groups, colored yellow and blue in figure 3(c), so that when parallelized, all blocks in the same group are entirely independent of each other and can proceed concurrently, while blocks that belong to distinct groups wait for one another. To account for dependence across time steps, the block size changes at each time step to avoid cross-thread communications, resulting in tile shapes that resemble diamonds. *Full diamond tiling* exploits the nearest neighbor accesses in all dimensions; however, this can lead to overhead due to overly complex tile shapes. *Partial diamond tiling* only considers a subset of a stencil's dimensionality to reduce this overhead.

Multiplicative inversion[17]. this optimization substitutes division operations with constant denominators by using multiplication of the multiplicative inverse, i.e., $\frac{a}{b}$ becomes $a * b^{-1}$. By pre-computing and saving the multiplicative inverse value outside of loops, the overall computation overhead is reduced.

3 Stencil Properties

A key incentive of this paper is to extract and model the impact of important properties of stencil computations and hardware platforms to identify an optimization workflow that can effectively enhance their performance. The following subsections introduce these factors.

3.1 Domain-Level Properties of Stencils

The main factors that impact a stencil's performance are its computation and data access patterns, which can be categorized using domain-level properties of the stencil computation. For our purpose of modeling optimization effectiveness, we have identified four domain-level properties, detailed below, which we show are particularly relevant in the effectiveness of tiling and parallelization optimizations.

Spatial Dimensionality. This property defines the number of independent variables, discounting the time variable,

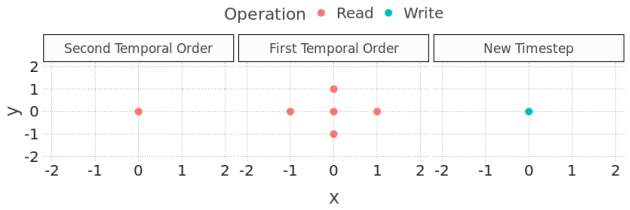


Figure 5. The impact temporal order has on stencil data-access pattern

used in the overarching partial differential equation. For example, although the 3D heat equation $\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right)$ has four terms with independent variables, t is a time variable, therefore the spatial dimensionality is 3. The spatial dimensionality property defines both 1) the number of spatial dimensions of the underlying storage array in the stencil implementation, and 2) the shape of the stencil lattice update. To illustrate, figure 4 shows the data access pattern of a one-dimensional stencil, which accesses three data points in a line at each iteration of its nested loop. Increasing the spatial dimensionality to two not only adds an additional dimension to the storage array but also changes the stencil update shape to a cross, by accessing the neighboring points of each stencil update in both dimensions.

Order of Accuracy. This property defines how many neighboring elements are used in each dimension to update each element of the stencil grid, to ensure a desired rate of error reduction by each time step. For example, the first two stencils in figure 4 are second order accurate, as they use two neighbors in each dimension to compute each lattice update. The third stencil uses four neighbors in each dimension to perform the update, therefore increasing the order of accuracy to four.

Temporal Order. Conceptually representing the order of the derivative taken with respect to the time independent variable in the PDE, this property is represented in the stencil implementation by how many results computed by the previous time steps are accessed in each lattice update. Specifically, higher temporal order problems require more results of the previous time steps to be saved for later uses. For example, the heat equation ($\frac{\partial u}{\partial t} = \alpha\Delta u$) is first temporal order, because it is derived from the term $\frac{\partial u}{\partial t}$. In its implementation (listing 1), the lattice update uses the previous time step, $(t\%2)$, to calculate the new value, at $((t+1)\%2)$.

Directional Dependence. Conceptually this property represents whether each spatial dimension of a stencil diffuses in an *isotropic* or *anisotropic* fashion: isotropic stencils diffuse equally in all directions/dimensions, whereas anisotropic diffusions are direction dependent. At the implementation level, this property is represented by how each data element in a lattice update (LUP) is weighted. Because isotropic diffusion assumes uniform grid spacing [14], its stencil implementation uses constant values as coefficients to uniformly scale

```
for(int t = 0; t < T; t++)
    for(int i = 1; i < N - 1; i++)
        A[(t+1)%2][i] =
            c[0][i] * u_0_0[t%2][i]
          + c[1][i] * u_0_0[t%2][i-1]
          + c[2][i] * u_0_0[t%2][i+1];
```

Listing 2. Fully anisotropic 1D heat stencil

the neighboring points at each LUP. For example, the heat equation in listing 1 weighs all neighboring points with a constant value 1, and prior values of the updated point with -4. In contrast, an anisotropic stencil scales each point differently by storing the desired coefficients inside additional grids. For example, $\frac{\partial u}{\partial t} = \nabla \cdot (\alpha(x)\nabla u)$, is an anisotropic variant of the heat equation and uses the function $\alpha(x)$ to weigh the points. Listing 2 shows the corresponding code for this equation, where the coefficients for each term are stored in an extra array c. Additionally, for stencils with spatial dimensionality higher than one, only distinct dimensions can be anistropic, e.g. $\nabla \cdot (\alpha(x)\nabla u(x,y))$. This would result in corresponding coefficient grids with smaller dimensionality than the stencil's spatial grid; however, in the scope of this paper, we assume anisotropy in all dimensions, as it is a super-set and can still be used for semi-anisotropic stencils.

Three additional properties, mixed derivatives, where derivatives are taken with respect to multiple independent variables, e.g. $\frac{\partial^2 u}{\partial x \partial y}$; skewing, which changes the pattern in which nearest neighbors are accessed, e.g. from using directly adjacent points to corner points; and iterative method, which determines how the solution will converge, are additional important properties which we leave for our future studies. This study focuses on the widely popular Jacobi method[10], non-skewed data accesses, and pure derivatives.

3.2 Performance Of Stencils

While a wide variety of hardware platforms, including CPUs, GPGPUs, and other accelerators[22–24], can be used to run stencils efficiently, for the scope of this paper, we focus on CPU-based multicore architectures, where the performance of a stencil computation is largely determined by their usage of the underlying memory hierarchy and processors. We intend to address a wider variety of hardware architectures in our future work.

Without explicit optimization, the computation and data-access patterns of stencil computations typically run inefficiently on existing multi-core architectures, as the computation traverses a large amount of data within each time step. Additionally, the degree of concurrency among the different threads often determines the overall performance. Another factor that impacts stencil performance is division, which often compute coefficients and step sizes. Division can take 3-6 times as long as other operations [3], making it difficult to hide its long latency.

The performance of a stencil can also be influenced by its execution configuration, specifically any pre-set constraints

on the environment in which the stencil must run. Examples of such factors include the size and layout of the input data a stencil computation needs to operate on, the number of time steps that the computation needs to run before convergence can be achieved, and the number of threads available to be dedicated to the stencil computation.

4 Modeling Optimization Effectiveness

In this paper, we focus on modeling the effectiveness of five optimizations: rectangular loop tiling, partial diamond tiling, full diamond tiling, OpenMP parallelization, and multiplicative inversion. The goal is to predict the optimization that is expected to produce the highest speedup based on the data access patterns (represented by domain-level properties of a stencil) and execution configurations of a given stencil computation. The key strategy is to identify the factors which contribute to a stencil's inefficient use of hardware resources, and predict the optimization which should best address these factors and improve performance.

Figure 2 presents the overall workflow of our prediction process, which includes two main components. First, the *applicability filter* identifies a set of optimizations which target the underlying stencil and hardware platform. More specifically, this component cross-references coding patterns (exposed via opportunity indicators) with properties of the underlying hardware to identify potential inefficiencies of hardware resources. The set of potentially beneficial optimizations is the combination of all optimizations which target these inefficiencies. The *prediction model* then predicts and configures the optimization from this set that should provide the highest speedup for the stencil. More specifically, this prediction uses regression analysis that correlates properties which contribute to a stencils data access pattern to an optimizations effectiveness. These properties consist of domain-level properties, which are extracted automatically from the stencil's source code using a preliminary code analyzer we developed, and execution configuration, which is provided by a user. The optimization is then tuned to the underlying hardware resources using the execution configuration and hardware properties.

The workflow is repeatable to allow for multiple rounds of optimization to be applied. In particular, after applying an optimization, the stencil's source code properties will change. Upon performing a new round of optimization, the applicability filter will not detect as many potential inefficiencies, and the set of potentially beneficial optimizations suggested to the prediction model will be reduced. The repetition ends when either the applicability filter can no longer identify any relevant optimizations, or the prediction model no longer predicts any optimization to be beneficial.

4.1 The Applicability Filter

The *applicability filter* in our workflow provides the needed check to make sure only potentially profitable optimizations are applied to a stencil. It is assumed that the correctness of each optimization will be guaranteed either manually by the developer or automatically by some software tools.

Algorithm 1 Applicability Filter Algorithm

```
function IdentifyPromisingOptimizations(OI, HP)
    promising_opts ← ∅
    for each opt ∈ SUPPORTED_OPTIMIZATIONS do
        is_opt_favorable ← TRUE
        for each oi ∈ OI do
            if ! is_beneficial_for(opt, oi) then
                is_opt_favorable ← FALSE
                break
            end if
        end for
        for each hp ∈ HP do
            if ! targets(opt, hp) then
                is_opt_favorable ← FALSE
                break
            end if
        end for
        if is_opt_favorable then
            append(promising_opts, opt)
        end if
    end for
    return promising_opts
end function
```

Algorithm 1 details how the applicability filter determines the set of potentially beneficial optimizations. We define a function, $IdentifyPromisingOptimizations$, which takes two inputs: OI, for opportunity indicators, and HP, for hardware properties; and outputs a set of optimizations, O, which are potentially beneficial to the stencil. The output set of optimizations, $promising\_opts$, is initialized as an empty set ($\varnothing$). An optimization is considered promising if it is applicable to both 1) the underlying source code, which is exposed via opportunity indicators, and 2) the hardware, which is exposed via hardware properties.

Table 1 defines the opportunity indicators and hardware properties we use for the scope of this study, and their correlations with optimizations. Specifically, we define three opportunity indicators and two hardware properties, which are referenced by the functions $is\_beneficial\_for$ and $targets$ in algorithm 1, respectively. The first opportunity indicator, *iteration-space traversal of linear-order*, classifies whether all loops in a stencil traverse the stencil in a single pass. This hints that the code may not be efficiently using the cache hierarchy when the stencil is large, and is an indicator for all the tiling optimizations, i.e. rectangular and partial/full diamond tiling. The second indicator, *serial execution*, is set to *true* if the source code is sequential (not already parallelized). It is therefore an indicator for parallelization, i.e., OpenMP, which modifies the computation to use multiple cores of the hardware platform. The third indicator, *number of constant denominators per lattice update (LUP) > 0*, indicates whether the stencil computation contains divisions with denominator values that don't change across LUPs, and identifies multiplicative inversion.

Optimization	Opportunity Indicators	Hardware Target
Rectangular Tiling	iteration-space traversal of linear-order	Multi-level Cache
Partial Diamond Tiling	iteration-space traversal of linear-order	Multi-level Cache, Multicore
Full Diamond Tiling	iteration-space traversal of linear-order	Multi-level Cache, Multicore
OpenMP Parallelization	serial execution	Multicore
Multiplicative Inversion	num constant denominators per LUP > 0	Any

Table 1. Conditions for an optimization to be valid

The applicability filter component uses two hardware-level properties to select applicable optimizations. The first property, *multi-level cache*, indicates whether the hardware has multiple levels of cache, which is an indicator for optimizations that target multi-level caches, e.g., variations of tiling. The second property, *multicore*, indicates how many cores are available in the underlying platform and thereby indicates whether any parallization optimization is applicable. Additional opportunity indicators likely need to be included to accommodate new types of optimizations or to target alternative types of architectures (e.g., GPUs). However, our experimental results show that the five indicators in table 1 are sufficient for selecting the five optimizations and the multi-core architecture we target.

4.2 The Prediction Model

After identifying potentially profitable optimizations for a stencil, the prediction model predicts the one which should provide the highest speedup. This is a two-step process: 1) predicting the optimization which should provide the highest speedup, and 2) predicting a configuration which should maximize the speedup from applying the optimization. We build a series of multi-variate linear regression models with interaction[4] to predict both the maximum speedup that an optimization can attain (step 1), and the speedup that an optimization will provide under a given configuration of tuning parameters (step 2). Fundamentally, each model predicts the speedup of a particular optimization using a given set of inputs. The regression models for step 1 predict the maximum speedup of each supported optimization using two inputs: the input stencil's domain-level properties and execution configuration. The regression models for step 2 extend the prediction from step 1 to predict the real speedup of the optimization by adding another input: a particular configuration of the optimization's tuning parameters. The set of tuning parameters can change depending on the spatial dimensionality of the stencil, as higher dimensionalities will use more loop blocking factors. The prediction for step two then chooses from multiple potential models for each optimization based on the spatial dimensionality of the stencil.

To build the regression models, we automatically generate a data-set of 456,516 stencil variants that differ from one another by domain-level properties of the stencil, execution configuration, optimizations applied, and tuning parameters of each optimization. To ensure the results of each prediction are consistent with one-another, their models are constructed using the same data set. As we maintain a separate regression model for each optimization, and each model is trained with a subset of this overall data set. Each prediction model is implemented by using the *lm*() function from the *stats R* package.

Problem	1D	2D	3D
T_blk	{1,8,64}	{1,8,64}	{1,8,64}
X_blk	{512,1024,2048}	{4,16,64}	{1,4,16}
Y_blk		{256,512,1024}	{2,8.32}
Z_blk			{64,128,256}

Table 2. Tuning Parameter Values for Prediction

Algorithm 2 details the underlying logistics of this process using a function, $PredictBestOpt$, which takes three inputs of an input stencil, D, its domain-level properties, E, its execution configuration, and HP, the hardware properties where it will run; and has two outputs: $best\_opt$, the optimization which should provide the highest speedup, and $best\_cfg$, the tuning parameters which should yield the highest speedup for $best\_opt$. The algorithm has two steps. *Step 1* selects the optimization which should provide the highest speedup for the stencil. It predicts the maximum speedup of each supported optimization, and the optimization which is predicted to have the highest speedup is stored in $best\_opt$. The regression model is selected using get_opt_model by looking up the predictor for the optimization in the current iteration. The maximum speedup is predicted by calling $predict\_max\_speedup$ with the model and predictor variables. *Step 2* predicts the optimal configuration for the selected optimization's tuning parameters. In particular, it predicts the actual speedup that the optimization would provide under various different configurations of tuning parameters, with each configuration stored in T. The function get_cfg_model determines the correct regression model to use with a mapping based on the selected optimization and spatial dimensionality of the stencil. The speedup is then predicted by calling $predict\_real\_speedup$. Where T stores the tuning parameters that were used to train the model on a specific machine, the function $scale\_for\_machine$ updates these values for the target architecture.

The tuning parameters that the model currently considers are the blocking factors of loop tiling optimizations and the number of threads for parallelization. We consider four blocking factors: one for the time dimension t, and three for the spatial dimensions x, y, and z. Table 2 provides the values we consider for each blocking factor. The values we consider for each blocking factor depends on the spatial dimensionality of the problem, as the size of a block is determined

Algorithm 2 Optimization Prediction Algorithm

```
function PredictBestOpt(D, E, HP)
    best_opt ← ϵ                          ▷ Step 1: Predicting the best optimization
    speedup_max ← 0
    for each opt ∈ SUPPORTED_OPTIMIZATIONS do
        opt_model ← get_opt_model(opt)
        predictors ← D ∪ E
        speedup_ϕ ← predict_max_speedup(opt_model, predictors)
        if speedup_ϕ > speedup_max then
            best_opt ← opt
            speedup_max ← speedup_ϕ
        end if
    end for
    if S_max < 1 then
        return ϵ, ϵ
    end if
    best_cfg ← ϵ                          ▷ Step 2: Configuring the optimization
    speedup_max ← 0
    tuning_params ← get_tuning_params(best_opt)
    for each T ∈ enumerate_values(tuning_params) do
        cfg_model ← get_cfg_model(opt, D.spatial_dim)
        predictors ← D ∪ E ∪ T
        speedup_ϕ ← predict_real_speedup(cfg_model, predictors)
        if speedup_ϕ > speedup_max then
            scaled_cfg ← scale_for_hardware(T, HP)
            best_cfg ← scaled_cfg
            speedup_max ← speedup_ϕ
        end if
    end for
    return best_opt, best_cfg
end function
```

by the length of its sides. These values aim to individually capture when each tuning parameter is most beneficial to the effectiveness of the optimization, independent of architecture. More specifically, they represent a small, medium, and large factor with respect to the dimensionality of the problem. The small, medium, and large factors are translated to values that are more appropriate to the specific multi-core architecture where the stencil will run. In particular, we scale the tuning parameters in spatial dimensions to preserve the ratio between the working set size and L3 cache size between the training and target machines. Specifically, we use the following equation to scale the tuning parameters for each dimension

$$\forall i \in d, \beta_{it} \leftarrow \sqrt[d]{\frac{(\beta_{is})^d * C_t}{C_s}} \quad (1)$$

where d is the spatial dimensionality of the stencil, β_{is} and β_{it} are the blocking factors for the source (i.e. training) and target machines in the i dimension, and C_s and C_t are the L3 cache sizes of the source and target machines. For the number of threads to use, we consider all values lower than that given in the execution configuration, where each iteration divides the last value by two, and uses it if a power of two, otherwise the next highest power of two.

The success of our prediction model depends on our selection of domain-level properties to represent data-access patterns of an arbitrary stencil. The list of domain-level properties we currently support serves to demonstrate the usefulness of this approach and currently includes spatial dimensionality, temporal order, order of accuracy, and directional dependence, explained in details in section 3.1. The implementation of a stencil computation can be more effectively optimized by our prediction model if its entire data access pattern can be derived from these domain-level properties. Our future work will include additional properties , e.g., iterative methods, skewing, and derivative types.

CPU		Intel Xeon Gold 6130	Intel E5-2420
Cache	L1-Data	32 KB	32 KB
	L2-Private	1,024 KB	256 KB
	L3-Shared	22,528 KB	15,360 KB
Main Memory		64 GB	12 GB
Number of Cores		16	12

Table 3. Machine configuration

4.3 Training the Prediction Model

To train the prediction model with a sufficiently representative data set that encompasses the varying interactions among data-access patterns, execution configurations, and optimization profitability, we developed a stencil kernel generator which automatically generates un-optimized stencil computations from input domain-level properties. We then define a sub-domain for each supported domain-level property to enumerate valid combinations of the property values: spatial dimension: {1,2,3}, temporal order: {0, 1, 2}; order of accuracy:{2,4}, and directional dependence: {isotropic, anisotropic}. We use these values because their combinations can fully describe the data access patterns of many stencils, e.g. wave[6] and Laplacian[7]. By enumerating the variations of these properties (in contrast to enumerating the actual variations of their underlying implementations), we can focus on the semantic-level variations of the stencil computations, without being limited by idiosyncrasies of their underlying implementations (e.g., how the data is laid out in memory or whether registers/caches are reused).

For each auto-generated un-optimized stencil kernel, we create a large collection of its optimized variants by applying each optimization both individually and by combining it with other related optimizations. For example, we combine OpenMP parallelization with each distinct tiling strategy to generate optimized variants. We invoked the Pluto[9] compiler v0.12 to apply Rectangular tiling, partial/full diamond tiling, and OpenMP parallelization. Multiplicative inversion is applied by our special-purpose stencil generator. For each tiling optimization, we use the same blocking factors to train the model as for the prediction itself, where the values are detailed in table 2. This is to keep the training and prediction input constant, as the predicted values can be adjusted to match the cache size of the target architecture itself. Each differently optimized stencil variant is run under different combinations of execution configurations, including number of time steps {32, 64, 128}, input sizes {16MB, 32MB, and

65Mb}, and number of threads for combinations that include OpenMP parallelization {1,4,16}. The performance of each execution is calculated by normalizing its execution time against the total number of floating point operations performed.

5 Experimental Results

5.1 Experimental Configuration

We have implemented our applicability filter and prediction model in R. Our special-purpose stencil C code generator was implemented using the POET language [29].

We validated our regression-based prediction model by analyzing its prediction accuracy in correctly correlating domain-level properties and execution configurations with optimization effectiveness and by using it to optimize five real stencil computations: Laplacian, Poisson for electrostatics, heat, wave, and Allen-Cahn. These kernels were chosen because they are common in the scientific simulation domain, e.g. fluid mechanics[7], electromagnetics[12], elasticity[6], diffusion[5], and materials science[1]. The un-optimized versions for the heat, wave, Laplacian, and Poisson PDEs were obtained from Burkadt's website[2]. The Allen-Cahn source code was derived from the Yamanka Lab[1], which was originally written in Python, and we converted to C.

We compare the performance attained via our workflow against that attained by having the stencil computations invoke the libraries, AMReX[13] and Pochoir[26]. AMReX uses a generalized divide-and-conquer parallelization technique that is well-suited to the general computational pattern of structured grid. We use AMReX's single-node OpenMP parallelization scheme, as our work focuses on single-node optimization. This scheme uses OpenMP to distribute threads among processors, alongside rectangular tiling of the spatial domain. Pochoir is a domain-specific language for stencils that uses a cache-oblivious dynamic programming algorithm to decompose sub-regions of the grid into trapezoidal shapes, and thread-level parallelism via PThreads to compute subgrids concurrently.

All training and testing measurements were run on an Intel Xeon Gold 6130 CPU (shown in table 3). All stencil implementations were compiled with GCC 7.5.0 to generate their binaries. Each stencil was run four times, with a normalized standard deviation of 2%.

To test our prediction model across different machines, we optimized the five stencil case studies on an Intel E5-2420 machine (shown in table 3) using the previously produced model from the Xeon Gold machine. Each stencil was compiled on the Intel E5-2420 machine using GCC 4.8.5.

5.2 Accuracy of Prediction

We analyze the accuracy of our prediction model by splitting the automatically generated stencil performance data set into a training and testing set, where we analyze the accuracy and error of predicting optimization strategies of the testing set on our model built from the training set. As the user-level inputs into the prediction model are the domain-level properties of a stencil, its execution configuration, and the number of divisions in the underlying stencil, we separate the training and testing subsets based on unique combinations of these predictor variables. This results in 4,860 unique combinations. We randomly select 5% of these combinations, or 243 unique input combinations, to be used as testing data, and use the optimization combination and configuration that resulted in the highest speedup to compare our prediction against. The remaining 95% of inputs, along with all combinations of optimizations and configurations, or 433,690 differently optimized stencil variants, are used as the training set of the model.

To evaluate the accuracy when predicting the optimization strategy of the testing set, we analyze the residuals of the predictions and the relative root mean squared error (rRMSE) of the model. 66% of the predictions resulted in the optimal strategy, 78% of the predictions had speedups within 5% of the optimal speedup, and 89% of the predictions had speedups within 10% of the optimal speedup. The rRMSE was 5.3%.

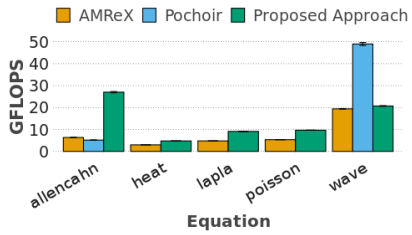


Figure 6. Comparing the performance attained from optimizing the case studies in table 4 using our proposed approach against optimized libraries

5.3 Optimization Effectiveness

Figure 6 presents the GFLOPS attained for the five PDE use cases from table 4 by comparing results attained by using our optimization workflow with those attained by invoking the optimized software libraries AMReX and Pochoir. Pochoir is only able to handle isotropic stencils, and is thereby only used on the Allen-Cahn and wave case studies.

For Allen-Cahn, our tool applied partial diamond tiling with multiplicative inversion, which resulted in 4.3x and 5.2x higher speedups than AMReX and Pochoir, respectively. In particular, the speedup over AMReX comes from specializing the tiling strategy to take better advantage of the data access pattern; the speedup over Pochoir comes from a smaller input size, resulting in Pochoir's dynamic programming approach having more overhead. Additionally, our tool applied multiplicative inversion, which neither of the other software libraries utilize. For heat, our tool applied partial diamond tiling with OpenMP parallelization with 16 threads, which resulted in a 1.6x higher speedup than AMReX. This speedup comes from specializing the tiling strategy to the specific

Stencil	*Laplacian*	*Poisson*	*Wave*	*Heat*	*Allen-Cahn*
Equation	$0 = \nabla \cdot (\alpha\,(x)\,\nabla u)$	$-\frac{\rho}{\epsilon} = \nabla \cdot (\alpha\,(x,\,y)\,\nabla u)$	$\frac{\partial^2 u}{\partial t^2} = c^2 \Delta u$	$\frac{\partial u}{\partial t} = \nabla \cdot (\alpha(x,\,y,\,z) \nabla u)$	$\frac{\partial \phi}{\partial t} = -\frac{\sqrt{2W}}{6\alpha} M \left(\frac{\partial(\phi)}{\partial \phi} (g_B - g_A) + W \frac{\partial q(\phi)}{\partial \phi} - \alpha^2 \Delta u \right)$
Dimension	1	2	2	3	3
Order of Accuracy	4	4	4	2	2
Temporal Order	0	0	2	1	1
Directional Dependence	Anisotropic	Anisotropic	Isotropic	Anisotropic	Isotropic
Timesteps	32	256	128	64	32
Input Size	131MB	16MB	134MB	80MB	33MB
N Threads	8	16	16	16	1
Is iteration space traversal of linear order	T	T	T	T	T
Is serial execution	T	T	T	T	T
Is constant denominators per LUP > 0	F	T	F	F	T

Table 4. Equation, domain-level properties, execution configurations, initial opportunity indicator values

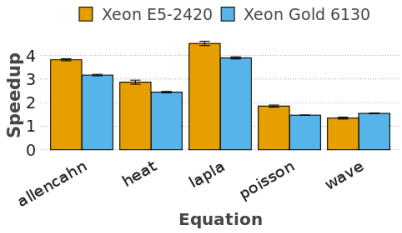


Figure 7. Comparing the speedups attained using our optimization approach using different machines

data access pattern. For Laplacian, our tool only applied rectangular tiling, which resulted in a 1.9x higher speedup than AMReX. In this case, both our proposed strategy and AMReX use rectangular tiling; however, the speedup from our approach comes from the lack of parallelization. The 8 threads used by AMReX incurred additional overhead that our strategy recommended against. For Poisson, our tool applied full diamond tiling with OpenMP parallelization with 16 threads, which resulted in a 1.8x higher speedup than AMReX. This speedup comes from specializing the tiling strategy to the specific data access pattern. For wave, our tool applied full diamond tiling with OpenMP parallelization with 16 threads, which resulted in 1.1x higher speedup than AMReX, but a 60% slowdown compared to Pochoir. Here, Pochoir outperforms our strategy, as the overhead induced by dynamic programming is lessened due to the larger input size of the program. In comparison to AMReX, our strategy specializes a tiling strategy to attain a higher level of performance. Overall, the key advantages of our proposed strategy are 1) individual optimizations can be strategically selected from a choice of multiple with similar purposes, 2) each optimization can be specially configured to the problem at hand, and 3) optimizations can be strategically disabled, if its application will result in slowdown.

5.4 Reproducibility Across Architectures

Here, we analyze the consistency of the optimization strategies tailored by our model work on machines of differing numbers of cores and cache/memory sizes. In particular, we optimize the same five stencils from table 4 on the Intel E5-2420 machine shown in table 3 which has 12 cores, 12GB of main memory, and a 16 MB L3 cache size; compared to the machine which trained the model, which has 16 cores, 64GB of main memory, and a 23 MB L3 cache size. The blocking factors for the tiling optimizations are scaled to match this machine's L3 cache using equation 1. The resulting difference in speedups for the Allen-Cahn, heat, Laplacian, Poisson, and Wave equations are then 17%, 15%, 14%, 21%, and 13% between the two machines.

6 Related Work

Our research belongs in the general area of performance analysis and optimization for computational workloads. Specifically, we investigate performance modeling and optimization prediction for stencil computations.

Exploring analytical modeling to predict performance and optimizations of stencils has been widely studied. Hammer[16] and Laukemann[19] develop tools to predict performance and scaling behavior of loops using the roofline[27] and execution-cache-memory[25] analytical models. Guerrera[15] leverages analytical models to generate optimized stencil kernels for specific architectures. In contrast to using analytical modeling to predict performance, our work uses code-level properties to indicate a potential presence of a bottleneck, and uses regression modeling to predict the best optimization to eliminate the identified bottleneck.

Rahman [22] presents a stencil optimization process which similarly uses regression analysis of stencil performance. In particular, their work derives formulas that will predict a stencil's execution time using hardware counters as input, and uses this to compose and tune optimizations to reduce bottlenecks on certain hardware resources. Our work, in contrast, leverages domain-level knowledge about the stencil at hand to predict optimization profitability.

Domain specific languages (DSLs) provide an alternative method to attain high performance applications using higher-level specifications of the problem at hand, in

which a special purpose compiler translates into machine code. Prominant examples of stencil DSLs are Pochoir[26], which uses a divide-and-conquer parallelization algorithm, and Snowflake [30], which uses micro-compilers to perform multiple optimization passes over its AST to generate more customizable code. Our work, in contrast, identifies optimizations for existing code using higher-level properties of an input stencil rather than generating new code from a higher-level specification.

7 Conclusion

To summarize, we show that domain-level properties can be used to predict effective optimization strategies for stencils. We define a repeatable two step optimization approach that first identifies an initial set of optimizations that are compatible with the stencil, and then uses regression to predict the most beneficial optimization among them. We use our optimization approach to optimize five different partial differential equations to attain an average of 2.4x higher speedup over two highly-optimized stencil libraries.

References

[1] [n.d.]. Allen-Cahn-2d. http://web.tuat.ac.jp/~yamanaka/pcoms2019/Allen-Cahn-2d.html (Accessed on 08/19/2020).

[2] [n.d.]. C Codes. https://people.sc.fsu.edu/~jburkardt/c_src/c_src.html (Accessed on 10/18/2020).

[3] [n.d.]. Is division slower than multiplication? | searchivarius.org. http://searchivarius.org/blog/division-slower-multiplication (Accessed on 07/06/2020).

[4] [n.d.]. Multiple Linear Regression with Interactions | Introduction to Statistics | JMP. https://www.jmp.com/en_us/statistics-knowledge-portal/what-is-multiple-regression/mlr-with-interactions.html (Accessed on 10/18/2020).

[5] 2020. FD1D_HEAT_EXPLICIT - Time Dependent 1D Heat Equation, Finite Difference, Explicit Time Stepping. https://people.sc.fsu.edu/~jburkardt/cpp_src/fd1d_heat_explicit/fd1d_heat_explicit.html (Accessed on 10/02/2020).

[6] 2020. FD1D_WAVE - Finite Difference Method, 1D Wave Equation. https://people.sc.fsu.edu/~jburkardt/cpp_src/fd1d_wave/fd1d_wave.html (Accessed on 10/02/2020).

[7] 2020. LAPLACIAN - The Discrete Laplacian Operator. https://people.sc.fsu.edu/~jburkardt/cpp_src/laplacian/laplacian.html (Accessed on 10/02/2020).

[8] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. 2016. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (2016), 1285–1298.

[9] Uday Bondhugula and Jagannathan Ramanujam. 2007. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. (2007).

[10] José María Cecilia, José Manuel García, and Manuel Ujaldón. 2010. CUDA 2D stencil computations for the Jacobi method. In *International Workshop on Applied Parallel Computing*. Springer, 173–183.

[11] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[12] Steven Ellingson. 2018. *Electromagnetics Volume 1 (beta)*. Virginia Tech Libraries.

[13] Zhang et al. 2019. AMReX: A Framework for Block-Structured Adaptive Mesh Refinement. *Journal of Open Source Software* 4, 37 (2019), 1370. https://doi.org/10.21105/joss.01370

[14] Bengt Fornberg. 1988. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of computation* 51, 184 (1988), 699–706.

[15] Danilo Guerrera. 2021. Stempel. https://github.com/RRZE-HPC/stempel

[16] Julian Hammer, Georg Hager, Jan Eitzinger, and Gerhard Wellein. 2015. Automatic loop kernel analysis and performance modeling with kerncraft. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. 1–11.

[17] Intel Intel. 64. and IA-32 Architectures Optimization Reference Manual, September 2014.

[18] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2007. Effective automatic parallelization of stencil computations. *ACM sigplan notices* 42, 6 (2007), 235–244.

[19] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. 2018. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM performance modeling, benchmarking and simulation of high performance computer systems (PMBS)*. IEEE, 121–131.

[20] Pei-Hung Lin, Qing Yi, Daniel Quinlan, Chunhua Liao, and Yongqing Yan. 2016. Automatically optimizing stencil computations on many-core NUMA architectures. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 137–152.

[21] Brandon Nesterenko, Qing Yi, and Jia Rao. 2018. Improving resource utilization through demand aware process scheduling. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[22] Shah M Faizur Rahman, Qing Yi, and Apan Qasem. 2011. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*. 1–10.

[23] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. 2013. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems* 25, 3 (2013), 695–705.

[24] Andreas Schäfer and Dietmar Fey. 2011. High performance stencil code algorithms for GPGPUs. *Procedia Computer Science* 4 (2011), 2027–2036.

[25] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. 2015. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 207–216.

[26] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 117–128.

[27] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[28] Michael Wolfe. 1989. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. 655–664.

[29] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. 2007. POET: Parameterized optimizations for empirical tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[30] Nathan Zhang, Michael Driscoll, Charles Markley, Samuel Williams, Protonu Basu, and Armando Fox. 2017. Snowflake: A lightweight portable stencil dsl. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 795–804.

Acknowledgments

Prepared by LLNL under Contract DE-AC52-07NA27344 with IM release number LLNL-CONF-815179