FOURST: A code generator for FFT-based fast stencil computations

Zafar Ahmad*, Mohammad Mahdi Javanmard†, Gregory Croisdale‡, Aaron Gregory*,
Pramod Ganapathi*, Louis-Noël Pouchet§ and Rezaul Chowdhury*

* Department of Computer Science, Stony Brook University, Stony Brook, New York, USA
{zafahmad, afgregory, pramod.ganapathi, rezaul}@cs.stonybrook.edu
†Meta Platforms, Inc., New York, USA
mjavanmard@fb.com
‡Department of Computer Science, University of Tennessee, Knoxville, Tennessee, USA
gcroisda@vols.utk.edu
§Department of Computer Science, Colorado State University, Fort Collins, Colorado, USA
pouchet@colostate.edu

***Abstract*—Stencil computations are ubiquitous in modern grid-based physical simulations. In this paper, we present FOURST – a compiler to generate programs computing time iterated linear periodic and aperiodic stencil computations with fast Fourier transform methods. This paper outlines the design and implementation of the code generation approach in FOURST, to automatically generate FFT-based stencil solvers. We present experimental results on the state-of-the-art Ookami supercomputer housing Fujitsu A64FX and Intel Skylake processors, to study the performance of FOURST and a state-of-the-art tiling-based optimized code generator PLuTo on various stencil shapes and varying the number of time iterations. We discuss the performance profiles, and limitations, of both approaches on high-end modern hardware.**

***Index Terms*—Fast-Fourier Transform, FFT, Stencil, Stencil Computations, Fast Stencil Computation, Grid Simulation, Simulation, FOURST**

I. INTRODUCTION

A stencil is a small cellular automaton-like pattern used to update the values of cells in a spatial grid; it is used once per cell per timestep. The entire process of evolving cell values in the entire spatial grid according to some stencil for a large number of timesteps is called a stencil computation [1]–[3].

A stencil computation is formally defined as follows. The initial grid data is a_0. The grid data after $i+1$ timesteps is computed by applying the stencil $\mathcal{S}$ on the grid data obtained after i timesteps, i.e., $a_{i+1} = \mathcal{S}a_i$. We want to compute the time evolution of the grid after T timesteps, that is, compute a_T from a_0 and $\mathcal{S}$.

Any of the two types of boundary conditions can be used for computing the cells at the boundary of the spatial grid: periodic and aperiodic. Periodic boundary conditions are used when the spatial grid is a *hypertorus*, i.e., every dimension wraps around itself to form a closed object. We use modular or clock arithmetic to perform calculations on a periodic grid. In contrast, aperiodic boundary conditions are used when the spatial grid is an *orthotope/hyperrectangle*, i.e., every dimension does not wrap around itself to form a closed object. Several types of aperiodic boundary conditions can be used such as Dirichlet [4] and von Neumann [5].

Stencils computations appear in a wide variety of scientific computing and engineering applications, most often for the simulation of physical systems [6]–[8] such as dynamics of smoke, fire, liquids, hair, clothing, skin, sand, and snow. They are also used for less obvious applications such as image processing [9]–[11]. These computations generally operate on a principle of "bigger is better (or more realistic)" and in some cases computation depends on varying large timesteps [12], so it is of considerable scientific interest to improve the efficiency with which these computations can be implemented and evaluated for huge grids and a large number of timesteps.

There exist several techniques to implement stencil computations. Stencils can be implemented using simple looping codes but such iterative codes do not exploit temporal data locality. Cache-aware tiled-looping programs [13]–[24] exploit temporal locality and/or parallelism however they can be less portable across machines. Cache-oblivious recursive divide-and-conquer trapezoidal algorithms [2], [3], [25]–[31] make the programs cache-efficient and portable. All these algorithms have a computational complexity of $\Theta(NT)$, where N is the grid size and T is the number of timesteps, and they work with explicit stencils[1].

Recently, Ahmad et al. [1] designed stencil algorithms based on fast Fourier transforms (FFT) that work with linear homogeneous stencils and arbitrary boundary conditions. These are the first stencil algorithms that have a computational complexity of $o(NT)$. Ahmad et al. [1] show that the implementations of these algorithms run orders of magnitude faster than that of state-of-the-art for periodic grids, and 1.3× to 8.5× faster for large aperiodic grids.

The FFT-based stencil algorithms presented in Ahmad et al. [1] are examples of solvers with optimal preprocessing, which

[1] *explicit stencils* – stencils that depend on prior timesteps.

978-1-6654-5954-9/22/$31.00 ©2022 IEEE
DOI 10.1109/ISPASS55109.2022.00010

have made them converge to the exact solution in a single step, thus rendering them also direct. By converting to the Fourier domain, they are able to take computations which would normally look like dense matrix multiplications and transform them into computations which look like vector-vector multiplications. This then allows for greatly improved computational complexity and parallelism.

These recent FFT-based solvers are by no means the first occurrence of Fourier transforms in the field of stencil computation. On the contrary, it is common for discrete Fourier transforms (DFTs) to be used either in the analysis [32] or implementations [33]–[35] of Krylov subspace methods, as Fourier analysis is useful for proving scheme stability [36], [37] and convergence rates [38], and DFT matrices are good preconditioners [39] for a large class of matrix equations [40]. In some instances choosing the DFT matrix as a Krylov preconditioner can even convert an approximate solver into a direct one [41], [42], as was the case for Ahmad et al. [1]. However, even though DFTs were commonly applied and well known as a tool for numerical analysis, they were not applied to evolve grid data for many timesteps at a time before Ahmad et al.'s work [1].

Implementing stencil algorithms to achieve high performance is hard. This process might require sophisticated techniques such as dependence analysis, recursive divide-and-conquer, parallel programming, and problem-specific mathematical analysis. Due to the complicated nature of these codes, significant effort is spent coding and debugging them.

Several automatic stencil code generators exist. For example, PLuTo [43] is an automatic parallelizing and locality optimizing code generator for affine loop nests based on the *polyhedral model* [44]. Other code generators based on the polyhedral model are [45]–[48]. The Pochoir stencil compiler [3] is a domain specific language stencil code compiler that takes a simple specification of a stencil code and automatically generates a parallel cache-oblivious implementation of the divide-and-conquer trapezoidal algorithm. There are also stencil code generators for GPUs [49]–[53] and distributed-memory machines [54]. For FPGAs direct efficient source code generator for stencil computation in not widely used but efficient tuning has been analyzed [55].

In this paper, ***we present Fourst — the first stencil code generator that automatically generates the implementations of Ahmad et al.'s [1] FFT-based stencil algorithms for linear homogenoues stencils***. As has already been shown in [1], the implementations of these FFT-based stencil algorithms are now the state-of-the-art for linear stencils on periodic and aperiodic grids with significant speedups over existing best stencil algorithms. Even though libraries such as FFTW [56] and Intel MKL [57] can be used to implement FFT, using FFT to implement the stencil algorithms is hard. Programming gets even harder if practitioners want to implement the aperiodic algorithms as these algorithms are full of recursive divide-and-conquer calls along with calls to periodic counterparts. Programming gets worse when the number of dimensions increases as the complexity of coding is proportional to the number of dimensions due to the necessity of covering all corner cases. The Fourst code generator shields researchers and practitioners from all these details by automatically generating the FFT-based stencil code for a given problem in a single click in both a web-based online and an offline mode.

For several stencil benchmarks, we compare the performance of Fourst generated programs with PLuTo generated programs on the Ookami HPE Apollo 80 system [58], [59], which is the first installation of A64FX architecture outside of Japan with state-of-the-art computing technology. Comparing periodic and aperiodic stencils, we show that as the number of timesteps increases, programs generated with Fourst eventually outperform the PLuTo generated programs, which is mostly due to the difference in their running time complexity. Additionally, we show that Fourst-generated programs may be more energy-efficient than PLuTo-generated programs, which makes them best candidates for large-scale stencil computations at data centers.

Section II summarizes the FFT-based stencil algorithms for both periodic and aperiodic boundary conditions. Section III explains different applications of the Fourst code generator and then illustrates how to generate an efficient FFT-based stencil program using Fourst from an inefficient loop-based input program. In section IV, we provide detailed experimental results illustrating the performance, scalability, and power consumption of several stencil benchmarks on two different architectures of Ookami. Finally, Section V concludes the paper.

II. FFT-based Stencil Algorithms

In this section, we summarize Ahmad et al.'s FFT-based stencil algorithms [1] for both periodic and aperiodic boundary conditions. We first describe the applicability of the algorithms. We then describe the periodic and aperiodic versions of the algorithms.

A. Applicability

The FFT-based stencil algorithms are applicable to homogeneous linear stencils across vector-valued fields. A homogeneous stencil remains the same throughout the grid. A vector-valued field assumes that each cell is treated as a vector. The algorithms are not applicable to nonlinear stencils, that includes stencils with conditionals (e.g. max, min, if-else ladder) and quadratic and cubic dependence. The algorithms cannot also be applied to inhomogeneous stencils where different regions of the spatial grid might have different stencils.

B. Periodic FFT-based Stencil Algorithm

Let a_0 and a_T be the initial and final data, respectively, of the d-dimensional spatial grid of total volume N. We are given a_0 and we want to compute a_T using stencil $\mathcal{S}$ and periodic boundary conditions. We want to compute

$$a_T = \mathcal{S}(\mathcal{S}(\cdots \mathcal{S}(a_0) \cdots)),$$

Algorithm	Class	Work (T_1)	Span (T_∞)	Code Generators
Nested Loop	Explicit	$\Theta(NT)$	$\Theta(T \log N)$	
Tiled Loop [14]	Explicit	$\Theta(NT)$	$\Theta\left(T \log M + \frac{T}{M^{1/d}} \log \frac{N}{M}\right)$	PLuTo [14]
D&C [60]	Explicit	$\Theta(NT)$	$\Theta\left(T(N^{1/d})^{\log(d+2)-1}\right)$	Pochoir [60]
FFT-P [1]	Linear homogeneous	$\Theta(N \log(NT))$	$\Theta(\log N \log\log N)$	👍 Fourst
FFT-A [1]	Linear homogeneous	$\Theta\left(TN^{1-1/d} \log\left(TN^{1-1/d}\right) \log T + N \log N\right)$	$\begin{cases}\Theta(T) & \text{if } d = 1\\ \Theta(T \log N) & \text{if } d \geq 2\end{cases}$	👍 Fourst

TABLE I
COMPLEXITY ANALYSIS AND CODE GENERATORS TO IMPLEMENT EXISTING STENCIL ALGORITHMS. HERE, CLASS REPRESENTS THE CLASS OF STENCILS, N = HYPERRECTANGULAR GRID SIZE, T = #TIMESTEPS, M = CACHE SIZE, AND 👍 REPRESENTS THIS PAPER. WE ASSUME THAT STENCIL SIZE AND d ARE $\Theta(1)$. ALGORITHMS NOT BASED ON FFT WORK ON BOTH LINEAR AND NONLINEAR STENCILS. FFT-BASED ALGORITHMS WORK ON BOTH EXPLICIT AND IMPLICIT LINEAR STENCILS. FFT-A ALGORITHM SPAN IS SIMPLIFIED ASSUMING $T = \Omega(\log N \log\log N)$. THE SPAN FOR THE TILED LOOP ALGORITHM IS $\Omega(T \log\log N)$.

where $\mathcal{S}$ is applied for T times. If we represent $\mathcal{S}$ as a *circulant matrix* then a_T can be written as

$$a_T = \mathcal{S}^T a_0.$$

That is, applying $\mathcal{S}$ repeatedly on a_0 for T times is same as applying the big stencil $\mathcal{S}^T$ on a_0 once.

As the direct computation of $\mathcal{S}^T a_0$ is inefficient, we perform this computation efficiently by moving both a_0 and $\mathcal{S}$ to the Fourier domain. Let the discrete Fourier transform (DFT) matrix be $\mathcal{F}[i,j] = \omega_N^{-ij}/\sqrt{N}$, where $\omega_N = e^{2\pi\sqrt{-1}/N}$, and let $\mathcal{F}^{-1}$ (or IFFT) be the inverse DFT matrix. We have

$$\begin{aligned}
a_T &= \mathcal{S}^T a_0 && (1)\\
&= (\mathcal{F}^{-1}\mathcal{F})\mathcal{S}^T(\mathcal{F}^{-1}\mathcal{F})a_0 \qquad (\mathcal{F}^{-1}\mathcal{F} = \text{identity}) && (2)\\
&= \mathcal{F}^{-1}(\mathcal{F}\mathcal{S}^T\mathcal{F}^{-1})(\mathcal{F}a_0) \qquad (\text{reorder}) && (3)\\
&= \mathcal{F}^{-1}(\mathcal{F}\mathcal{S}\mathcal{F}^{-1})^T(\mathcal{F}a_0) \qquad (\mathcal{F}\mathcal{S}^T\mathcal{F}^{-1} = (\mathcal{F}\mathcal{S}\mathcal{F}^{-1})^T) && (4)\\
&= \mathcal{F}^{-1}\left((\mathcal{F}\mathcal{S}\mathcal{F}^{-1})^T(\mathcal{F}a_0)\right) \qquad (\text{regroup}) && (5)\\
&= \text{InverseFFT(BigStencil} \otimes \text{InitialData)} && (6)
\end{aligned}$$

Here, BigStencil refers to $(\mathcal{F}\mathcal{S}\mathcal{F}^{-1})^T$, which is in fact the big stencil generated at time step T in the preprocessing stage being in the Fourier domain; InitialData refers to the initial data in the Fourier domain $\mathcal{F}a_0$; and $\otimes$ represents the elementwise product. The block diagram for computing a_T from a_0 and $\mathcal{S}$ using FFT (through Equation 6) is shown in Figure 1. The algorithm is shown in Figure 2. The complexity of the periodic algorithm is given in Table I.

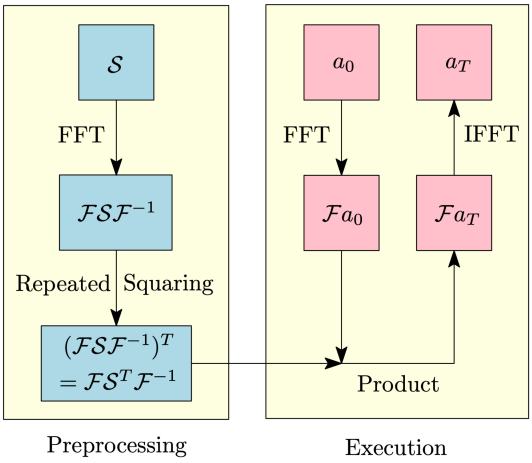

Fig. 1. Block diagram of FFT-based stencil algorithm for periodic grids.

FFTSTENCIL-PERIODIC$(a_0, \mathcal{S}, T)$

Input: Initial grid data a_0, stencil $\mathcal{S}$, and evolution time T.
Output: Final grid data a_T.

[Preprocessing Stage]
1) $\mathcal{F}\mathcal{S}\mathcal{F}^{-1} \leftarrow$ Use FFT on stencil $\mathcal{S}$
2) $\mathcal{F}\mathcal{S}^T\mathcal{F}^{-1} \leftarrow$ Apply repeated squaring using binary representation of T

[Execution Stage]
3) $\mathcal{F}a_0 \leftarrow$ Apply FFT on initial grid data a_0
4) $\mathcal{F}a_T \leftarrow$ Take elementwise product of $\mathcal{F}\mathcal{S}^T\mathcal{F}^{-1}$ and $\mathcal{F}a_0$
5) $a_T \leftarrow$ Apply IFFT on $\mathcal{F}a_T$
6) **return** a_T

Fig. 2. FFT-based stencil algorithm for periodic grids.

C. Aperiodic FFT-based Stencil Algorithm

In this section, we describe the aperiodic algorithm first presented in [1]. This algorithm uses a divide-and-conquer strategy to improve the asymptotic complexity of evolving grid data in the presence of aperiodic boundary conditions. These boundary conditions are a very large class, consisting of everything which cannot be represented periodically. The algorithm can evolve data across a grid of N cells for T timesteps in serial time complexity $\Theta\left(TN^{1-1/d} \log\left(TN^{1-1/d}\right) \log T + N \log N\right)$ and span $\Theta(T)$ for 1-D grids and $\Theta(T \log N)$ for higher dimensions.

The core idea of the aperiodic algorithm is to recursively break up the spacetime grid by performing time cuts, and to solve blocks of the grid by applying the periodic solver locally. This works because the cells A whose values are used to compute new values in some region B can be found by tracing back in time (expanding in all directions) from the new cells B, and if no boundary cells are contained in A then the value of cells in B is completely independent of boundary conditions. We can therefore use a periodic solver

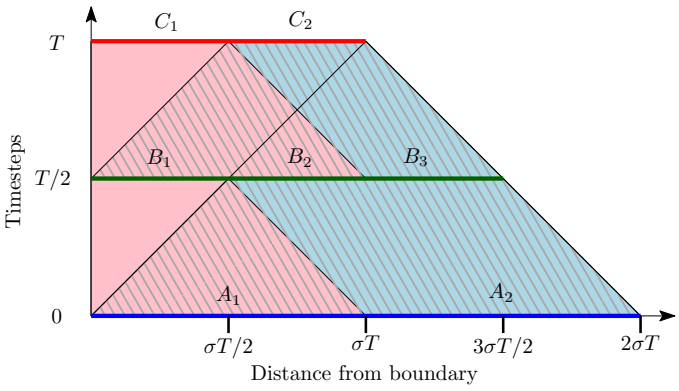

Fig. 3. Block diagram of the recursive divide-and-conquer FFT-based stencil algorithm for aperiodic grids.

FFTStencil-Aperiodic$(a_0, \mathcal{S}, T)$

Input: Initial grid data a_0, stencil $\mathcal{S}$, and total evolution time T. Consider $a_0 = A_1 \cup A_2$ (see Figure 3).
Output: Final grid data a_T. Consider $a_T = C_1 \cup C_2$.

1) If T is small, then execute the base case and return
   [Solving for time range $[0, T/2]$**]**
2) $(B_2 \cup B_3) \leftarrow$ FFTStencil-Periodic$(A_1 \cup A_2, \mathcal{S}, T/2)$
3) $B_1 \leftarrow$ FFTStencil-Aperiodic$(A_1, \mathcal{S}, T/2)$
   [Solving for time range $[T/2, T]$**]**
4) $C_2 \leftarrow$ FFTStencil-Periodic$(B_1 \cup B_2 \cup B_3, \mathcal{S}, T/2)$
5) $C_1 \leftarrow$ FFTStencil-Aperiodic$(B_1 \cup B_2, \mathcal{S}, T/2)$
6) $a_T \leftarrow (C_1 \cup C_2)$
7) **return** a_T

Fig. 4. A divide-and-conquer FFT-based stencil algorithm for aperiodic grids.

to compute B from A. As more time cuts are performed, the cells which can be computed via a periodic solver will make up a greater and greater fraction of the entire spatial grid. The scheme based on this idea is shown diagrammatically in Figure 3 and detailed in the pseudocode given in Figure 4.

III. The Fourst Stencil Code Generator

In this section, we first explain how to use Fourst. We then explain how Fourst generates FFT-based stencil computation programs from specific loop-based input programs.

A. Using Fourst

The Fourst tool can be used either the web-based online interface, or offline as a stand-alone compiler built from sources.

The online tool provides a user interface to obtain stencil coefficients in both dense and sparse representation in any number of dimensions. A demo of the Fourst web tool can be accessed via the following link: https://tealab-org.github.io/fourst-web/.

The second way to use Fourst is to enclose the loop-based stencil code within `#pragma BEGIN_FOURST` and `#pragma END_FOURST` as shown in Listing 1.

The command-line tool is run simply as `./fourst input.c output.c`. The Fourst-CLI repository is accessible at https://github.com/TEALab-org/fourst-cli

```
1 void function(...) {
2  #pragma FOURST boundary 0
3  #pragma FOURST begin aperiodic
4  for (int t = 0; t < 30; t++)
5   for (int i = 0; i < 30; i++)
6     mat[t][i] = -3 * mat[t-1][i-1] + mat[t-1][i] +
7                 13 * mat[t-1][i+1];
8  #pragma FOURST end
9  // The rest of the computation ...
10 }
```

Listing 1. Using `#pragma` directives in the input program.

Though customized boundary conditions can be specified for specifying aperiodic stencil computation, the current version of Fourst supports only the *Dirichlet boundary conditions*. We will support more boundary condition types in future versions.

B. Design of Fourst

Both Fourst-Web and Fourst-CLI utilize a code generation mechanism implemented in C++. This generation mechanism takes designated environment options and a stencil matrix, and returns the FFT-optimized stencil computation code for any arbitrary grid dimension. The web component uses a WebAsm version of the generation script created using the emscripten compiler. WebAsm can run natively on many modern browsers and can be interpreted through JavaScript for unsupported platforms. The CLI component's build process involves the compilation of the generation script using the native C++ Compiler.

C. Code generation

Our code generator emits efficient parallel source code for both periodic and Dirichlet aperiodic boundary condition stencil computations. The code generator module gets the high level problem description from the web tool or the CLI module. The high level problem description contains the class of the computation (periodic or aperiodic), radius of the stencil (σ), coefficient matrix (sparse format or dense format), and the preferred FFT Library information. If the provided coefficient matrix is in dense format, the code generator picks the non-zero data points to reduce computation and emits respective optimized code for the base case looping-based computation. Our technique emits dynamic memory allocated source code with FFT library descriptors for efficient memory usage.

Aperiodic boundary condition stencil computation requires a manual looping-based implementation in the time iteration for boundary adjustments. Efficient implementation of the aperiodic algorithm requires a trade-off between recursion depth and base case computation size described in the aperiodic algorithm in Figure 4. Our code generation technique allows users to find the best combination using the provided input arguments in the emitted source code. At the base case computation, careful loop iteration is required for

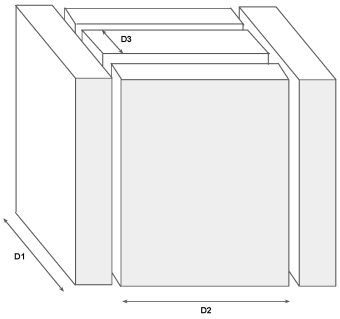

Fig. 5. Recursive exploration of aperiodic boundary condition base cases.

efficient implementation of the stencil computation. We use a recursive algorithm in our code generator to emit base case computation for multi-dimensional stencil computations. Figure 5 shows how our code generator recursively explores each dimension of the base case. To avoid the redundant computation in the emitted code, our code generator excludes the cells that are already computed while exploring multiple dimensions. In Figure 5, the second dimension ($D2$) excludes the computed cells at the boundaries in first explored dimension ($D1$). For the third explored dimension ($D3$), our recursive algorithm excludes the computed cells in both $D1$ and $D2$.

IV. Evaluation

In this section, we present and analyze experimental results for benchmarking 1D, 2D, and 3D heat computations, using both periodic and aperiodic Dirichlet boundary conditions, on two significantly different architectures: x86-64 and ARMv8.2.

A64FX	Cores	12 cores per CMG, 4 CMG (total: 48 cores)
	Cache sizes	L1 64 KB, L2 8 MB (shared), Per CMG 8 GB
	Memory	512 GB SSD
	Compiler	fujitsu (FCC) v4.5
	SIMD Instruction	SVE (512 wide)
	Compiler Flags	-Kfast -KSVE -Koptmsg=2
SKX	Cores	Dual socket 36 physical cores
	Cache sizes	L1 32 KB, L2 1 MB, L3 33 MB
	Compiler	Intel C++ Compiler (ICC) v19.1.2.254
	SIMD Instruction	AVX512
	Compiler Flags	-xhost -ansi-alias -ipo -AVX512
Parallelization		OpenMP 4.1.0

TABLE II
Experimental setup on the Ookami Supercomputer using Intel Skylake (SKX) and A64FX nodes.

Benchmark	`heat1d`	`heat2d`	`heat3d`
Stencil points (α)	3pts	5pts	7pts
Stencil radius (σ)	1	1	1

TABLE III
Benchmark problems with the number of points in the corresponding stencils.

A. Experimental Setup

The Ookami System: The Ookami HPE Apollo 80 system [58], [59] is the first system outside of Japan, which hosts A64FX machines with state-of-the-art computing technology. It includes 176 A64FX compute nodes running at 1.8GHz, each with 32GB high-bandwidth memory, 48 cores, and a 512GB SSD. The A64FX-700 series processor used in Ookami has 48 cores arranged in four core memory groups (CMG) of 12. Each CMG forms a NUMA region with 8 Gbyte of high-bandwidth memory (256 Gbyte/s). Other nodes available on Ookami are a dual socket AMD Milan (64 cores) with 512GB memory, an Intel Skylake (32 cores) with 192 GB memory including two Nvidia V100 GPUs. This allows users to directly compare portability and performance across all current major architectures. We focused our experiments on Intel Skylake nodes, but we also present early results using the Fujitsu A64FX and its associated compilation toolchain.

Benchmarks.: For benchmarking, we use a 1D 3-point, 2D 5-point, 3D 7-point heat stencil for both periodic and aperiodic boundary condition stencil computation. The heat stencil computation computes the heat equation which is a partial differential equation that models the physical transfer of heat in a region over time. The following partial differential equation (PDE) describes heat diffusion in two-dimensional space:

$$\frac{\partial h_t(x,y)}{\partial t} = \alpha \left(\frac{\partial^2 h_t(x,y)}{\partial x^2} + \frac{\partial^2 h_t(x,y)}{\partial y^2} \right),$$

where, $h_t(x,y)$ is the heat at a point (x,y) at time t and α is the thermal diffusivity.

By discretizing space and time, we obtain the following stencil equation for approximating the heat diffusion PDE given above.

$$\begin{aligned} h_{t+1}(x,y) = h_t(x,y) \\ + \frac{\alpha \Delta t}{\Delta x^2} \left(h_t(x-1,y) + h_t(x+1,y) - 2h_t(x,y) \right) \\ + \frac{\alpha \Delta t}{\Delta y^2} \left(h_t(x,y-1) + h_t(x,y+1) - 2h_t(x,y) \right). \end{aligned}$$

We use the benchmarks directly from [61].

PLuTo-Generated tiled Programs.: The tiled loop implementations were generated using PLuTo [43]. We used the latest version (v0.11.4-963-ge5a0390) from Github, but for some cases, it generated tiled codes with poor performance and sometimes errors. Hence, we also used a prior released version (v0.11.4). We compared our FFT-based auto-generated algorithm implementation results with the PLuTo version that provided a better runtime. For the outer and the innermost dimensions we explored the tile sizes {8, 16} and {32, 64, 128} respectively [43]. For other dimensions we kept the tile size 16 in our tile exploration step.

Fourst-generated FFT-based programs.: Fourst can generate both FFTW [56] and Intel MKL [57]-based implementations. The Intel MKL Library is not available on A64FX nodes. Therefore, the Fujitsu (FCC) compiler with opensource FFTW [56] (v3.3.10) was used to benchmark Fourst-generated programs. For the Skylake node, we used the Intel MKL [57]-based Fourst-generated programs with the Intel compiler.

Stencil Grid Sizes.: The stencil grid sizes for our benchmarks were $1.6m$, $8K \times 8K$, and $300 \times 300 \times 300$ for 1D, 2D, and 3D respectively.

One key aspect we explore in our experiments is the relationship between the number of time iterations to be

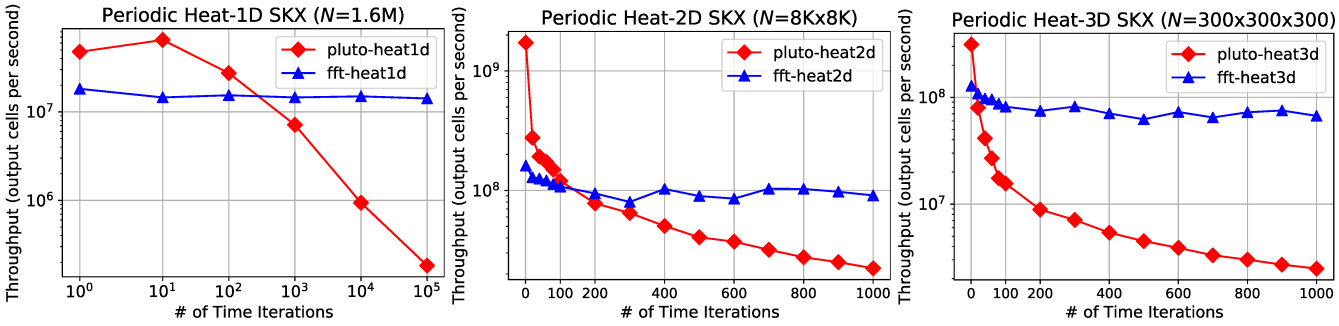

Fig. 6. Throughput of Periodic Heat 1D, 2D, and 3D Stencil Computations on SKX Machine

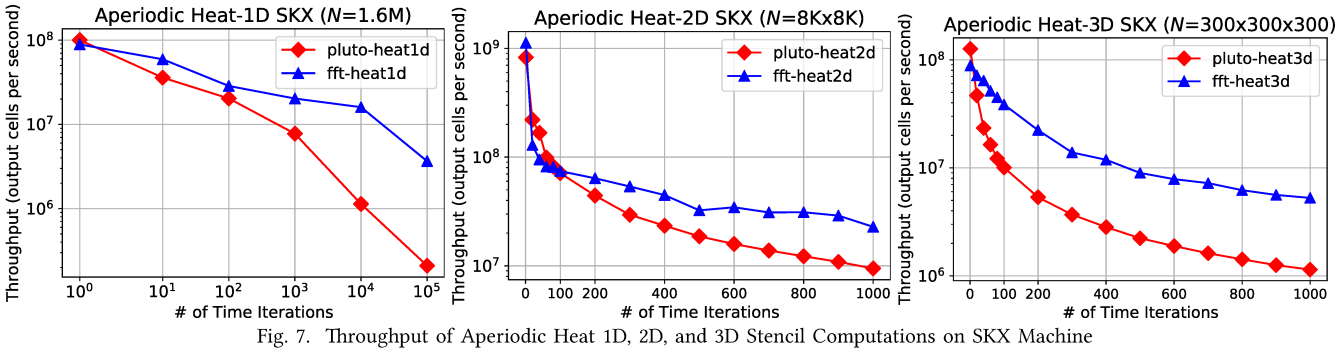

Fig. 7. Throughput of Aperiodic Heat 1D, 2D, and 3D Stencil Computations on SKX Machine

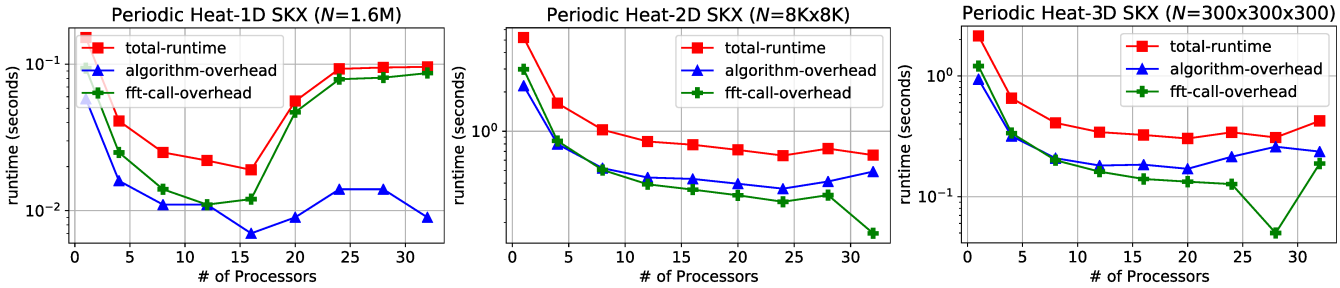

Fig. 8. Scaling of Periodic Heat 1D, 2D, and 3D Stencil Computations on SKX Machine

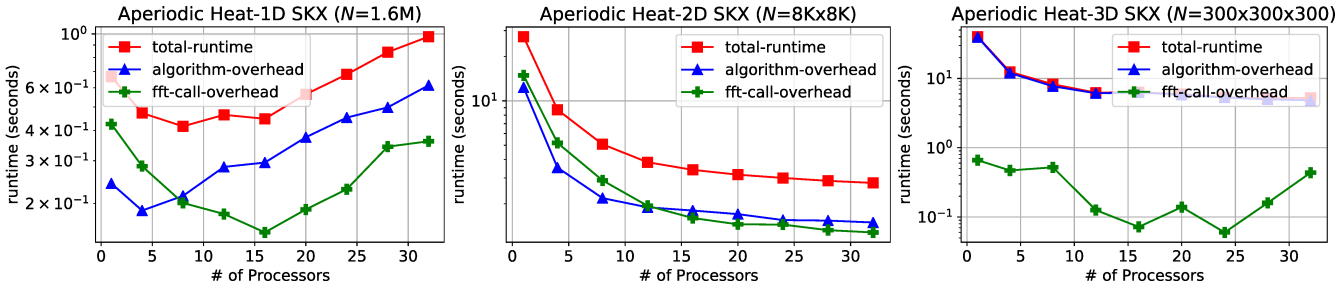

Fig. 9. Scaling of Aperiodic Heat 1D, 2D, and 3D Stencil Computations on SKX Machine

performed, and the relative performance of two fundamentally different approaches: one based on state-of-the-art time-tiling using diamond shapes [61], and one based on a high preprocessing cost but with near constant computational cost afterwards [1]. We therefore used 1000 time iterations for the 2D and 3D computations, and 10^5 iterations for 1D, to show longer trends.

B. Experimental Results on Intel Skylake

In this section we summarize the experimental results on Intel Skylake. Our experimental results has been evaluated in both the Intel Skylake (SKX) machine and the A64FX processors on the Ookami machine.

Figures 6 - 9 show the throughput compared against PLuTo and scalability plots of Fourst-generated programs for both periodic and aperiodic stencil boundary conditions. We define throughput as the number of output cells computed per second. Precisely, this metric contrasts the two approaches from an end-user point of view, as it reflects exactly execution time.

The number of operations performed by the processor is linear with the number of time iterations with the time-tiling

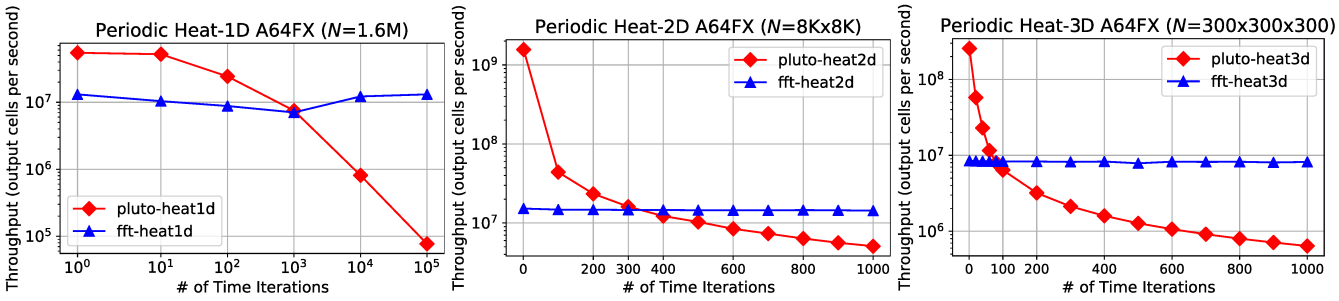

Fig. 10. Throughput of Periodic Heat 1D, 2D, and 3D Stencil Computations on A64FX Machine

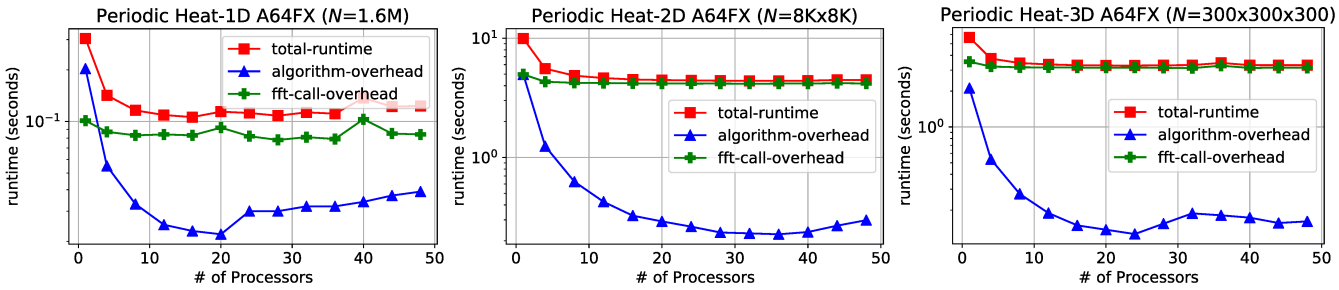

Fig. 11. Scaling of Periodic Heat 1D, 2D, and 3D Stencil Computations on A64FX Machine

approach of PLuTo, contrary to Fourst-generated codes. Consequently, with the increase of the number of time iterations, Fourst-generated programs eventually outperform PLuTo-generated tiled implementations. Figures 6, 7 and 10 allow to study the break-even point, that is the number of consecutive time steps to be executed before Fourst-generated programs start outperforming PLuTo-generated ones, for the experimental setup we considered.

For the cases of both periodic and aperiodic boundaries displayed in Figure 6 and 7, we observe that PLuTo codes systematically outperform Fourstones for very small number of timesteps. This is expected, and explained in large part due to the heavy preprocessing step required with the FFT-based approach, which requires at least several timesteps to be amortized overall. The highly regular loop-based stencil computation which has been parallelized and SIMD-vectorized executes at a reasonably constant fraction of the machine peak, but its workload is linear with the number of time iterations, thereby making the throughput per cell diminish regularly with the number of time iterations.

In contrast, the Fourst-generated codes achieve near-linear throughput in terms of output cells per second for periodic stencils as shown in Figure 6, as expected for such workload. In our experiments, the break-even points were around 400 time iterations in 1D, 120 in 2D and 50 in 3D, computing more time iterations tend to reinforce the performance advantage of FFT-based codes versus tiled ones.

For aperiodic stencils, the throughput of Fourst codes slows with the number of time iterations due to the number of computations required by the FFT approach, but remain very significantly faster with a gap tending to increase with the number of time iterations. The break-even points are around 8 time iterations in 1D, 100 in 2D and 30 in 3D.

We remark that for all experiments above, the break-even point, in terms of number of time iterations, may fluctuate across different machines, benchmarks, but also as a function of the tuning of the generated codes (e.g. tile size selection, SIMD code generation), for each approach. However the fundamental trends, originating from the time complexity differences between these approaches, is exemplified unambiguously in Figures 6 and 7.

C. Experimental Results on A64FX

Figures 10 and 11 show the throughput and scalability plots for Fourst-generated programs. Fourst-generated programs also outperform PLuTo-generated tiled codes in A64FX nodes. However, the throughput values are lower when compared to the Skylake machine due to the fact that the A64FX machine and its compilers are still not matured to generate fully optimized executables. The compilation flag *-Koptmsg* in Fujitsu compiler show that the compiler fails to SIMD conversion if any of the statement inside the loop has any conditional statement.

For stencil computations, boundary conditions typically require conditional statements, hence even with the *-KSVE* flag, the Fujitsu compiler fails to apply SIMD conversion and software pipelining. However, there have been studies [62], [63] to improve data-level parallelism for scalable vector extentions. Fourst-generated programs uses the MKL or the FFTW library for FFT implementation. Since the Intel MKL library is not available on A64FX nodes, we generated FFTW-based implementation using Fourst. Burford et al. [58] benchmarked a different standard library as part of an early study of the Ookami nodes. The benchmarks show the parallel FFTW installation fails to achieve 1% of the peak performance on the A64FX nodes. This bottleneck shows

Timesteps	Running time	Per-core power	Per-CMG L2 power	Per-CMG HBM power	Running time	Per-core power	Per-CMG L2 power	Per-CMG HBM power
200	8.45	1.993	1.856	2.155	3.257	2.031	1.912	2.163
400	16.90	1.992	1.856	2.155	3.262	2.035	1.917	2.174
600	25.33	1.993	1.857	2.151	3.278	2.028	1.936	2.201
800	33.79	1.993	1.858	2.150	3.282	2.030	1.925	2.183
1000	42.21	1.992	1.860	2.154	3.300	2.025	1.954	2.221

TABLE IV
POWER CONSUMPTION REPORT (IN WATTS) OF HEAT_3D PERIODIC STENCIL COMPUTATION OF PLUTO (LEFT) VERSUS FOURST (RIGHT).

up in our scaling plots. Initially, the overall running times of FOURST-generated programs don't seem to scale with the number of cores, but this is due to our benchmarking of the overhead of the FFTW API calls. The scalability plots confirm that the main scaling bottleneck is the FFTW API calls in FOURST-generated programs. FOURST-generated aperiodic boundary condition stencil computations heavily use FFTW API calls at each level of recursion. Since our periodic results show the current FFTW installation is not optimized enough to use FFTW APIs heavily, we did not benchmark our aperiodic algorithms on A64FX nodes.

D. Initial Results on Power Consumption

FOURST-generated programs shows the energy efficiency over the PLuTo-generated tiled algorithms. Table IV summarizes the energy consumption in watts *per cycle of core*, for the *L2 cache*, and *CMG local HBM memory* from counters available on the nodes.

The results show energy consumption per cycle is similar for both FOURST-generated programs and PLuTo-generated tiled loop programs. However, the execution time of PLuTo-generated programs are higher compared to FOURST-generated programs. Consequently, the total consumed energy for FOURST-generated programs are lower. Further analysis is required to determine the opportunities to further increase energy efficiency of both types of computations on A64FX. Solutions ranging from careful dynamic voltage and frequency scaling [64] to generation of more energy efficient SIMD code may be investigated.

E. Discussions and Future Work

Performance bottlenecks are vastly different for Pluto-style codes vs. FOURST-style codes. Improving the performance of loop-based stencil computations has been extensively studied, with techniques to increase data locality without limiting parallelism [43], [65] as well as for efficient SIMD code generation [66], [67]. For high performance one may resort to data layout transformations to ensure ideal alignment of data with SIMD vector slots [66], [68], as well as further improve the register usage and instruction level parallelism available for scuh stencils [51], [67]. For time-iterated stencils, one may unroll the time loop and/or the stencil itself to produce a high-order operator (typically with a stencil of a larger size and radius), however as demonstrated by Stock et al. high-order stencils quickly suffer from register spilling and see their performance in FLOP/s even decrease with respect to lower order stencils [67]. A stencil-specific optimization based on exploiting associativity of reductions may then be used to reduce register pressure and increase performance [67]. In this work, we did not search for the utmost performance achievable for loop-based iterative stencils: we limited to using the same benchmarks and experimental setup as presented in [43], for fair comparison. We did however do a stage of tile size tuning as described above.

Performance limitations of FFT implementations have been vastly studied, and high-performance customized code generators have been proposed, e.g. [69] and high-performance libraries are available such as FFTW [56]. Ayala et al. studied the parallel scalability of FFTW on A64FX [70], and Brier et al. provided careful initial characterization of the performance of Ookami, including FFTW's current deployment [71]. Its performance appear currently limited for lack of quality support of SVE SIMD extensions of the A64FX in the deployed version of FFTW. Progresses on the performance of FFT for A64FX processors are expected [72] and we hope to achieve significantly higher performance with FOURST-generated codes simply with more optimized FFT libraries for the target machines.

Overall the experimental study presented exposes the limits of FOURSTfor small number of time steps, where the preprocessing time of FOURSTcurrently far exceeds the time to execute a few time iterations with an efficient tiled implementation. We expect to significantly improve the performance of FOURSTcodes on Ookami, by improving the performance of the underlying FFT library first, ensuring SIMD vectorization is well exploited. However, the fundamental nature of each method suggests loop-based methods are likely to remain more efficient versus FOURSTfor small number of time steps, whichever the actual break-even point on one's setup.

V. CONCLUSION

In this paper, we introduced FOURST, the first stencil code generator that automatically generates the implementations of Ahmad et al.'s [1] FFT-based stencil algorithms for linear homogeneous stencils. We summarized the FFT-based stencil algorithms and theoretically and experimentally compared FOURST generated programs with PLuTo generated programs. Our experiments showed when the FOURST generated programs are more throughput efficient than the PLuTo generated tiled programs. The FOURST code generator is a work in progress which opens several research opportunities, including extending it to non-linear stencil computations, supporting more complex boundary conditions, and auto-generating programs for GPUs and distributed-memory systems.

Acknowledgement. This research was supported in part by the U.S. National Science Foundation, awards CNS-1553510, CCF-1750399 and CCF-2009020. The authors thank Stony Brook Research Computing and Cyber infrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a $5M U.S. National Science Foundation grant (#1927880). Part of this work used the Extreme Science and Engineering Discovery Environment (XSEDE) (XSE; Towns et al. 2014), which is supported by NSF grant ACI-1548562. Authors used the XSEDE resources available through startup allocation grant TG-ASC190066.

References

[1] Z. Ahmad, R. Chowdhury, R. Das, P. Ganapathi, A. Gregory, and Y. Zhu, "Fast Stencil Computations Using Fast Fourier Transforms," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021.

[2] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th annual international conference on Supercomputing*, 2005.

[3] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, 2011.

[4] T. Wang and G. W. Hohmann, "A finite-difference, time-domain solution for three-dimensional electromagnetic modeling," *Geophysics*, 1993.

[5] W. Liao, "A high-order adi finite difference scheme for a 3d reaction-diffusion equation with neumann boundary condition," *Numerical Methods for Partial Differential Equations*, 2013.

[6] T. Pang, *An introduction to computational physics*. American Association of Physics Teachers, 1999.

[7] T. J. Barth and H. Deconinck, *High-order methods for computational physics*. Springer Science & Business Media, 2013.

[8] B. Hamilton, C. Webb, A. Gray, and S. Bilbao, "Large stencil operations for gpu-based 3-d acoustics simulations," in *Proceedings of the 18th International Conference on Digital Audio Effects*, Nov. 2015.

[9] J. Weickert, *Applications of nonlinear diffusion in image processing and computer vision*. Acta Mathematica Universitatis Comenianae, 2000.

[10] G. Peyré, "The numerical tours of signal processing-advanced computational signal and image processing," *IEEE Computing in Science and Engineering*, 2011.

[11] L. A. Vese and S. J. Osher, "Numerical methods for p-harmonic flows and applications to image processing," *SIAM Journal on Numerical Analysis*, 2002.

[12] A. T. Layton and M. L. Minion, "Implications of the choice of quadrature nodes for picard integral deferred corrections methods for ordinary differential equations," *BIT Numerical Mathematics*, Jun. 2005.

[13] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[14] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[15] M. J. Wolfe, "Iteration space tiling for memory hierarchies," *Parallel Processing for Scientific Computing*, 1987.

[16] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *ACM SIGPLAN conference on Programming language design and implementation*, 1991.

[17] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *IEEE/ACM International Symposium on Microarchitecture*, 1996.

[18] D. Wonnacott, "Achieving scalable locality with time skewing," *International Journal of Parallel Programming*, 2002.

[19] U. Bondhugula, A. Acharya, and A. Cohen, "The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests," *ACM Transactions on Programming Languages and Systems*, 2016.

[20] R. Andonov and S. Rajopadhye, "Optimal orthogonal tiling of 2-d iterations," *Journal of Parallel and Distributed computing*, 1997.

[21] K. Högstedt, L. Carter, and J. Ferrante, "Selecting tile shape for minimal execution time," in *ACM Symposium on Parallel algorithms and architectures*, 1999.

[22] J. Zhang, T. M. Low, Q. Guo, and F. Franchetti, "A 3 d-stacked memory manycore stencil accelerator system," 2015.

[23] T. M. Malas, G. Hager, H. Ltaief, and D. E. Keyes, "Towards energy efficiency and maximum computational intensity for stencil algorithms using wavefront diamond temporal blocking," *ArXiv*, vol. abs/1410.5561, 2014.

[24] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," Jan. 2015.

[25] M. Frigo and V. Strumpen, "The cache complexity of multithreaded cache oblivious algorithms," *Theory of Computing Systems*, 2009.

[26] E. P. Natarajan, M. M. Dehnavi, and C. Leiserson, "Autotuning divide-and-conquer stencil computations," Aug. 2017.

[27] K. Sato, H. Takizawa, K. Komatsu, and H. Kobayashi, "Automatic tuning of CUDA execution parameters for stencil processing." Springer New York, Aug. 2010.

[28] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations." Association for Computing Machinery (ACM), Jun. 2007.

[29] M. M. Javanmard, P. Ganapathi, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury, "Toward efficient architecture-independent algorithms for dynamic programs," in *Lecture Notes in Computer Science*. Springer International Publishing, 2019.

[30] M. M. Javanmard, Z. Ahmad, M. Kong, L.-N. Pouchet, R. Chowdhury, and R. Harrison, "Deriving parametric multi-way recursive divide-and-conquer dynamic programming algorithms using polyhedral compilers," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. ACM, Feb. 2020.

[31] M. M. Javanmard, Z. Ahmad, J. Zola, L.-N. Pouchet, R. Chowdhury, and R. Harrison, "Efficient execution of dynamic programming algorithms on apache spark," in *2020 IEEE International Conference on Cluster Computing*. IEEE, Sep. 2020.

[32] J. P. Boyd, *Chebyshev and Fourier spectral methods*. Courier Corporation, 2001.

[33] O. Andreussi, I. Dabo, and N. Marzari, "Revised self-consistent continuum solvation in electronic-structure calculations," *The Journal of chemical physics*, 2012.

[34] M. Kabel, T. Böhlke, and M. Schneider, "Efficient fixed point and newton–krylov solvers for fft-based homogenization of elasticity at large deformations," *Computational Mechanics*, 2014.

[35] Y. Guan and I. Novosselov, "Two relaxation time lattice boltzmann method coupled to fast fourier transform poisson solver: Application to electroconvective flow," *Journal of computational physics*, 2019.

[36] T. F. Chan, "Stability analysis of finite difference schemes for the advection-diffusion equation," *SIAM journal on numerical analysis*, 1984.

[37] S. B. Yuste and L. Acedo, "An explicit finite difference method and a new von neumann-type stability analysis for fractional diffusion equations," *SIAM Journal on Numerical Analysis*, 2005.

[38] B. Gmeiner, T. Gradl, F. Gaspar, and U. Rüde, "Optimization of the multigrid-convergence rate on semi-structured meshes by local fourier analysis," *Computers & Mathematics with Applications*, 2013.

[39] Y. A. Erlangga, C. W. Oosterlee, and C. Vuik, "A novel multigrid based preconditioner for heterogeneous helmholtz problems," *SIAM Journal on Scientific Computing*, 2006.

[40] R. Borrell, O. Lehmkuhl, F. X. Trias, and A. Oliva, "Parallel direct poisson solver for discretisations with one fourier diagonalisable direction," *Journal of computational physics*, 2011.

[41] R. W. Hockney, "A fast direct solution of poisson's equation using fourier analysis," *Journal of the ACM*, 1965.

[42] V. Fuka, "Poisfft–a free parallel fast poisson solver," *Applied Mathematics and Computation*, 2015.

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

[44] M. Griebl, C. Lengauer, and S. Wetzel, "Code generation in the polytope model," in *In IEEE PACT*. IEEE Computer Society Press, 1998.

[45] "LLVM Framework for High-Level Loop and Data-Locality Optimizations," https://polly.llvm.org/.

[46] "The Polyhedral Compiler Collection." [Online]. Available: http://web.cs.ucla.edu/~pouchet/software/pocc/
[47] T. Yuki and L.-N. Pouchet, "Polybench 4.0," 2015. [Online]. Available: https://sourceforge.net/projects/polybench/
[48] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization*, 2013.
[49] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011.
[50] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures." Proceedings of the 26th ACM international conference on Supercomputing, 2012.
[51] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance gpu code for stencil computations," *Proceedings of the IEEE*, 2018.
[52] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on GPU clusters." Proceedings of the Tenth International Symposium on Code Generation and Optimization, 2012.
[53] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures." Proceedings of the 26th ACM international conference on Supercomputing, 2012.
[54] M. Sourouri, S. B. Baden, and X. Cai, "Panda: A compiler framework for concurrent cpu+ gpu execution of 3d stencil computations on gpu-accelerated supercomputers," *International Journal of Parallel Programming*, 2017.
[55] Q. Jia and H. Zhou, "Tuning stencil codes in OpenCL for FPGAs," in *IEEE 34th International Conference on Computer Design*, Oct. 2016.
[56] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, 2005.
[57] "Intel math kernel library," https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html, Intel MKL.
[58] A. Burford, A. C. Calder, D. Carlson, B. Chapman, F. CoŞKun, T. Curtis, C. Feldman, R. J. Harrison, Y. Kang, B. Michalow-Icz *et al.*, "Ookami: Deployment and initial experiences," *arXiv preprint arXiv:2106.08987*, 2021.
[59] "The Ookami HPE Apollo 80 system," https://www.stonybrook.edu/commcms/ookami/.
[60] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *ACM Symposium on Parallelism in Algorithms and Architectures*, 2011.
[61] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache, "Tiling and optimizing time-iterated computations over periodic domains," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques*. IEEE, 2014.
[62] A. Armejach, H. Caminal, J. M. Cebrian, R. Langarita, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Using arm's scalable vector extension on stencil codes," Apr. 2019.
[63] A. Armejach, H. Caminal, J. M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Stencil codes on a vector length agnostic architecture," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, Nov. 2018.
[64] W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan, "Static and dynamic frequency scaling on multicore CPUs," *ACM Transactions on Architecture and Code Optimization*, Dec. 2016. [Online]. Available: https://doi.org/10.1145/3011017
[65] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 127–138.
[66] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *International Conference on Compiler Construction (CC)*, 2011.
[67] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.
[68] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 13–24.
[69] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.
[70] A. Ayala, S. Tomov, M. Stoyanov, and J. Dongarra, "Scalability issues in fft computation," in *International Conference on Parallel Computing Technologies*, 2021.
[71] M. A. S. Bari, B. Chapman, A. Curtis, R. J. Harrison, E. Siegmann, N. A. Simakov, and M. D. Jones, "A64fx performance: experience on ookami," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021.
[72] N. Kitai, D. Takahashi, F. Franchetti, T. Katagiri, S. Ohshima, and T. Nagai, "An auto-tuning with adaptation of a64 scalable vector extension for spiral," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021.