Fast Stencil Computations using Fast Fourier Transforms

Zafar Ahmad  
Stony Brook University  
zafahmad@cs.stonybrook.edu

Rezaul Chowdhury  
Stony Brook University  
rezaul@cs.stonybrook.edu

Rathish Das  
University of Waterloo  
rathish.das@uwaterloo.ca

Pramod Ganapathi  
Stony Brook University  
pramod.ganapathi@cs.stonybrook.edu

Aaron Gregory  
Stony Brook University  
afgregory@cs.stonybrook.edu

Yimin Zhu  
Stony Brook University  
yimzhu@cs.stonybrook.edu

ABSTRACT

Stencil computations are widely used to simulate the change of state of physical systems across a multidimensional grid over multiple timesteps. The state-of-the-art techniques in this area fall into three groups: cache-aware tiled looping algorithms, cache-oblivious divide-and-conquer trapezoidal algorithms, and Krylov subspace methods. In this paper, we present two efficient parallel algorithms for performing linear stencil computations. Current direct solvers in this domain are computationally inefficient, and Krylov methods require manual labor and mathematical training. We solve these problems for linear stencils by using DFT preconditioning on a Krylov method to achieve a direct solver which is both fast and general. Indeed, while all currently available algorithms for solving general linear stencils perform $\Theta(NT)$ work, where $N$ is the size of the spatial grid and $T$ is the number of timesteps, our algorithms perform $o(NT)$ work.

To the best of our knowledge, we give the first algorithms that use fast Fourier transforms to compute final grid data by evolving the initial data for many timesteps at once. Our algorithms handle both periodic and aperiodic boundary conditions, and achieve polynomially better performance bounds (i.e., computational complexity and parallel runtime) than all other existing solutions.

Initial experimental results show that implementations of our algorithms that evolve grids of roughly $10^4$ cells for around $10^3$ timesteps run orders of magnitude faster than state-of-the-art implementations for periodic stencil problems, and $1.3 \times$ to $8.5 \times$ faster for aperiodic stencil problems.

Code Repository: https://github.com/TEAlab/FFTStencils

1 INTRODUCTION

A stencil is a pattern used to compute the value of a cell in a spatial grid at some time step from the values of nearby cells at previous time steps. A stencil computation [49, 130] applies a given stencil to the cells in a spatial grid for some set number of timesteps. Stencil computations have applications in a variety of fields including fluid dynamics [21, 45, 68], electromagnetics [10, 82, 128, 140], mechanical engineering [111, 113, 127], meteorology [11, 75, 114, 115], cellular automata [94, 102, 122, 123], and image processing [107, 117, 142, 146, 147]. In particular, they are widely used for simulating the change of state of physical systems over time [14, 105, 134, 143].

Due to the importance of stencil computations in scientific computing [41, 59, 108], various methods have been devised to improve their runtime performance on different machine architectures [37, 42, 76, 101, 120]. All currently available stencil compilers [25, 37, 65, 67, 92, 109, 130] that can accept arbitrary linear stencils perform $\Theta(NT)$ work\(^2\), where $N$ is the number of cells in the spatial grid and $T$ is the number of timesteps.

In this paper, we present the first $o(NT)$-work stencil computation algorithms that support general linear stencils and arbitrary boundary conditions. Our algorithms have polynomially lower work than all other known options of equivalent or greater generality.

Problem Specification. Consider a stencil computation to be performed over $T$ timesteps on a $d$-dimensional spatial grid of $N$ cells with initial data $u_0, \cdots$. Cell data at subsequent timesteps are defined via application of the linear stencil $S$ across the grid, formalized as $u_{t+1} = Su_t$, where $u_t$ is the spatial grid data at timestep $t$. The stencil must define the value of a grid cell in terms of a fixed size neighborhood containing cells from only the prior timestep\(^3\). We will not be able to apply the stencil to some cells near the boundaries of the grid; the values of these cells are instead defined via boundary conditions. Our goal is to compute the final grid data $u_T$ by evolving the initial data $u_0$ for $T$ timesteps.

There are two types of boundary conditions we can use: periodic and aperiodic. If the boundary conditions are periodic, it means that every dimension of the spatial grid wraps around onto itself, so the entire grid forms a torus. In this case modular arithmetic is used for all calculations involving spatial indices, and the stencil alone can be used to update all cells. On the other hand, if the boundary conditions are aperiodic, then the cells at the boundary of the grid have to be computed via some method other than straightforward application of the stencil. In this paper, we consider both types of boundary conditions.

Existing Algorithms. There are a handful of algorithms commonly used for carrying out general stencil computations, and many more designed for solving problems with specific boundary conditions and stencils. Here we will give an overview of algorithms

---

\(^1\) A linear stencil is one that uses exclusively linear combinations of grid values from prior timesteps.

\(^2\) Let $T_P$ denote a program’s runtime on a $p$-processor machine. Then, $T_1$ and $T_p$ are called work and span, respectively.

\(^3\) We will later extend the definition of stencils to allow for dependence on multiple prior timesteps.
1. [Looping Algorithms.] It is a simple matter to implement stencil computations using nested loops, as shown in Figure 1. However, such computations suffer from poor data locality and hence are inefficient both in theory and in practice. As can be seen from the pseudocode listing, looping codes require Θ(NT) work to evolve a grid of N cells forward T timesteps, assuming that the stencil only uses values from a constant size neighborhood.

The most common framework that can be used to derive tiled loop implementations is the polyhedral model [2–4, 141, 155], which uses a set of hyperplanes to partition the grid being solved for. The polyhedral model is extensively used in several code generators [2–4, 141, 155].

2. [Tiled Looping Algorithms.] Adopting a tiling [12, 24] of the spatial grid is a common way to improve the data locality [23, 24, 153] of looping algorithms. The tiles’ size and shape have a strong influence on the runtime of the algorithm, and generally the best performance is attained when each tile fits into the cache. Most modern multicore machines have a hierarchy of caches; to make better use of the cache hierarchy, loop nests may need to be tiled at multiple levels [42, 77].

The most common framework that can be used to derive tiled loop implementations is the polyhedral model, which uses a set of hyperplanes to partition the grid being solved for. The polyhedral model is extensively used in several code generators [2–4, 141, 155].

3. [Recursive Divide-and-Conquer Algorithms.] Instead of using a predetermined tiling of the spatial grid, these algorithms recursively break the region to be solved for into multiple smaller sub-regions. The trapezoidal decomposition algorithm [50, 51, 131] is the most well known divide-and-conquer stencil algorithm. Its recursive approach to tiling allows it to be not only both cache-oblivious [48] and cache-adaptive [16, 17], but also to achieve asymptotic cache performance matching that of an optimally tiled stencil code across all levels of the memory hierarchy in a multicore machine.

4. [Krylov Subspace Methods.] Krylov methods compose a diverse set of mathematical techniques which are extensively used in numerical analysis to find successively better approximations of the exact solution to a stencil problem. Such methods are often used to solve problems for which there is no known direct (tile-loop or divide-and-conquer) solution technique. Discrete Fourier transforms (DFTs) are frequently used in the analysis [27] and implementations [7, 62, 74] of these methods. Krylov methods that use DFTs in their implementations are very restricted in their applicability, usually applying only to stencils from specific PDEs that benefit from spectral analysis.

There are several limitations of Krylov methods as a whole: (i) their initial design requires nontrivial manual convergence analysis [29, 87, 103], (ii) they are mostly applicable only to very small classes of problems [8, 56, 74], (iii) they generally do not produce exact solutions in finite time, but exhibit a trade-off between runtime and accuracy. Improving this trade-off by finding near-optimal preconditioners [18, 19, 135] is a hard problem [33, 81] in the general case.

These common limitations should not be confused for rules, however: because of their diversity, Krylov methods can take on a variety of useful properties when specially designed. For example, when an optimal preconditioner is selected they can find the exact solution in a finite number of iterations. The algorithms we present in this paper will be partially based on an optimally preconditioned Krylov method that is applicable to a rather large class of stencil problems.

Our FFT-Based Algorithms. The computation $S^T \ a_0 = a_T$ (where $S^T$ denotes that the stencil $S$ is applied $T$ times) evolves the initial grid data $a_0$ for $T$ timesteps to produce the final data $a_T$. As will be seen in Section 4, any method of computing $a_{t+1} = S \ a_t$ is mathematically equivalent to a product where $S$ is viewed as a matrix and $a_t$ as a vector. All existing algorithms which find $a_T$ exactly do so by direct computation of $a_T = S(S(\cdots a_0)))$, where $S$ is applied for a total of $T$ timesteps, incurring $\Theta(NT)$ work in the process. The looping, tiled, and recursive algorithms we have described differ only in how they break up this series of matrix-vector products. We will instead evaluate this product by diagonalization and repeated squaring of the stencil matrix $S$.

In this paper, we present two FFT-based stencil algorithms: StencilFFT-P for problems with periodic boundary conditions, and StencilFFT-A for those with aperiodic boundary conditions. Our algorithms are applicable to arbitrary uniform linear stencils across vector-valued fields.

1. [Periodic Stencil Algorithm.] Let the discrete Fourier transform (DFT) matrix be written $\mathcal{F}[i, j] = \omega_N^{-ij} / \sqrt{N}$, where $\omega_N = e^{2\pi i / N}$, and let $\mathcal{F}^{-1}$ be the inverse DFT matrix. We use fast Fourier transforms to compute $a_T$ as follows:

$$a_T = \mathcal{F}^{-1}(\mathcal{F} S \mathcal{F}^{-1})^T a_0$$

where $\mathcal{F} S \mathcal{F}^{-1}$ is a diagonal matrix, and $T$ = #timesteps (not a transposition).

To the best of our knowledge, this is the first time that FFT is being applied directly to the problem of computing integral powers of the circulant [61] stencil matrix $S$ that appears in linear stencil computations, even though there is a strong history of using FFT to improve the efficiency of matrix computations [43, 58, 136].

2. [Aperiodic Stencil Algorithm.] When given aperiodic boundary conditions, we use a recursive divide-and-conquer strategy to solve for the boundary of the spatial grid; StencilFFT-P is used as a subroutine to compute cells whose values are independent of the boundary. This method allows us to compute every timestep of the boundary in serial, yet to skip over computing most timesteps of cells near the middle of the grid.

Before we analyze the complexities of our algorithms briefly described above, we give the performance metrics that will be used.

Performance Metrics. We use the work-span model [39] to analyze the performance of dynamic multithreaded parallel programs. Work $T_1(n)$ of an algorithm, where $n$ is the input parameter, denotes the total number of serial computations. Span $T_{ce}(n)$ of an
algorithm, also called critical-path length or depth, denotes the maximum number of operations performed on any single processor when the algorithm is run on a machine with an unbounded number of processors. Our analysis of span is performed according to the performance analysis of our algorithms on a KNL/SKX node.

**Our Contributions.** Our key contributions are as follows:

1. **[Theory.]** We present the first algorithms for general linear stencil computations (for both periodic and aperiodic boundary conditions) with $\Theta(NT)$ work and low span, achieving polynomial speedups over the best existing stencil algorithms.

2. **[Practice.]** We experimentally analyze the numerical accuracy and runtime of our algorithms as compared to PLuTo [3] code. Implementations of our algorithms for on the order of $10^5$ grid cells and $10^5$ timesteps suffer no more loss in accuracy from floating point arithmetic than PLuTo code, yet run orders of magnitude faster than the best existing implementations of state-of-the-art algorithms for periodic stencil problems, and 1.3x to 8.5x faster for aperiodic stencil problems. This is shown in Table 7. Our code is publicly available at https://github.com/TEAlab/FFTStencils.

## 2 RELATED WORK & ITS LIMITATIONS

There is substantial literature devoted to the applications and analysis of stencils and Discrete Fourier Transforms (DFTs). Here we give a background of the relationship between these two areas and examine some of the limitations which appear in the current approaches to stencil codes.

**Discrete Fourier Transforms.** DFTs are widely used in numerical analysis, with examples including Von Neumann stability analysis [106, 148] to show validity of numerical schemes, DFT-based preconditioning to optimize Krylov iterations [31, 63, 79], and time-domain analysis to achieve partial solutions of given PDEs [36, 69, 97, 104, 121].

A Fast Fourier Transform (FFT) is an algorithm that quickly computes the DFT of an array. The use of FFTs will be important for our algorithms, as they represent a particularly efficient type of matrix-vector multiplication. Several $O(N \log N)$-work FFT algorithms exist [30, 60, 149], the most famous among which is the Cooley-Tukey algorithm [38].

**Theorem 2.1 (Cooley-Tukey Algorithm, [38, 48]).** The generic Cooley-Tukey FFT algorithm computes the DFT of an array of size $N$ in $\Theta(N \log N)$ work, $O(\log N \log \log N)$ span, and $\Theta(N)$ space.

**Stencil Problems.** Stencils are often used in numerical analysis as discretizations of PDEs, since many simple PDEs have prohibitively complex analytical solutions [46, 110] but allow good numerical approximations with a proper choice of stencil.

There are two major types of methods related to stencils: those for deriving numerical schemes, and those for evolving grid data via a given stencil. A discretization method is a way of converting a PDE, which deals with quantities defined over a continuum, into a stencil, which relates quantities defined over discrete sets of variables. A stencil solver is an algorithm that takes stencils, boundary conditions, and initial data as input and performs stencil computations to output final data. This paper presents a pair of stencil solvers for linear stencils which support, respectively, periodic and aperiodic boundary conditions.

**Stencil Solvers.** Numerical results from stencils are obtained through two primary paths: direct solvers and Krylov methods [70, 118].

Direct solvers are those which find the solution to a stencil problem in a finite number of steps. They often involve feeding the

---

### Table 1: Complexity bounds for stencil algorithms, where $N = $ spatial grid size, $T = $ timestep, and $M = $ cache size. It is important to note that the bounds given for our algorithms are for computations with $\Theta(1)$-size stencils on $d = \Theta(1)$ dimensional hypercubic grids, and that we have simplified the span of StencilFFT-A by assuming $T = \Theta(\log N \log \log N)$. The nested loop, tiled loop, and D&C algorithms work for both periodic and aperiodic boundary conditions.

| Algorithm      | Work ($Td$) | Span ($\frac{T}{d}$) | Result |
|----------------|-------------|-----------------------|--------|
| Nested Loop    | $\Theta(NT)$ | $\Theta(T \log N)$ | [12, 24] |
| Tiled Loop     | $\Theta(NT)$ | $\Theta(T \log N + \frac{1}{d} \log \frac{N}{d})$ | [131] |
| D&C            | $\Theta(NT)$ | $\Theta(T \frac{N^{1/4} \log (d+1)}{d})$ | [131] |

### Table 2: Performance summary of parallel stencil algorithms on a KNL/SKX node.

| Benchmark | Stencil | $N$ | $T$ | Parallel runtime in seconds | Speedup factor over PLuTo |
|-----------|---------|-----|-----|-----------------------------|---------------------------|
|           |         |     |     | KNL | SKX | KNL | SKX | KNL | SKX | KNL | SKX |
| Heat 1d   | 6,000,000 | $10^6$ | 79 | 0.25 | 0.03 | 1784.3 | 769.6 |
|           | 5,000,000 | $10^6$ | 1.437 | 0.48 | 0.61 | 3,025.0 | 367.0 |
|           | 4,000,000 | $10^6$ | 500 | 0.48 | 0.64 | 1,032.7 | 1266.6 |
|           | 3,000,000 | $10^6$ | 2,895 | 10.7 | 0.48 | 6,000.7 | 1502.1 |
|           | 2,000,000 | $10^6$ | 816 | 0.08 | 0.28 | 10.9 | 20.7 |
|           | 1,500,000 | $10^6$ | 141 | 0.08 | 0.28 | 21.3 | 27.5 |
| Heat 1d   | 6,000,000 | $10^6$ | 50 | 0.08 | 0.69 | 8.5 | 7.2 |
|           | 5,000,000 | $10^6$ | 544 | 0.11 | 0.88 | 23.4 | 24.2 |
|           | 4,000,000 | $10^6$ | 1,245 | 3.52 | 3.7 | 4.0 | 4.2 |
|           | 3,000,000 | $10^6$ | 436 | 0.12 | 0.48 | 17.5 | 17.8 |
|           | 2,000,000 | $10^6$ | 156 | 0.08 | 0.22 | 4.7 | 5.5 |
|           | 1,500,000 | $10^6$ | 71 | 0.08 | 0.22 | 6.0 | 7.2 |

### Table 3: Numerical results from stencils are obtained through two primary paths: direct solvers and Krylov methods [70, 118]. Direct solvers are those which find the solution to a stencil problem in a finite number of steps. They often involve feeding the
stencil into a stencil compiler such as PLuTo [25], Pochoir [130], or Devito [91], which will output optimized code to compute the action of the stencil across some prespecified grid of initial data for multiple timesteps. Cutting-edge stencil code generators feature many improvements over simple looping algorithms, including better cache efficiency [49, 85], parallelism [83], and low-level compiler optimizations. These systems all perform the same set of updates on the stencil grid, although they vary in the order that these updates are performed. In general they make no use of FFTs.

Krylov subspace methods produce successively better approximations of the exact solution to a given stencil problem. Krylov solvers are often used to solve problems for which there is no known direct solution technique. It is common for DFTs to be used either in the analysis [27] or implementations [7, 62, 74] of Krylov subspace methods, as Fourier analysis is useful for proving scheme stability [32, 156] and convergence rates [57], and DFT matrices are good preconditioners [44] for a large class of matrix equations [26]. In some instances choosing the DFT matrix as a Krylov preconditioner can even convert an approximate solver into a direct one [54, 69].

Krylov subspace methods generally do not produce exact solutions in finite time, but exhibit a trade-off between runtime and accuracy.

### Limitations of Current Methods for Stencil Problems.

1. **Manual Analysis.** Krylov subspace methods are often accompanied by a mathematically nontrivial convergence analysis [29, 87, 103]; this requires human labor for every new method developed. Since this analysis is not automated [53, 64, 99], the quantity of time it takes varies widely from case to case. Also, the requirement of mathematical rigour strongly discourages the development of unnecessarily general Krylov methods, thus those in the literature usually only apply to specific stencils [8, 9, 56, 93] in order to simplify analysis.

2. **Specialization.** The methods [8, 56, 74, 121] published in most of the existing literature on computation with numerical schemes are only applicable to very small classes of problems. DFT preconditioning for Krylov iterations has been used for specific stencils before [52, 72, 124]. However, these techniques have not been generalized to work for higher dimensional grids with general linear stencils.

3. **Inexact Solution.** Krylov methods often cannot produce exact solutions, even in the absence of floating-point rounding errors. Using them optimally and reliably thus requires expertise in numerical analysis [70, 137], as well as for substantial effort to be put into uncertainty quantification [86, 89].

4. **Nonoptimal Computational Complexity.** All currently available code compilers [25, 37, 65, 67, 92, 109, 130] generate code which has linear work complexity in the number of grid cells and number of time steps to compute, no matter what stencil they are given. Improving this bound has not been addressed in the literature, even when only considering linear stencils.

   We show in this paper that when dealing with linear stencils it is possible to produce code that has significantly better asymptotic performance.

5. **No Support for Implicit Stencils.** Direct solvers for stencil problems usually do not support implicit stencils; stencil compilers in particular are weak in this respect. Neither Pochoir, PLuTo, nor Devito can be used for directly evolving data via an implicit stencil. This is a significant limitation, as several important stencils are implicit [6, 40, 80, 100, 115, 138].

### Significance of This Paper.

Current direct solvers for general linear stencil computations are inefficient, and Krylov methods require manual labor and mathematical training. We solve these problems for linear stencils by using DFT preconditioning on a Krylov method to achieve a direct solver which is both fast and general.

## 3 Applicability of Our Algorithms

In this section, we describe the classes of stencil problems on which our FFT-based stencil algorithms do or do not apply.

### Supported Stencil Types.

Our algorithms are most directly applicable to **homogeneous linear stencils across vector-valued fields.** A homogeneous stencil is one that does not vary across the entire spatial grid, and by vector-valued fields we mean we allow each cell value across the spatial grid to be treated as a vector.

All homogeneous linear PDEs can be discretized into supported stencils by using a finite difference approximation [71]. Thus all numerical results for these linear PDEs that were previously reached via analytically-motivated numerical schemes, including those set in the Fourier domain [95, 97], can easily be reached computationally by our algorithms.

It is noteworthy that we support both explicit and implicit linear stencils, and also that vector-valued fields can be used to enable stencils that are dependent on more than one previous timestep. In addition, vector-valued fields allow us to support certain types of inhomogeneity, as mentioned at the end of this section.

Linear stencils are quite common in computational numerics. In fact, the majority of stencils currently used to benchmark [3, 23, 25, 112] stencil compilers are linear.

### Unsupported Stencil Types.

Our algorithms are not applicable to **nonlinear stencils.** This is because introducing nonlinearity of any sort invalidates our technique of using DFTs to simplify the action of the stencil. Common examples of nonlinearity in stencils include conditionals, i.e. max/min/if-else, and quadratic dependence on cell values. Most discretizations of nonlinear PDEs pass the nonlinearity on to the stencil, so in general our algorithms cannot be used for stencils from nonlinear PDEs.

Our algorithms cannot be applied to **inhomogeneous stencils** either. There are two ways that a stencil can break homogeneity. The first is spatially, by having the stencil itself be dependent on local field data, such as is used in slope limiter and flux limiter methods [55, 88, 126] and in mixed media problems [34, 119, 132, 133, 144, 154]. The second way to break homogeneity is temporally, i.e. using a stencil that is dependent on time [66, 84], as would arise from the presence of a forcing term in the original PDE being discretized.

However, we note that there are some special types of inhomogeneity our algorithms can handle, such as those arising from forcing terms which are low-order polynomials in time. These are handled by discretizing homogeneous systems of PDEs to mimic the behaviour of a single inhomogenous PDE.
4 PERIODIC STENCIL ALGORITHM

In this section we present StencilFFT-P, an efficient parallel algorithm for performing stencil computations with periodic boundary conditions using fast Fourier transforms (FFT). We begin by considering explicit linear stencils on one-dimensional spatial grids, after which we give simple extensions to high-dimensional grids, implicit stencils, and grids where cells are vector-valued.

Mathematical Formulation. Suppose we have a spatial grid of data that evolves in time according to some fixed stencil: cells in the grid at time \( t \) are updated as a function of some local neighborhood of cell values at recent times before \( t \). For simplicity’s sake, we will assume that the spatial grid is one dimensional until Section 4.2.

In this section, we will exclusively consider linear stencils. A linear stencil \( S \) defines future array values \( a_{t+1}[0, \ldots, N-1] \) as a linear function of current array values \( a_t[0, \ldots, N-1] \). We will later allow array values to be higher dimensional, but for now these constraints are enough on their own to make a significant statement about stencils.

An arbitrary linear mapping from arrays of size \( N \) to arrays of size \( N \) is, by definition, an matrix \( S \), and therefore the update rule can be written \( a_{t+1}[i] = \sum_j S[i,j] a_t[j] \). We will usually omit the indices in formulas like this, writing \( a_{t+1} = S a_t \). As shown in Figure 2, on the surface this update rule may look incomplete - since we require \( S \) to use the same neighborhood around each point for updates, how should we proceed when the neighborhood extends beyond the bounds of our spatial array? As filling in these cells is exactly the purpose of boundary conditions, it should come as no surprise that in this section the resolution to this difficulty will come in the form of periodicity.

Periodic boundary conditions consist of the rule that \( a_t[i] = a_t[i + N] \), for all \( i \). This allows the spatial grid to be extended arbitrarily far in either direction by wrapping around instead of moving outside of the array bounds; in the presence of periodic boundary conditions the spatial grid is effectively a torus, with no clearly defined boundary.

For a stencil to be uniform across space means that it defines updated cell values only from a set of cells which are selected based on their relative location to the cell being updated. For example, if we were to reindex all cells in the spatial array, incrementing them all so the bounds became 1 and \( N \) rather than 0 and \( N-1 \), this change of index ought to be invisible to the stencil. Furthermore, in the presence of periodic boundary conditions, this reindexing is equivalent to cyclically shifting the field data \( a_t \), since we have \( a_t[0] = a_t[N] \).

We mathematically represent the concept of cyclically permuting grid data by introducing the right shift matrix\(^9\) \( [116] X \). The array \( Xa_t \) is defined to be the result of taking the rightmost element \( a_t[0] \) off and appending it to the left side of the array, i.e. its action on arrays is \((Xa_t)[i] = a_t[i-1]\). An equivalent definition is by \( X \)'s matrix elements \( X[i,j] = \delta_{i,j+1} \), where the Kronecker delta \( \delta_{i,j} \) is defined to be 1 if \( i = j \) and 0 otherwise, and the arithmetic is understood to be modular with base \( N \) in the presence of periodic boundary conditions.

![Figure 3: The stencil \( S \) does not care where cells are with respect to the start of the array; it only cares about where they are with respect to one another. The shift matrix \( X \) does not affect cells’ relative locations, so it does not change the values of the updated cells.](image)

Given that we have periodic boundary conditions and the update rule is fixed across space, the action of our stencils must be invariant under spatial shifts of the grid values. As shown in Figure 3, cyclically permuting \( a_t \) and then applying \( S \) should give the exact same result as applying \( S \) and then cyclically permuting \( a_{t+1} \). In symbols, we have \( S X a_t = X S a_t \), which implies that \( S \) must be a circulant matrix \([61]\), satisfying \( S[i,j] = S[i-j,0] \). If we name these elements \( S[i-j,0] = s_{i-j} \), we can write out the full update equation as follows:

\[
\begin{pmatrix}
  a_{t+1}[0] \\
  a_{t+1}[1] \\
  \vdots \\
  a_{t+1}[N-2] \\
  a_{t+1}[N-1]
\end{pmatrix} =
\begin{pmatrix}
  s_0 & s_{N-1} & \cdots & s_2 & s_1 \\
  s_1 & s_0 & \cdots & s_2 & s_1 \\
  \vdots & \vdots & \ddots & \vdots & \vdots \\
  s_{N-2} & s_{N-3} & \cdots & s_1 & s_0 \\
  s_{N-1} & s_{N-2} & \cdots & s_1 & s_0
\end{pmatrix}
\begin{pmatrix}
  a_t[0] \\
  a_t[1] \\
  \vdots \\
  a_t[N-2] \\
  a_t[N-1]
\end{pmatrix}
\]

A useful representation of circulant matrices is found through the use of shift matrices, which act on a circulant matrix by cyclically shifting its elements. An example of this matrix with concrete numbers is constructed in Appendix A.2.

A useful representation of circulant matrices is found through the use of shift matrices, which act on a circulant matrix by cyclically shifting its elements. An example of this matrix with concrete numbers is constructed in Appendix A.2.

Under periodic boundary conditions, shifting the array is equivalent to rotating the array.

---

\(^{9}\)Here we are using the word matrix in the strictly mathematical sense, i.e. the object which is a stencil behaves in all respects identically to the way in which a matrix behaves. This should not be taken to mean that we will store stencils using the data structure called a matrix. In fact, we shall show that there are other more efficient ways of storing our stencils.
Reformulating the Final Data. We now turn back to the definition of the final data $a_T = S^T a_0$ and the DFT matrix $F$. Here the exponent $T$ will always denote a matrix power, not a transposition. As before, the DFT matrix has elements $F[i, j] = e^{2\pi i N^{-1} ij}$, where $e^{2\pi i N^{-1} ij}$ is a primitive $N$th root of unity, and $F$'s inverse has elements $F^{-1}[i, j] = e^{i N^{-1} jN i}$. Since we know that $F^{-1}F$ is the identity, it can make no difference mathematically to drop it into our equation for the final data:

$$a_T = F^{-1} S^T F^{-1} F a_0.$$  

Continuing to insert identities and regrouping, we find that $F S^T F^{-1} = F S^T F^{-1} F S^T F^{-1} = (F S^T F^{-1})^T$, so we can rewrite our equation as

$$a_T = F^{-1} (F S^T F^{-1})^T F a_0.$$

This form of the final data equation points to a remarkably efficient way of computing $a_T$. We first apply the convolution theorem [28], which states that if $S$ is a circular matrix, then $F S^T F^{-1} = \Lambda$ is diagonal. The equation for final data can now be regrouped with $F S^T F^{-1} = \Lambda$ and $F a_0 = x$:

$$a_T = F^{-1} \Lambda^T x.$$  

(1)

This equation may appear to be more complicated than what we started with, but really all we have done is made a change of basis into the frequency domain and performed all actions of the stencil there. This will now be shown to be computable with only a couple calls to highly efficient FFTs and some repeated squarings of scalars.

4.1 1-D Explicit Stencil Algorithm

Let $a_0[0, \ldots, N-1]$ be the initial 1-D spatial grid data to be acted on with the stencil $S = \sum_{j=1}^T S[0,0]X^j$. We will impose periodic boundary conditions; these allow us to benefit from pushing the stencil and initial data into the frequency domain with $F S F^{-1} = \Lambda$ and $F a_0 = x$. Our goal is to compute the final data

$$a_T = S^T a_0 = F^{-1} \Lambda^T x.$$  

The prevalent approach to finding $a_T$ is currently through iterative applications of the stencil to field data, grouping $S^T a_0$ into $S(S(\cdots S(a_0) \cdots))$ and evaluating according to parentheses. Here we will instead compute a power of the diagonalized stencil $(F S F^{-1})^T = \Lambda^T$ by repeated squaring, after which we will apply it to $F a_0$, giving us $F S^T a_0$, from which we can recover $S^T a_0$ with an inverse FFT.

It is shown in Appendix A.8 that we can write the elements of the diagonal matrix $\Lambda$ as $\Lambda[i, i] = (F S)[i, i]$, where $s$ is the column of $S$ that we are storing in memory, i.e. $S[i] = S[0, i]$. Since $F$ is the DFT matrix, this means that $\Lambda$ can be computed with a single FFT.

Blindly using repeated squaring only allows us to compute $\Lambda^T$ when $T$ is an exact power of 2; arbitrary positive integer powers are computed as follows. Let $\sum_j b_j 2^j$ be the binary representation of $T$.

As we compute successive squares of $\Lambda$, i.e. $\Lambda^{2^j}$, we will multiply them into a running total only if $b_j = 1$. Thus the final result will be

$$\Lambda^T = \prod_{i: b_i = 1} \Lambda^{2^i}.$$  

Since $\Lambda$ is diagonal, elements of the large matrix power $\Lambda^T$ can be computed by taking powers of the original elements, $\Lambda^T[i, i] = \Lambda[i, i]^T$. Evaluating each of these elements in parallel will improve the span of our algorithm.

Wrapping up, we evaluate Equation 1 as follows: we find $x = F a_0$ by applying FFT to the initial data, $\Lambda^T x$ by elementwise multiplication, and then $a_T$ with an inverse FFT.

We now present the PStencil-1D-FFT algorithm, which efficiently performs stencil computations with periodic boundary conditions by transferring almost all relevant calculations to within the frequency domain. A diagrammatic outline is shown in Figure 4, and the pseudocode is given in Figure 5.

[Step 1. FFT.] We compute $(i) F S F^{-1}$ from $S$, and $(ii)$ $F a_0$ from $a_0$. Since $S$ is circular, we know that the FFT of $S$'s first column contains exactly the same information as $F S F^{-1}$. Thus for $(i)$ an FFT is applied to the first column of $S$ to get $S$'s eigenvalues; this FFT will be computed for $N$ points, irrespective of how many nonzero coefficients are present in the stencil. Note that only the first column of $S$ is needed here, which is why the rest of $S$ is never constructed or stored in memory. Likewise, for $(ii)$ an FFT is applied to $a_0$.

[Step 2. Repeated Squaring.] We compute $F S^T F^{-1}$ from $F S F^{-1}$. Since $F S F^{-1}$ is diagonal, the individual elements of $F S^T F^{-1} = (F S F^{-1})^T$ can be computed in parallel by performing $[\log T]$ sequential squarings for each element along the principal diagonal of $F S F^{-1}$ according to the decomposition of $\Lambda^T$ given earlier.

[Step 3. Elementwise Product.] We compute $F a_T$ by taking the product of $F S^T F^{-1}$ and $F a_0$. As in step 2, every element of $F a_T = (F S^T F^{-1})(F a_0)$ can be computed in parallel, since we are multiplying a vector by a diagonal matrix.

[Step 4. Inverse FFT.] We now compute $a_T$ by applying an inverse FFT to $F a_T$.

Theorem 4.1. Stencil-FFT-P computes the $T$th timestep of a stencil computation on a periodic grid of $N$ cells in $\Theta(N \log(N T))$ work and $\Theta(\log T + \log N \log \log N)$ span.

Proof. Theorem follows from bounds given in Table 3. □

4.2 Generalizations

We now give overviews of some straightforward generalizations of the 1-D algorithm given above. These greatly enhance the scope of stencil problems our algorithm can handle, yet do not require significant conceptual work beyond what we have already presented. There are no changes in the asymptotic complexities of our algorithm when any of these generalizations are applied.

T Multi-Dimensional Stencils. The first extension of the 1-D version of our algorithm is to support arbitrarily high dimensional grids. The notions involved with solving across a grid of size $l_1 \times \cdots \times l_d = N$ differ from a 1-D grid only in that we now require indices for every grid dimension; details on the math can be found
in Appendix A.1. For our complexity analysis we assume that \( d = \Theta(1) \).

**Implicit Stencils.** The second way we expand the set of problems our algorithm can handle is by giving a method for handling implicit stencils [80]. This is accomplished by mapping them to mathematically equivalent explicit stencils, as explained in Appendix A.4.

Supporting implicit stencils is a significant capability in a stencil solver, as implicit stencils can often be designed to be more stable than explicit ones.

**Vector-Valued Fields.** The third way we extend our algorithms is to handle vector-valued fields, i.e. fields where the grid data for each cell is an array of fixed length instead of being just a scalar. We allow a set of scalar-valued fields \( \{a_t^{(i)}\} \) to evolve according to linear stencils as

\[
a_{t+1}^{(i)} = \sum_j S^{(i,j)} a_t^{(j)},
\]

where \( S^{(i,j)} \) is a circulant stencil matrix describing how \( a_{t+1}^{(i)} \) is dependent on \( a_t^{(j)} \). The details are given in Appendix A.5. It is noteworthy that if there are a constant number of scalar fields \( a_t^{(i)} \), making up our grid data, then the complexity of finding the final data is only a constant factor greater than if the field were scalar.

Vector-valued grid data allows us to support much larger classes of stencils, such those that are affine or require data from multiple prior steps.

5 **APERIODIC STENCIL ALGORITHM**

In this section we will first consider how introducing aperiodicity into our boundary conditions leads to changes in the final data, then present an algorithm for computing the values of the specific cells that are affected by the aperiodic boundary conditions.

5.1 **The Effect of Aperiodicity**

Often numerical computations require boundary conditions that are not periodic [15, 20, 73, 98, 129], including well-known classes such as Dirichlet [145] and Von Neumann [90], as well as more exotic options [125]. Indeed, the set of potential aperiodic boundary conditions is extremely diverse. Here we will not attempt to describe how given aperiodic boundary conditions change the final data, but rather what sections of the final data are changed.

The set of cells that are dependent on values from the grid boundary are called the boundary’s region of influence (ROI) [13, 35, 96, 139]. It is common for the boundary’s region of influence to be smaller than the entire spatial grid, as would be the case in a simulation of a large spatial region for a small time period. A cell’s influence on its neighbors is mediated by the stencil, so the larger the stencil is the fewer timesteps it takes for one cell to influence the whole grid.

Consider a stencil with radius \( \sigma \), i.e. one which only uses values from a neighborhood extending up to \( \sigma \) cells away from its center. After one timestep of a computation based on this stencil, the set of cells which are influenced by aperiodic boundary conditions will be exactly those that are at distance \( \sigma \) or less from the grid boundary. After \( T \) timesteps, all cells within distance \( \sigma T \) of the boundary will be influenced. This suggests that we should visualize how the boundary’s region of influence grows by drawing a diagram in two variables (as in Figure 6), one being time and the other distance from the boundary. Figures of this type will be of significant use to us while describing our aperiodic algorithm.

We now move to our aperiodic algorithm **StencilFFTA**, at the core of which is **RECURSIVEBOUNDARY**, a divide and conquer technique for correcting the values of the cells in the boundary’s region of influence.
5.2 Correcting Final Values in the Boundary’s Region of Influence

Our aperiodic algorithm breaks up the final data into two regions which will be computed with different methods. The interior region consists of cells whose values will not change no matter what boundary conditions we apply. In particular, periodic boundary conditions would not change the value of these cells. This region will thus be dealt with by using our efficient periodic algorithm.

The exterior region is the boundary’s region of influence, made up of cells that are dependent on boundary values.

Since we already know how to solve for the interior region, let us turn our attention to the boundary’s region of influence, which we solve for with a recursive divide and conquer approach. The core idea here will be to perform a time-cut to reduce the number of cells affected by the aperiodic boundary; by splitting one large step into two smaller steps we will be able to use the periodic solver on more space, at the cost of having to compute cell values at an intermediate time.

Figure 7 diagrammatically shows how RecursiveBoundary solves for the boundary region. First all values in region A are used to solve for region B, then region B is used to solve for region C. These regions can be much smaller than the entire spatial grid, since even though they wrap around the entirety of the boundary, they are only a fixed number of cells thick.

Suppose again that we are performing computations with a stencil of radius \( \sigma \). Then set \( A \) is the boundary’s region of influence after \( T \) timesteps, which includes all cells within distance \( \sigma T \) of the boundary. Since we want to be able to fully compute C from B and B from A, we must continue to extend our regions as we go \( T/2 \) steps back for each; B includes everything within \( 3\sigma T/2 \) cells of the boundary, and A includes everything out to a distance of \( 2\sigma T \).

We now refine our naming of these regions, defining subregions \( A_1, A_2, B_1, B_2, B_3, C_1 \), and \( C_2 \), whose sizes and locations are shown in Figure 8.

The language we have used up to this point has been dimension-free, in that every statement made has been independent of the number of dimensions in the spatial grid we are performing computations on. Now, to make clear the regions shown in Figures 7 and 8, we will consider what exactly they look like when we make a choice of spatial grid dimension.

In 1-D, when the spatial grid \( a[0,\ldots,N-1] \) is just a linear array of \( N \) cells, the boundary lies to the left of \( a[0] \) and to the right of \( a[N-1] \). The set of all cells within distance \( d \) of the 1-D grid’s boundary is given by the union of \( a[0,\ldots,d-1] \) and \( a[N-d,\ldots,N-1] \). For example, the region \( C \) is the union of 1-D grid's boundary:

\[
\begin{align*}
A_1 &= [0,\ldots,(n_1-1)], \quad A_2 = [0,\ldots,(n_2-1)], \\
B_1 &= a[0,\ldots,(n_1-1),0,\ldots,(n_2-1)], \quad B_2 = a[0,\ldots,(n_1-1),0,\ldots,(n_2-1)], \quad B_3 = a[0,\ldots,(n_1-1),0,\ldots,(n_2-1)].
\end{align*}
\]

In 2-D things become more complicated, since the boundary of the \( n_1\times n_2 \) spatial grid \( a[0,\ldots,(n_2-1),\ldots,(n_1-1)] \) lies on the outside of the connected set of cells made up of the 1-D subarrays \( a[0,\ldots,(n_2-1)], a[n_1-1,\ldots,(n_2-1)], a[0,\ldots,(n_1-1),0], \) and \( a[0,\ldots,(n_1-1),n_2-1] \). The set of all cells within distance \( d \) of the 2-D grid’s boundary is the union of 2-D grid's boundary:

\[
\begin{align*}
A_1 &= [0,\ldots,(d-1),\ldots,(n_2-1)], \quad A_2 = [0,\ldots,(d-1),\ldots,(n_2-1)], \\
B_1 &= a[0,\ldots,\ldots,(d-1)], \quad B_2 = a[0,\ldots,\ldots,(d-1)].
\end{align*}
\]

Note that we are listing some cells twice here, specifically those in the corners of the grid.

Now we are prepared to describe in detail\(^1\) the RecursiveBoundary algorithm for computing the correct values of cells in the boundary’s region of influence. Consider again Figure 8. The RecursiveBoundary algorithm solves for two time-slices which are done sequentially. Each time-slice is divided into two distinct regions: one which is solved for with a periodic solver, and one which is solved for recursively. These two regions are handled in parallel.

---

\(^1\)For brevity here we use the region names as shown in Figure 8. The reader who would prefer to see these regions specified directly in terms of their distance from the boundary is referred to Appendix A.7.
(1) **Solving for B from A.** We will begin by feeding data from region \( A = A_1 \cup A_2 \) into our periodic solver to find values for region \( B_2 \cup B_3 \). For a 2-D grid this will take 2d calls to the periodic solver. At the same time (in parallel), a recursive call is made to the RecursiveBoundary algorithm with \( A_1 \) as input, writing the output to \( B_1 \). This completes all the values of \( B \).

(2) **Solving for C from B.** Now \( B = B_1 \cup B_2 \cup B_3 \) is fed into our periodic algorithm to find \( C_2 \), while in parallel we feed \( B_1 \cup B_2 \) into another recursive call to RecursiveBoundary, which will find the values for \( C_1 \). Thus all cell values in \( C = C_1 \cup C_2 \) are computed.

The base cases in recursion occur when the only cells to be computed are within a constant distance of the boundary. In this case a standard looping algorithm is applied. We theoretically can set the cutoff distance to any constant greater than or equal to \( \sigma \), but in practice the constant is chosen such that the cost of computing the base case is balanced with the cost of further recursion. See Figure 9 for pseudocode.

Combining our periodic solver for the interior region with this recursive solver for the boundary region gives our aperiodic algorithm StencilFFT-A. A diagrammatic outline of StencilFFT-A is given in Figure 9, and pseudocode is given in Figure 9.

**Theorem 5.1.** The StencilFFT-A algorithm can compute the \( T \)th timestep of a stencil computation on a grid of size \( N \) with aperiodic boundary conditions in \( \Theta \left( k N \log b T \log N + N \log N \right) \) work and \( \Theta \left( T \log b + \log N \log N \right) \) span, where \( b \) is the number of grid cells defined by boundary conditions.

The proof is presented in Appendix A.3.

We believe that the bounds we achieve here for work and span are near-optimal (within a polylogarithmic factor) for fully general aperiodic stencil problems. This is because specific choices of nonlinear boundary conditions in combination with linear stencils can result in arbitrary cellular automata being embedded into the boundary of the spatial grid. These automata can be computationally universal (any computation can be mapped to them in a way that preserves time and space complexity), hence the \( \Theta \left( T N^{1-1/d} \right) \) (size of the space-time boundary) component in our work complexity and the \( \Theta \left( T \right) \) component in our span complexity.

## 6 EXPERIMENTS

In this section, we present the experimental evaluation of our algorithms as compared with the state-of-the-art stencil codes. Our experimental setup is shown in Table 4.

| KNL | Cores | Cache sizes | Memory |
|-----|-------|-------------|--------|
|      | 68 cores per socket, 1 socket (total: 68 threads) | L1 32 KB, L2 1 MB, L3 16 GB (shared) | 96 GB DDR RAM |

| SKX | Cores | Cache sizes | Memory |
|-----|-------|-------------|--------|
|      | 24 cores per socket, 2 sockets (total: 48 cores) | L1 32 KB, L2 1 MB, L3 33 MB | 144 GB/tmp partition on a 200GB SSD |

| Compiler | Compiler flags | Parallelization | Thread affinity |
|----------|----------------|----------------|-----------------|
| Intel C++ Compiler (ICC) v18.0.2 | -03 -xhost -ansi-alias -ipo -AVX512 OpenMP 5.0 | GOMP_CPU_AFFINTY |

**StencilFFT-A**

1. \( \Delta \leftarrow \sigma T/2 \)
2. result \( \leftarrow \text{Array of size } t_1 \times \ldots \times t_d \)
3. \( \text{parallel (1):} \)
4. center \( \leftarrow \text{StencilFFT-P}(s, a_0, t_1, \ldots, t_d, T) \)
5. result[2A < dist] \( \leftarrow \text{center}[2A < dist] \)
6. \( \text{parallel (2):} \)
7. boundary \( \leftarrow \text{RecursiveBoundary}(s, a_0, t_1, \ldots, t_d, T) \)
8. result[0 < dist < 2A] \( \leftarrow \text{boundary} 

**RecursiveBoundary**

1. \( \Delta \leftarrow \sigma T/2 \)
2. if \( T < \text{cutoff} \) then
3. iteratively solve for cells within 2A of the boundary at time \( T \)
4. return

(Step 1)

5. \( \text{parallel (1):} \)
6. \( A_1 \leftarrow a_0[0 < \text{dist} < 2A] \)
7. \( B_{21} \leftarrow \text{StencilFFT-P}(s, a_1, t_1, \ldots, t_d, T) \)
8. result[0 < \text{dist} < 2A] \( \leftarrow B_{21}[0 < \text{dist} < 2A] \)

(Step 2)

10. \( A_1 \leftarrow a_0[0 < \text{dist} < 2A] \)
11. \( B_1 \leftarrow \text{RecursiveBoundary}(s, A_1, t_1, \ldots, t_d, T) \)
12. result[0 < \text{dist} < \Delta] \( \leftarrow B_1 \)
13. \( B_{21} \leftarrow \text{result}[0 < \text{dist} < 2A] \)

**Figure 9:** The StencilFFT-A algorithm and the RecursiveBoundary subroutine. The parallel keyword is used here to mark blocks of code which are run on separate processes. Throughout both listings we use a dimension-free indexing notation where \( A[a < \text{dist} \leq b] \) represents the set of cells in \( A \) that have distance to the boundary in the range \((a,b)\).

**Benchmarks and Numerical Accuracy.** For benchmarks we use a variety of stencil problems including those with periodic and aperiodic boundary conditions and across 1, 2, and 3 dimensions. Our test stencils, primarily drawn from [112] and listed with details in Table 5, are the following: heat1d, heat2d, sei1d2d, jacobi2d, heat3d, and 19pt3d (called poisson3d in [112]). We test two primary aspects of our algorithms: numerical accuracy and computational complexity.

To evaluate numerical accuracy we use max relative error against analytical solutions for the heat equation in 1, 2, and 3 dimensions. This is shown in Table 6 against a naïve iterative looping implementation which is numerically equivalent to PLuTo. We see that our algorithms show no significant difference in loss from floating point accuracy when compared against standard looping codes.

| Dim | Benchmark | Stencil points (\( \sigma \)) | Stencil radius (\( \sigma \)) |
|-----|-----------|----------------------------|-----------------------------|
| 1D  | heat1d   | 3pts 1                     |
| 2D  | heat2d   | 5pts 1                     |
|     | sei1d2d  | 9pts 1                     |
|     | jacobi2d | 25pts 2                    |
| 3D  | heat3d   | 7pts 1                     |
|     | 19pt3d   | 19pts 2                    |

**Table 5:** Benchmark problems with the number of points in the corresponding stencils.
Table 6: Numerical accuracy comparison between our algorithms and looping code. Our analytical solutions were chosen so that the truth values fell within [0.5, 2] everywhere in the solution domain.

Table 7: Performance summary of parallel stencil algorithms on a KNL/SKX node.

PluTo-Generated Stencil Programs. The tiled looping implementations were generated by PluTo [3] – the state-of-the-art tiled looping code generator. The two main types of tiling methods used for performance comparison are: standard and diamond, whose tiles have the shapes of parallelograms and diamonds, respectively. In the plots, we use diamond and square symbols to denote diamond and standard, respectively. These parallel implementations were run on 68-core KNL and 48-core SKX nodes. The tile sizes were selected via an autotuning phase, exploring sizes from \{8, 16, 32\} for the outer dimensions and from \{64, 128, 512\} for the inner-most dimension to ensure that enough vectorization and multithreaded parallelism were exposed by PluTo, while ensuring the tile footprint neared the cache size.

Our FFT-Based Stencil Programs. Implementations of our FFT-based algorithms use FFT implementations available in the Intel Math Kernel Library (Intel MKL) [1]. In the plots, we use the triangle symbol to represent our FFT-based implementations. The base case sizes used for 1d, 2d, 3d are 128, 64 × 64, 16 × 16 × 16, respectively.

6.1 Periodic Stencil Algorithms

Figure 11 shows runtime, speedup (w.r.t. PluTo), and scaling plots for our FFT-based periodic stencil algorithm on Intel KNL nodes with Figure 14 in the Appendix showing additional runtime plots. Figure 14 also includes the same set of plots for SKX nodes. We identify the implementations of our algorithms in the plots by prefixing the stencil name with "FFT" while PluTo-generated implementations are identified by a "PluTo" prefix.

For 1D, 2D, and 3D stencils we keep the value of \(N\) fixed to 1.6M, 8K × 8K, and 800 × 800 × 800, respectively, and vary \(T\), where \(1M = 10^6\) and \(1K = 10^3\).

In all of our 1D and 2D periodic stencil experiments diamond ran faster than standard while in case of 3D standard outperformed diamond. When \(N\) is fixed, performance of our algorithm improved over PluTo’s as \(T\) increased, significantly outperforming PluTo for large \(T\), e.g., for seidel2d our algorithm ran around 6000× faster on KNL when \(T = 10^6\). This increase in speedup with the increase of \(T\) follows from theoretical predictions. Indeed, theoretical speedup of our algorithm over any existing stencil algorithms is \(\Theta (T/\log T)\) when \(N\) is fixed (see Table 1). Hence, the speedup of our algorithm w.r.t. PluTo-generated codes increased almost linearly with \(T\) when \(N\) was kept fixed.

Figures 11(e) and 14(xi) show the scalability of our algorithm on KNL and SKX nodes, respectively, when the number of threads is varied. Our implementations are highly parallel and should scale accordingly. However, we use FFT computations that are memory bound – they perform only \(\Theta (N \log N)\) work on an input of size \(\Theta (N)\) and thus have very little data reuse. We believe that as a result of this issue, our programs do not scale well beyond 32 threads on KNL and 16 threads on SKX. Indeed, we observe that the FFT and the inverse FFT computations are the scalability bottlenecks of our algorithms.

6.2 Aperiodic Stencil Algorithms

We performed two types of experiments for aperiodic stencils: (1) grid size \(N\) was kept fixed while time \(T\) was varied, and (2) grid size was set to \(N^{1/d} \times \cdots \times N^{1/d}\) and \(T = N^{1/d}\) for \(d\) dimensions, and \(N\) was varied.

Figure 12 shows the runtime (w.r.t. PluTo) plots of our aperiodic stencil algorithm for both experiments on KNL. Figure 15 in the Appendix includes additional runtime plots for both experiments on KNL. Figures 16 and 17 in the Appendix show the corresponding plots on SKX for Experiments 1 and Experiment 2, respectively.

In Experiment 1, diamond outperformed standard for all stencils except for heat1d on both machines. Our algorithm always ran faster than PluTo-generated code, reaching speedup factors of 8.5, 2.3–2.4, and 1.3–1.5 for 1D, 2D, and 3D stencils, respectively, on KNL (see Table 7 for details). The corresponding figures on SKX were 5.2, 1.7–4.5, and 1.3–1.4, respectively. Theoretical bounds in Table 1 imply that our algorithm will run around \(\Theta \left(\frac{N^{1/d}/(\log N)}{N^{1/d}}\right)^{\log T}\) factor faster than PluTo code for any given \(N\) and \(T\). So, for a fixed \(N\), the speedup factor will not increase (may even slightly decrease) with the increase of \(T\). The speedup plots match this prediction.

In Experiment 2, diamond ran faster than standard for all stencils except for heat1d and heat3d on KNL and heat1d on SKX. Our algorithm ran up to 5.7, 2.3–2.5, and 1.4–1.8 factor faster than PluTo-generated code for 1D, 2D, and 3D stencils, respectively, on KNL (see Table 7). The corresponding speedup factors on SKX were 3.3, 1.4–2.6, and 1.4, respectively. Our theoretical prediction for rough speedup factor from the previous paragraph implies that our speedup over PluTo code will increase with the increase of \(N\) which is confirmed by the speedup plots for this experiment. However, the speedup plot of heat2d, seidel12d, and jacobi12d on KNL, as shown in Figure 12(xii) has speedup drops at \(N = 9000 \times 9000\) and \(N = 15000 \times 15000\). Similar performance drops are also observed on SKX nodes (see Figure 17(xiii) in the Appendix). We believe that this happens mainly because of a known phenomenon which is the
drastic performance variations MKL suffers from when the sizes of
the spatial grid dimensions change [78]. This performance drop is
also partly due to the changes in the base case kernel size of our
implementations resulting from the changes in the grid size.

Figure 10 and Appendix Figure 13 show the scalability plots of
our FFT-based aperiodic stencil algorithm on KNL and SKX nodes,
respectively. We used $N = 1M$, $N = 16K \times 16K$, and $N = 800 \times 800 \times$
800 for 1-D, 2-D, and 3-D stencils, respectively, and set $T = N^{1/d}$
for our scalability analysis, where $d$ is the number of dimensions.
Our implementations show highly scalable performance on KNL
and almost similar scalability for 1D and 2D stencils on SKX.

![Figure 10: Scalability of our aperiodic algorithms for Exper-
iment 2 on a KNL node.](image)

7 CONCLUSION

In this paper, we presented a pair of efficient algorithms based
on fast Fourier transforms for performing linear stencil com-
putations with periodic and aperiodic boundary conditions. These
are the first high-performing $o(NT)$-work\textsuperscript{9} stencil algorithms of
significance generality for computing the spatial grid values at the
final timestep from the input grid without explicitly generating
values for most of the intermediate timesteps. Our stencil algo-
rithms improve computational complexity and parallel running
time bounds over the state-of-the-art stencil algorithms by a polyno-
mial factor. Experimental results show that implementations of
our algorithms run orders of magnitude faster than state-of-the-art
implementations for periodic stencils and 1.3× to 8.5× faster for
aperiodic stencils, while exhibiting no significant loss in numerical
accuracy from floating point arithmetic.

A few interesting problems that one could aim to solve in the fu-
ture include: (1) designing efficient algorithms for certain classes of
nonlinear stencils and stencils with conditionals, (2) designing low-
span algorithms for aperiodic stencils, and (3) designing algorithms
to approximate inhomogenous stencils.

\textsuperscript{9}$N$ and $T$ are the spatial grid size and the number of timesteps, respectively.
Figure 11: Performance comparison of our FFT-based periodic algorithms with the existing best stencil programs.
Figure 12: Performance comparison of our FFT-based aperiodic algorithms with the existing best stencil programs.
Figure 13: Scalability of our aperiodic algorithms for Experiment 2 on an SKX node.

A APPENDIX
A.1 Supporting d-D Spatial Grids
Throughout this section we will use arguments identical to those given for the 1-D version of StencilFFT-P, but with the modification that whereas the stencil used and grid data used to be indexed by an integer, they will now be indexed by a d-D vector \( i = [i_1, \ldots, i_d] \). Let \( \mathcal{S}[i] \) be the initial data for a grid of size \( f_1 \times \cdots \times f_d = N \) to be acted on by stencil \( \mathcal{S} = \sum_i \mathcal{S}[i] X^i \), where \( X^i \) is a shift operator whose action is given by \( (X^k a)(i) = a[i-k] \) and whose matrix elements are \( X^k [i, j] = \delta_{i+k,j} \). We have periodic boundary conditions, so \( X_j \) can be viewed as cyclically permuting the \( j \)th dimension. Future times are defined with \( a_{t+1} = S a_t = S^{t+1} a_0 \), where matrix multiplications are contractions over the internal vector indices, \( (Sa)[i] = \sum_j S[i,j] a[j] \).

As with the 1-D case, \( S \) is a circulant matrix, so we can apply DFTs over every spatial dimension to diagonalize it. These DFT matrices together make up an operator with elements given by \( \mathcal{F}[i, j] = \omega_{i_1} \omega_{i_2} \cdots \omega_{i_d} \) and \( \mathcal{F}^{-1}[i, j] = \frac{1}{N} \mathcal{F}[i, j] \). Note of course that “diagonal” here mean that \( \Lambda[i,j] = (\mathcal{F}\mathcal{F}^{-1})[i,j] = 0 \) for all \( i \neq j \). The proof that \( \Lambda \) can be found by applying a single Multi-FFT (see Figure 5) over the first column of \( S \) as given in Appendix A.8. Here we only restate the result that \( \Lambda[i,j] = (\mathcal{F}s)[i,j] \), where \( s[i] = S[i,0] \).

Switching from 1-D to d-D results in the version of the StencilFFT-P algorithm whose pseudocode is given in Figure 5 in the main body of this paper. Since Multi-FFT is of equivalent work, span, and serial cache complexity to a standard FFT [47], this version of the StencilFFT-P algorithm is of identical complexity to that already shown in Theorem 4.1.

A.2 Stencil Matrix Example
Suppose we have a 1-D periodic spatial grid of 4 cells, and we want to act on it with a stencil \( S \) that computes the new value of a cell according to \( a_{t+1}[i] = -2a_t[i-1] + a_t[i] + 3a_t[i+1] \). We can write out the update equation in full as

\[
\begin{bmatrix}
a_{t+1}[0] \\
a_{t+1}[1] \\
a_{t+1}[2] \\
a_{t+1}[3]
\end{bmatrix}
= \begin{bmatrix}
1 & 3 & 0 & -2 \\
-2 & 1 & 3 & 0 \\
0 & -2 & 1 & 3 \\
3 & 0 & -2 & 1
\end{bmatrix}
\begin{bmatrix}
a_t[0] \\
a_t[1] \\
a_t[2] \\
a_t[3]
\end{bmatrix}.
\]

\( a_{t+1} \)

A.3 Proof of Theorem 5.1
Proof. The complexities for StencilFFT-P have already been given in Theorem 4.1, so here we need only derive the complexities for the RecursiveBoundary computation algorithm.

Let the spatial grid be of size \( f_1 \times \cdots \times f_d = N \), with \( d = \Theta(1) \). We bound the number of cells in the boundary’s region of interest by \( b \sigma T \), where \( b = 2N(t_1^{-1} + \cdots + t_d^{-1}) = \Omega(N^{1-1/d}) \) is the size of the spatial grid’s boundary and \( \sigma = \Theta(1) \) is the size of the stencil being applied.

[Work.] At every stage of the divide and conquer algorithm we make two recursive calls and apply the periodic solver to \( (bT) \) grid cells. This gives us the recurrence

\[
T_1(T) = \begin{cases} 
\Theta(b), & T < c, \\
2T_1(T/2) + \Theta(bT \log(bT)), & T \geq c,
\end{cases}
\]

for some positive constant \( c \). This gives \( T_1(T) = \Theta(bT \log(bT) \log T) \). Taking into account the \( \Theta(N \log(TN)) \) work that will be done by the periodic solver for the interior region gives a final work bound of \( \Theta(bT \log(bT) \log T + N \log N) \).

[Spawn.] At every level of recursion, both calls to RecursiveBoundary are done in sequence, but in parallel with the \( \Theta(1) \) associated periodic solver calls. The recurrence for span is thus

\[
T_{\text{ov}}(T) = \max\{2T_{\text{ov}}(T/2), c' \log T \log \log T\} + \Theta(\log b),
\]

where \( c' = \Theta(1) \). This gives \( T_{\text{ov}}(T) = \Theta(T \log b) \), and incorporating the \( \Theta(T \log T + \log N \log \log N) \) span of the periodic solver across the interior region gives a final bound of \( \Theta(T \log b + \log N \log \log N) \).

A.4 Supporting Implicit Stencils
Here we show how to extend StencilFFT-P to handle stencils which depend implicitly on field data from the timestep which is being computed. This will be done by computing a pseudoinverse after diagonalizing the implicit part of the stencil.

Let \( S \) and \( Q \) be explicit stencils, and suppose we wish to solve a stencil problem with update equation \( Qa_{t+1} = Sa_t \). First we diagonalize \( \mathcal{Q}\mathcal{F}\mathcal{F}^{-1} = \mathcal{Q} \) and \( \mathcal{F}\mathcal{S}\mathcal{F}^{-1} = \mathcal{S} \) by pushing them to the frequency domain, after which we will take the pseudoinverse of \( \mathcal{Q} \), defined as \( \Lambda^\dagger_Q[i,j] = \mathcal{Q}^{-1}[i,j] \) if \( \Lambda_Q[i,j] \neq 0 \), and 0 otherwise. Now we find a new diagonalized stencil \( \Lambda_R = \Lambda^\dagger_Q \Lambda_S \). This new stencil \( R \) has the following property: if \( a_{t+1} = Ra_t \), then \( Qa_{t+1} = Sa_t \).

Thus we can evolve data according to \( R \), and the evolved data will satisfy the original implicit stencil equation.

A.5 Supporting Vector Valued Fields
Here we show how to alter our periodic algorithm to handle vector-valued fields, i.e. fields where the grid data for each cell is an array of fixed length instead of being just a scalar. Consider a set of scalar-valued fields \( a_{t+1}^{(i)} \) which evolve according to linear stencils as

\[
a^{(i)}_{t+1} = \sum_j S^{(i,j)} a_j^{(i)},
\]

where \( S^{(i,j)} \) is the \( j \)th column of the stencil matrix. To evolve a vector field, we simply apply the stencil to each component separately, just as we did for the scalar case.
where \( S^{(i,j)} \) is a circulant stencil matrix describing how \( a_{t+1}^{(i)} \) is dependent on \( a_t^{(j)} \). As in the scalar-field case, we can diagonalize all the stencils \( S^{(i,j)} \) by moving them to the frequency domain.

Since we now have more than one stencil, we have to revisit our treatment of sequential squaring. Let \( \{ R^{(i,j)} \} \) be a set of stencils which evolves \( a_t \) forward \( r \) timesteps to \( a_{t+r} \).

\[
a_{t+r}^{(i)} = \sum_j R^{(i,j)} a_t^{(j)}.
\]

We can find a new set of stencils \( \{ Q^{(i,j)} \} \) which evolve data forward \( 2r \) timesteps by reading them off of

\[
\sum_j Q^{(i,k)} a_t^{(k)} = \sum_j R^{(i,j)} a_{t+2r}^{(j)} \Rightarrow \sum_j R^{(i,j)} \sum_k Q^{(i,k)} a_t^{(k)} = \sum_k R^{(i,j)} a_{t+2r}^{(j)}.
\]

This gives the pleasing result that

\[
Q^{(i,k)} = \sum_j R^{(i,j)} P^{(j,k)},
\]
so all we have to do to support vector-valued fields is to swap out our sequential squaring of $\Lambda$ with a sequential squaring of a matrix of stencils.

Furthermore, when there are only a constant number of scalar fields $\{a_t^{(i)}\}$ that make up our grid data, the computational complexity of finding $\{Q^{(i)}\}$ differs only by a constant factor from squaring $\Lambda$ in our scalar-field version of the algorithm.

As an example of what can be done with vector-valued fields, suppose we want to implement an affine stencil $A_{t_{ij}} = S_{t_{ij}} + c$ on some originally scalar field, where $S$ is a linear stencil and $c$ is a constant. We could then add a spatial field (making the underlying data consist of vectors with 2 elements each) and define $a_t^{(0)} = c$, $S^{(1,1)} = S^{(0,1)} = I$, $S^{(1,0)} = 0$, and $S^{(0,0)} = A$. This realized the behavior of the affine stencil on $a_t^{(0)}$.

### A.6 Proof of Shift Matrix Decomposition of Circulant Matrices

Let $S$ be a circulant matrix, and let $X$ be the right shift matrix. Then for any vector $a$, we have

$$\begin{align*}
(Sa)[i] &= \sum_j S[i, j]a[j] = \sum_j S[i-j, 0]a[j] \\
&= \sum_j S[j, 0]a[i-j] = \sum_j S[j, 0](X^j a)[i] \\
&= \left(\sum_j S[j, 0]X^j\right)a[i],
\end{align*}$$

which shows that $S = \sum_j S[i, 0]X^i$.

### A.7 $\ell$-shell Spatial Grid Decomposition

In the main body of this paper we assume that stencils with some radius $\sigma$ use the values of cells at distance $\sigma$ in any particular direction. However, this is not necessarily the case; one could imagine a stencil that requires a large number of values along one dimension and only a few along another. Upwind stencils also break this pattern, requiring values from very asymmetric neighborhoods of cells.

The concept of region of influence used in our derivation of RecursiveBoundary is itself a sufficient basis for defining the regions $A_{1,2}, B_{1,2,3}$, and $C_{1,2}$ shown in Figure 8. Let us define an $\ell$-shell of the boundary of a spatial grid to be the spatial region consisting of all cells that enter the boundary’s region of influence after exactly $\ell$ timesteps. Obviously, there can be only one time $\ell$ when a cell begins to be influenced by the boundary; the set of all $\ell$-shells thus fill the spatial grid without overlapping one another.

To generalize the regions shown in Figure 8 to those for arbitrary stencils, all we have to do is switch the interpretation of the horizontal axis from “distance” to “$\ell$” and scale it by setting $\sigma = 1$. This yields an algorithm that is more efficient for upwind schemes and other biased stencils.

### A.8 Proof of Eigenvalue-FFT Relation

Expanding the diagonal matrix of eigenvalues $\mathcal{F}A\mathcal{F}^{-1} = \Lambda$ directly gives us
\[
\Lambda[i, i] = (\mathcal{F}A\mathcal{F}^{-1})[i, i] = \left(\mathcal{F} \left( \sum_j A[j, 0] X^j \right) \mathcal{F}^{-1} \right)[i, i] \\
= \sum_j A[j, 0] (\mathcal{F}X^j\mathcal{F}^{-1})[i, i] \\
= \sum_j A[j, 0] \sum_k \omega_N^{-i(k+j)} \delta_{k, k+j}/N \\
= \sum_j A[j, 0] \sum_k \omega_N^{-i(k+j)} \omega_N^{k_j}/N = \sum_j \omega_N^{-i} A[j, 0] \\
= (\mathcal{F}A)[i, 0] = (\mathcal{F}a)[i],
\]

where in the last line \(a\) is the first column of \(A\).
Figure 17: Performance comparison of our FFT-based aperiodic algorithms with the existing best stencil programs.
REFERENCES

[1] Intel math kernel library. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html, Intel MKL.

[2] Lym framework for high-level loop and data-locality optimizations. https://ollylvm.org/. LYM MKL.

[3] An automatic parallelizer and locality optimizer for affine loop nests. http://pluto-compiler.sourceforge.net/, Pluto.

[4] The polyhedral compiler collection. http://web.cs.uc.edu/~pouc/software/pcc/, PCC.

[5] The stampled2 supercomputing cluster. https://www.tacc.utexas.edu/systems/stampled2, Stamped2.

[6] A. Atangana and J. J. Nieto. Numerical solution for the model of rlc circuit. Advances in Mechanical Engineering, 10(3):1687140115613758, 2015.

[7] M. Benzi, M. Ng, Q. Niu, and Z. Wang. A relaxed dimensional factorization method for heterogeneous linear systems with immersed boundaries. Journal of computational physics, 331:227–256, 2017.

[8] A. Ashrafizadeh, C. Devaud, and N. Aydemir. A Jacobian-free newton–krylov method for thermalhydraulics simulations. International Journal for Numerical Methods in Fluids, 77(10):590–615, 2015.

[9] M. A. Bender, E. D. Demaine, B. Ebrahimis, T. J. Fineman, R. Johnson, A. Lincoln, J. Lynch, and S. McCauley. Cache-adaptive algorithms. In ACM Symposium on Parallelism in Algorithms and Architectures, pages 135–144, 2016.

[10] A. Atangana and J. J. Nieto. Numerical solution for the model of rlc circuit via the fractional derivative without singular kernel. Advances in Mechanical Engineering, 10(1):1687140115613758, 2015.

[11] B. V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–11, 2012.

[12] J. Blazek. Advances in Mechanical Engineering, 10(3):1687140115613758, 2015.

[13] M. A. Bender, E. D. Demaine, B. Ebrahimis, T. J. Fineman, G. Gashperstein, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In ACM-SIAM Symposium on Discrete Algorithms, 2014.

[14] M. Benzi and G. H. Golub. A preconditioner for generalized saddle point problems. SIAM Journal on Matrix Analysis and Applications, 26(1):20–41, 2004.

[15] M. Benzi, M. Ng, Q. Niu, and Z. Wang. A relaxed dimensional factorization preconditioner for the incompressible navier–stokes equations. Journal of Computational Physics, 236(16):6815–6822, 2017.

[16] G. H. Golub, J. M. Ortega, et al. Scientific computing. volume 31999. McGraw-Hill New York, 1986.

[17] M. A. Bender, E. D. Demaine, B. Ebrahimis, T. J. Fineman, G. Gashperstein, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In ACM-SIAM Symposium on Discrete Algorithms, 2014.

[18] M. Benzi and G. H. Golub. A preconditioner for generalized saddle point problems. SIAM Journal on Matrix Analysis and Applications, 26(1):20–41, 2004.

[19] M. Benzi, M. Ng, Q. Niu, and Z. Wang. A relaxed dimensional factorization preconditioner for the incompressible navier–stokes equations. Journal of Computational Physics, 236(16):6815–6822, 2017.

[20] G. H. Golub, J. M. Ortega, et al. Scientific computing. volume 31999. McGraw-Hill New York, 1986.

[21] M. A. Bender, E. D. Demaine, B. Ebrahimis, T. J. Fineman, G. Gashperstein, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In ACM-SIAM Symposium on Discrete Algorithms, 2014.

[22] M. Benzi and G. H. Golub. A preconditioner for generalized saddle point problems. SIAM Journal on Matrix Analysis and Applications, 26(1):20–41, 2004.

[23] M. Benzi, M. Ng, Q. Niu, and Z. Wang. A relaxed dimensional factorization preconditioner for the incompressible navier–stokes equations. Journal of Computational Physics, 236(16):6815–6822, 2017.

[24] G. H. Golub, J. M. Ortega, et al. Scientific computing. volume 31999. McGraw-Hill New York, 1986.

[25] M. A. Bender, E. D. Demaine, B. Ebrahimis, T. J. Fineman, G. Gashperstein, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In ACM-SIAM Symposium on Discrete Algorithms, 2014.

[26] M. Benzi and G. H. Golub. A preconditioner for generalized saddle point problems. SIAM Journal on Matrix Analysis and Applications, 26(1):20–41, 2004.

[27] M. Benzi, M. Ng, Q. Niu, and Z. Wang. A relaxed dimensional factorization preconditioner for the incompressible navier–stokes equations. Journal of Computational Physics, 236(16):6815–6822, 2017.

[28] G. H. Golub, J. M. Ortega, et al. Scientific computing. volume 31999. McGraw-Hill New York, 1986.

[29] M. A. Bender, E. D. Demaine, B. Ebrahimis, T. J. Fineman, G. Gashperstein, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In ACM-SIAM Symposium on Discrete Algorithms, 2014.

[30] M. Benzi and G. H. Golub. A preconditioner for generalized saddle point problems. SIAM Journal on Matrix Analysis and Applications, 26(1):20–41, 2004.

[31] M. Benzi, M. Ng, Q. Niu, and Z. Wang. A relaxed dimensional factorization preconditioner for the incompressible navier–stokes equations. Journal of Computational Physics, 236(16):6815–6822, 2017.

[32] G. H. Golub, J. M. Ortega, et al. Scientific computing. volume 31999. McGraw-Hill New York, 1986.

[33] M. A. Bender, E. D. Demaine, B. Ebrahimis, T. J. Fineman, G. Gashperstein, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In ACM-SIAM Symposium on Discrete Algorithms, 2014.

[34] M. Benzi and G. H. Golub. A preconditioner for generalized saddle point problems. SIAM Journal on Matrix Analysis and Applications, 26(1):20–41, 2004.

[35] M. Benzi, M. Ng, Q. Niu, and Z. Wang. A relaxed dimensional factorization preconditioner for the incompressible navier–stokes equations. Journal of Computational Physics, 236(16):6815–6822, 2017.

[36] G. H. Golub, J. M. Ortega, et al. Scientific computing. volume 31999. McGraw-Hill New York, 1986.
