Automatic Code Generation and Optimization of Large-scale Stencil Computation on Many-core Processors

Mingzhen Li[1,2], Yi Liu[2], Hailong Yang[1,2], Yongmin Hu[2], Qingxiao Sun[2], Bangduo Chen[2], Xin You[2], Xiaoyan Liu[2], Zhongzhi Luan[2] and Depei Qian[2]
State Key Laboratory of Software Development Environment, Beijing, China[1]
Beihang University, Beijing, China[2]
{lmzhhh,yi.liu,hailong.yang,varinic,qingxiaosun,chenbangduo,youxin2015,liuxiaoyan,07680,depeiq}@buaa.edu.cn

ABSTRACT

Stencil computation is an indispensable building block of many scientific applications and is widely used by the numerical solvers of partial differential equations (PDEs). Due to the complex computation patterns of different stencils and the various hardware targets (e.g., many-core processors), many domain-specific languages (DSLs) have been proposed to optimize stencil computation. However, existing stencil DSLs mostly focus on the performance optimizations on homogeneous many-core processors such as CPUs and GPUs, and fail to embrace emerging heterogeneous many-core processors such as Sunway. In addition, few of them can support expressing stencil with multiple time dependencies and optimizations from both spatial and temporal dimensions. Moreover, most stencil DSLs are unable to generate codes that can run efficiently in large scale, which limits their practical applicability. In this paper, we propose MSC, a new stencil DSL designed to express stencil computation in both spatial and temporal dimensions. It can generate high-performance stencil codes for large-scale execution on emerging many-core processors. Specially, we design several optimization primitives for improving parallelism and data locality, and a communication library for efficient halo exchange in large scale execution. The experiment results show that our MSC achieves better performance compared to the state-of-the-art stencil DSLs.

CCS CONCEPTS

• **General and reference** → *Performance*; • **Software and its engineering** → *Compilers*; **Domain specific languages**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

KEYWORDS

Stencil, Domain Specific Language, Performance Optimization, Many-core Architecture

ACM Reference Format:
Mingzhen Li[1,2], Yi Liu[2], Hailong Yang[1,2], Yongmin Hu[2], Qingxiao Sun[2], Bangduo Chen[2], Xin You[2], Xiaoyan Liu[2], Zhongzhi Luan[2] and Depei Qian[2]. 2021. Automatic Code Generation and Optimization of Large-scale Stencil Computation on Many-core Processors. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3472456.3473517

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3473517

1 INTRODUCTION

Stencil computation is an important and indispensable building block of modern scientific applications, and it has been widely used in the fields such as computational electromagnetics [35], computational fluid dynamics [9], climate modeling [2], and seismic processing [22]. Applications in these fields commonly rely on the numerical solvers of partial differential equations (PDEs). Since the PDEs are defined by continuous functions, the first step to solving them on a computer is discretization with finite difference methods (FDM), finite volume methods (FVM) or finite element methods (FEM). The discretization methods introduce stencil computations that dominate the performance of PDE solvers.

A stencil defines a particular computation pattern on the structural grid (or regular grid). As for spatial dimension, it updates each element on a grid using the values from a subset of the neighboring elements, where these elements can have different coefficients. As for temporal dimension, it updates the values of the grid at the current timestep based on the values from the previous timesteps. The temporal update can iterate over many timesteps until convergence. A stencil can be defined from many aspects, such as grid dimensions (e.g., 2D, 3D), shapes (e.g., box, star), number of neighbors (e.g., 7-point, 27-point). Particularly, the stencil computations with temporal dimension are also named iterative stencil loops (ISLs). In general, a stencil on a n-dimensional grid with timesteps can be represented as $(n + 1)$-dimensional nested loops, where the outermost loop iterates over the temporal dimension and the inner loops traverse all elements of the grid over the spatial dimension.

Stencil computation is usually memory bounded [31] and therefore can only achieve a limited fraction of the theoretical computational power on existing hardware. To address such limitation, tilling optimizations [4, 11, 16] have been found quite effective to mitigate the memory-bound constraint and become the widely studied techniques to accelerate the stencil computation. In general, a tiling optimization partitions the nested loops into smaller tiles according to certain tile shape, so that the stencil computation inside a tile can better exploit the data locality with improved parallelism. Existing tilling optimizations can be distinguished by their tile shapes, such as diamond tilling [4], trapezoid tilling [11], and overlapped tiling [16]. Besides, vectorization [20] has also been explored for optimizing stencil computation, which leverages the loop unrolling and data layout transformation to utilize better the SIMD units available on modern multicore/many-core processors.

However, due to the various patterns of stencil computations (e.g., different shape, data dimension, etc.), optimizing stencil computation by hand-tuning stencil codes tends to be tedious and error-prone. To address such drawback and alleviate the burden of optimizing various stencil patterns manually, many domain-specific languages (DSLs) / compilers [5, 7, 27, 30, 32, 40] have been proposed. For example, PLUTO [5] is a source-to-source compiler for C code that can leverage the polyhedral model to optimize the nested loops including stencils [14]. ARTEMIS [32] proposed a DSL and code generator targeting to accelerate complex stencils on GPU. In general, all stencil DSLs take the stencil definitions described by their programming languages as the input, and generate the optimized codes (or binaries) on hardware targets as the output. The transformation between stencil definitions and code generation incorporates optimization methods (e.g., tiling, vectorization), targeting the particular stencil patterns and hardware architectures.

To support hardware architectures other than CPU, existing stencil DSLs [7, 32, 40] can already generate optimized stencil codes on many-core processors such as KNLs and GPUs. However, the many-core processors are still under rapid development, various architecture designs have been proposed, such as Sunway processor [13] and Matrix processor [39]. These emerging many-core processors have unique architecture designs to manage computation parallelism, memory locality, and data transfer through customized programming models. However, how to embrace such many-core processors to generate efficient stencil codes is seldom addressed by existing stencil DSLs. Besides, except few (e.g., YASK [40], Physis, STELLA [18]), most stencil DSLs focus on optimizations on a single node and fail to scale across more nodes. Thus, they cannot handle large-scale stencil computations in real-world cases, such as simulations with fine-grained resolutions or large input domains. Moreover, when solving PDEs (e.g., second-order wave functions such as mechanical waves, electromagnetic waves, and gravitational waves), a sequence of stencil sweeps are usually performed. Thus a point can be updated by a subset of its neighbors in both space and time. However, most stencil DSLs can only support expressing the stencil computation in spatial dimension and do not consider the multiple time dependencies in temporal dimension.

To address the above limitations of existing stencil DSLs, we propose MSC, a new stencil DSL designed to express stencil computation in both spatial and temporal dimensions, and generate high-performance stencil codes for large-scale execution on emerging many-core processors such as Sunway and Matrix. Specifically, this paper makes the following contributions:

- We propose MSC, a new stencil DSL that *1)* decouples the stencil expression, computation optimization and code generation through layered design, *2)* addresses the multiple time dependencies by separating **Kernels** and **Stencils** (with multiple Kernels from different timesteps). Such design can easily adapt optimization passes tailored for many-core processors.
- We design an efficient communication library to work in synergy with our stencil DSL to generate large-scale stencil codes automatically. The library is optimized to support the halo exchange for large-scale stencil computation.
- We evaluate MSC with representative stencil benchmarks, and compare it with the state-of-the-art stencil DSLs. In addition, we provide both strong and weak scalability results to demonstrate the effectiveness in large-scale execution.

The rest of this paper is organized as follows. Section 2 describes the background and related work of the stencil optimizations as well as stencil DSLs. Section 3 and section 4 present the design overview and detailed implementations of our MSC stencil DSL. Section 5 presents the evaluation results and compares MSC with the state-of-the-art stencil DSLs. Section 6 concludes this paper.

2 BACKGROUND AND RELATED WORK

2.1 General stencil optimizations

The general stencil optimizations accelerate the stencil computations primarily by improving the parallelism and utilizing the data locality. Tilling is a well-established optimization for stencils, which explores parallelism and data locality. It has been well applied on both CPUs and GPUs. Hyper-rectangle tiling [34] is widely used in hand-tuned stencil implementations, which generally divides the input grids into hyper-rectangle tiles and calculates them in parallel. When adopting tiling optimization on the temporal dimension, the overlapped tilling [21] improves parallelism by redundant computation, while ensuring the dependency between tiles. The $3.5D$ tilling [29] also belongs to hyper-rectangle tiling, which tiles a $2D$ plane of the $3D$ spatial grid, streams in another dimension ($+0.5D$), and then tiles the temporal dimension ($+1D$). The $3.5D$ tilling is suitable for thread-level data parallelism and can easily scale with more computation units. In order to reduce the redundant computation, other tilling methods [4, 11, 26] have been proposed to apply different tiling shapes at the expense of limited parallelism. Besides, the split tilling [20] and hybrid tilling [14] adopt multiple tiling shapes to improve the parallelism further.

2.2 Emerging many-core processors

The Sunway SW26010 heterogeneous many-core processors (Sunway processor in short) have been adopted in the world-first hundred-PFLOPS supercomputer Sunway TaihuLight. The Sunway processor runs at 1.45GHz and offers 3.06TFlops peak performance in double precision. The architecture of Sunway processor is shown in Figure 1. It contains 4 core groups (CGs), where each CG consists of a management core (MPE) and 64 acceleration cores (CPEs). Specifically, each CPE contains no data cache but a 64 KB scratchpad memory (SPM), whose bandwidth and latency are similar to L1 cache. However, the usage of SPM requires explicit control by the program. Moreover, the CPEs can access the main memory through direct memory access (DMA) for continuous accesses. A customized parallel programming paradigm (*Athread*) is provided to allow programmers to manipulate the architecture features explicitly [13].

The Matrix MT2000+ many-core processors (Matrix processor in short) have been adopted in the next generation (prototype Tianhe-3) of Tianhe-2 supercomputer. The architecture of Matrix processor is shown in Figure 2. Each processor contains 128 compute cores running at 2.0GHz. Each core can deliver eight double precision flops per cycle, with an in-order 8-to-12-stage pipeline extended with vectorization. The compute cores are further organized into four Supernodes (SNs), connected through a scalable on-chip communication network. Each supernode has four panels, and each panel containing eight cache-coherent compute cores. The entire

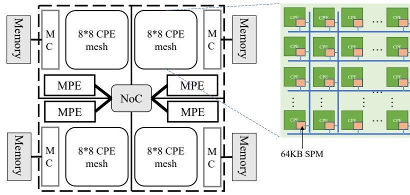

Figure 1: The architecture of Sunway processor.

processor delivers around 2.048TFlops double-precision peak performance. The processor supports eight DDR4-2400 channels and is integrated with the PCIe 3.0 interface.

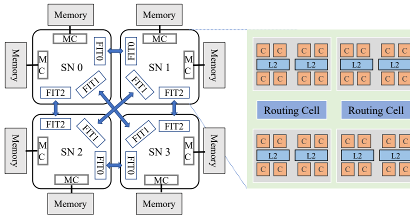

Figure 2: The architecture of Matrix processor.

2.3 Stencil optimizations on emerging many-core processors

There are a few works optimizing stencil computation on Sunway many-core processor. For example, two large-scale applications [12, 38] on Sunway TaihuLight supercomputer that won the Gordon Bell Prizes optimized *3d13pt* stencil computation. The atmospheric dynamics application [38] adopted the 2.5*D* tilling and performed corresponding optimizations across computation and memory hierarchies. The earthquake simulation application [12] adopted on-the-fly data compression, the on-chip halo exchange, as well as coalesced DMA access to mitigate the memory bound constraint. Other works [2, 22] have also focused on memory-related optimizations of stencil computation. Although not directly related, the highly optimized computation kernels, such as convolution [10], and Cholesky factorization [24], have educated us on the necessary techniques to implement efficient stencil DSL on Sunway. And Matrix MT2000+ many-core processor lacks corresponding works on optimizing stencil computation lack. However, since it is pretty similar to many-core CPUs, existing CPU optimizations such as cache-oblivious tilling [11] can be adopted to implement high-performance stencil DSL on Matrix processor.

2.4 Stencil DSLs

To address the diverse stencil patterns and to provide better performance portability across different processors, many stencil DSLs have been proposed with their uniqueness [7, 18, 27, 30, 32, 40]. Specifically, YASK generates vectorized stencil codes using the vector folding method [40], which packs multi-dimension data into one SIMD vector for better vectorization. Physis [27] focuses on large-scale stencil computations on GPU based cluster. It optimizes the computation kernel and provides an MPI runtime for halo data exchange, enabling the overlapping of communication and computation. STELLA [18] focuses on stencils with multiple stages in PDEs, and it supports updating the halo data through boundary conditions or its halo-exchanging library (GCL). Halide [30] is designed to optimize stencil computations in image processing. The fundamental idea of Halide is to decouple the computation definition from its implementation (a.k.a., schedule). The above idea enables large optimization space for stencil computation, where auto-tuning methods [1] can be applied to select a near-optimal schedule with various searching techniques automatically. The comparison between MSC and existing stencil DSLs is briefly summarized in Table 1.

Although significant efforts have been devoted to stencil DSLs, we notice that the existing works miss support for emerging many-core processors such as Sunway and Matrix. In addition, most DSLs focus on expressing and optimizing stencil computations on the spatial dimension, and fail to support stencil expression with multiple time dependencies. Theoretically, these DSLs (e.g., Halide) can also support such stencils by writing extra glue codes. For example, domain experts should manually duplicate the generated stencil kernels, reserve the memory space to store the intermediate results, and finally derive the output grid of a timestep by assembling these intermediate results, which are less friendly and efficient.

Moreover, few stencil DSLs (except YASK, Physis, and STELLA) can optimize stencil computations at large scale, which constrains the practical usage of existing DSLs. The above observations motivate us to propose a new stencil DSL that supports expressing stencil computation on both spatial and temporal dimensions, and allows easy adaption to emerging many-core processors and flexible integration of large-scale communication optimizations.

3 DESIGN OVERVIEW

The MSC stencil DSL primarily contains three parts: the frontend, the backend, and the communication library, as shown in Figure 3.

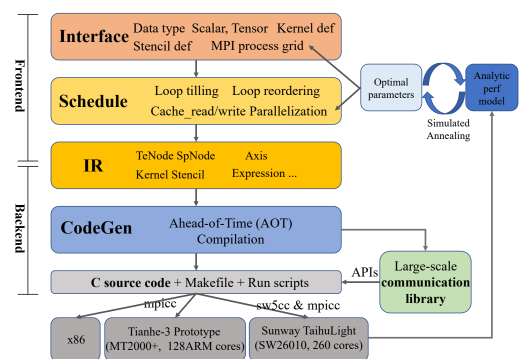

Figure 3: Design overview of MSC stencil DSL.

The frontend takes the definition of a stencil computation as input. Specifically, the definition is expressed with the MSC's programming language, consisting of the stencil pattern, time iteration, grid domain, and optimization primitives. The optimization primitives further include loop tilling, loop reordering, parallelization,

Table 1: The comparison between MSC and existing stencil DSLs.

		MSC	Halide [30]	Pluto [5]	Tiramisu [3]	Patus [7]	Artemis [32]	YASK [40]	STELLA [18]	Physis [27]	OPS [33]	Devito [25]	Lift [19]	AN5D [28]	Polly [15]	Pochoir [36]	Loo.py [23]
Stencil	Single timestep	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Multiple timestep	✓					✓		✓			✓			✓		✓
Hardware	CPU	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓
	GPU		✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓
	Manycore	✓															
Optimization	Spatial tiling	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
	Streaming						✓				✓			✓			
	Temporal tiling			✓	✓						✓			✓	✓	✓	
	Auto-tuning	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	
Distributed	Halo exchange	✓	✓ [8]		✓			✓	✓	✓	✓	✓					
	Pluggable library	✓	✓ [8]														

as well as data access control across memory hierarchy. The intermediate representation (IR) connects the frontend and the backend. Generally, IR is an abstraction of the stencil computation and is independent of the hardware details, which decouples the stencil definition from detailed optimizations.

The backend takes the IR after transformations and optimizations as input, and generates optimized codes for different hardware targets. Considering the emerging many-core processors (e.g., Sunway) that do not support Just-In-Time (JIT) compilation due to the runtime overhead, MSC provides the Ahead-Of-Time (AOT) compilation to generate standard C codes as well as corresponding building scripts (e.g., Makefiles). These C codes can then be compiled by the native compilers available on the processors. The AOT compilation in MSC's code generator enables easy adaption to other many-core processors.

The communication library handles the data exchange of halo regions when generating large-scale stencil codes running across multiple nodes. We implement the communication optimizations as a library to ensure a clear separation of stencil kernel optimizations and large-scale communication optimizations. Specifically, the compilation of the MSC DSL identifies the size and location of the halo regions that need to be exchanged during the computation. Then, it invokes the corresponding APIs in the communication library to realize halo exchange during code generation. In addition, the computation codes are interleaved with the communication codes to optimize the performance further.

4 METHODOLOGY AND IMPLEMENTATION

4.1 Intermediate Representation

MSC adopts a single level IR that is embedded in the abstract syntax tree. The IR of MSC mainly contains the tensor, nested loop and expression IRs, as shown in Table 2. The tensor IR includes ***SpNode*** and ***TeNode***, which represents 1D/2D/3D tensor and record the dimensions (ndim), shape of each dimension (shape), datatype (dt). The ***SpNode*** also records halo size of each dimension (halo). The nested loop IR is represented by ***Axis***, and an *Axis* contains its id (id_var), order in nested loops (order), start/end position and stride. The expression IR is represented by ***Expressions***, which contains several types including value assignment (*AssignOpExpr*), unary / binary math operator (*OperatorExpr*), external function call (*CallFuncExpr*, tailored for halo-exchanging library), and index calculation (*IndexExpr*).

The ***Kernel*** defines the basic stencil kernel (e.g., 3D Laplacian operator), and is composed of *Tensor*, *Nested loop*, and *Expression* IRs. And ***Stencil*** defines the stencil with multiple time dependencies, and is composed of *Kernels*, *Tensor* and *Expression* IR. Besides, the optimization primitives can rewrite the *Axis* and *Expression* IR in *Kernel* for better code generation.

Table 2: The descriptions of IR nodes in MSC.

Type	Nodes	Description
Tensor	- SpNode - TeNode	Tensor w/i halo region Tensor w/o halo region
Nested loop	- Axis	Axis of nested loops
Expression	- AssignExpr - OperatorExpr - CallFuncExpr - IndexExpr	Value assignment expr. Unary / Binary expr. External function call expr. Index calculation expr.
Kernel	-	Basic stencil kernel
Stencil	-	Stencil with multiple time dependencies
Primitive	- tile, reorder, parallel - cache_read/write - compute_at	Optimization passes which rewrite the IR.

4.2 Domain specific language definition

The abstraction of a DSL realizes a tradeoff between expressibility and efficiency. MSC achieves good expressibility by enabling users to define complex stencils with arbitrary shapes easily, in addition to standard stencils (e.g., 3d7pt). Moreover, MSC achieves good efficiency by performing effective optimizations. MSC supports stencils with multiple time dependencies and scales to multiple MPI nodes, where the user only needs to write few lines of DSL codes compared to C-code implementations.

To better illustrate the programming language supported in MSC, Listing 1 presents the code implementation of a *3d7pt* stencil from HPGMG using MSC. MSC is implemented as an extension to standard C++ with its unique syntax supported. The variables M, N, P describe the dimension (256^3) of the input/output grid in Line 1. The halo region's width in each spatial dimension and the size of the time window in temporal dimension are defined in Line 3-4, respectively. The time window is determined by the maximum timesteps that the stencil depends on to update its current value.

Listing 1: MSC implementation of *3d7pt* stencil

```
1  ...
2  const int M = N = P = 256;
3  const int halo_width = 1;
4  const int time_window_size = 2;
5  DefVar(k, i32); DefVar(j, i32); DefVar(i, i32);
6  DefTensor3D_TimeWin(B, time_window_size, halo_width, f64, 256, 256, 256);
7  Kernel S_3d7pt((k,j,i), c0*B[k,j,i] + c1*B[k,j,i−1] + c2*B[k,j,i+1] + c3*B[k−1,j,i] + c4*
       B[k+1,j,i] + c5*B[k,j−1,i] + c6*B[k,j+1,i], schedule);
8  // Optimizations
```

```
9   ... Several optimization primitives
10  auto t = Stencil :: t;
11  Result Res( (i,j) , B[i,j] );
12  Stencil st((i,j), Res[t] << S_3d7pt[t−1] + S_3d7pt[t−2]);
13  DefShapeMPI3D(shape_mpi, 4, 4, 4)
14  st.input(shape_mpi, B, "../data/rand.data");
15  st.run(1,10);
16  st.compile_to_source_code("3d7pt");
```

MSC supports data types including 32-bit integer (i32), 32-bit float (f32) and 64-bit float (f64). The variables k, j, i represents the subscripts of the elements in the grid, as shown in Line 5. They are defined through the DefVar(α, i32) function, which specifies a scalar α in 32-bit integer datatype.

The input 3D tensor B in f64 datatype is defined in Line 8, with shape of $256 \times 256 \times 256$. The halo width of input tensor B is set to *halo_width*. In MSC, there are two kinds of tensors, *SpNode* and *TeNode*. The *SpNode* can be explicitly defined by users through the *DefTensor2D/DefTensor3D* functions. Then, the MSC automatically allocates the extra memory space for *SpNode* to store the halo regions and the intermediate data within the time window. The *TeNode* is used by the MSC compiler and is transparent to users. It acts as a temporary buffer to store the intermediate data of the computation domain per timestep without halo region.

The *3d7pt* stencil kernel is defined in Line 7 through the *Kernel* function, which updates the element (k, j, i) with six neighboring elements $(k \pm 1, j, i)$, $(k, j \pm 1, i)$, and $(k, j, i \pm 1)$. The kernel traverses all grid elements and handles the halo regions automatically. After the kernel definition, the users can call the kernel to specify the stencil computation further. MSC provides various optimization primitives to optimize the stencil computation in Line 9, which are illustrated in Section 4.3.

After the optimizations, the stencil computation along the time dimension *st* is defined in Line 12, which aggregates the output of the *3d7pt* kernel at timestep $(t-1)$ and $(t-2)$. The MPI grid for large-scale execution is defined in Line 13 with a size of (4×4×4). After specifying the input data and the time iterations in Line 14 and 15, respectively, MSC generates the optimized codes of *3d7pt* stencil through compilation in Line 16.

4.3 Compilation optimizations

To achieve high performance on emerging many-core processors, we implement the optimization primitives including loop tilling (*tile*), loop reordering (*reorder*) and parallelization (*parallel*) to optimize the nested loops within stencil codes. Additionally, to support the cache-less processors such as Sunway, we provide *cache_read*, *cache_write* and *compute_at* primitives to manage the local memory and the DMA data transfer. As shown in Figure 4, the MSC walks through the compilation optimizations of (a)~(c) on Matrix processor, whereas for Sunway processor it walks through (a), (b), (d) and (e). Listing 2 presents the *3d7pt* stencil code generated by MSC on Sunway processor that is optimized by the above primitives. We explain each of the primitives as follows.

Listing 2: Optimizations of *3d7pt* stencil with MSC on Sunway processor

```
1   // Kernel defination
2   DefTensor3D_TimeWin(B, time_window_size, halo_width, f64, 256, 256, 256);
3   Kernel S_3d7pt((k,j,i), c0*B[k,j,i] + c1*B[k,j,i−1] + c2*B[k,j,i+1] + c3*B[k−1,j,i] + c4*
        B[k+1,j,i] + c5*B[k,j−1,i] + c6*B[k,j+1,i], schedule);
4   // Optimizations
5   const int tile_size_x = 8, tile_size_y = 8, tile_size_z = 32;
6   Axis xo, yo, zo, xi, yi, zi;
7   CacheRead buffer_read;
8   CacheWrite buffer_write;
9   S_3d7pt.tile(tile_size_x, tile_size_y, tile_size_z, xo, xi, yo, yi, zo, zi);
10  S_3d7pt.reorder(xo, yo, zo, xi, yi, zi);
11  S_3d7pt.cache_read(B, buffer_read, "global");
12  S_3d7pt.cache_write(buffer_write, "global");
13  S_3d7pt.compute_at(buffer_read, zo);
14  S_3d7pt.compute_at(buffer_write, zo);
15  S_3d7pt.parallel(xo, 64);
16  S_3d7pt.build("sunway");
```

Tile primitive - Loop fission [37] is one of the most common loop optimizations, which splits one loop into two nested loops denoted as *outer* and *inner*. The size of the inner loop is specified by fission factor τ, and the size of outer loop is derived as N/τ, where N is the original loop size. In MSC, we implement the loop fission optimization as the *tile(*τ*,* ax_{outer}*,* ax_{inner} *)* primitive, where τ is the given fission factor and ax_{outer}, ax_{inner} are the split axises. As shown in Figure 4(a), after kernel definition of *3d7pt* stencil, MSC allocates three axises x, y, and z to represent three nested loops i, j and k, respectively. Then we define the fission factor $\tau_x = 8$, $\tau_y = 8$, $\tau_z = 32$ along x, y, and z axises and corresponding six axises xo, xi, yo, yi, zo, and zi. Then, we utilize the *tile* primitive as shown in Figure 4(a) to optimize the nested loops of the stencil code. Note that the order of argument lists of fission factor and split axises are fixed. As shown in Figure 4(b), the *tile* primitive splits the original three nested loops into six nested loops, denoted as $i_outer \in [0, 32)$, $i_inner \in [0, 8)$ split from $i \in [0, 256)$, $j_outer \in [0, 32)$, $j_inner \in [0, 8)$ split from $j \in [0, 256)$, and $k_outer \in [0, 8)$, $k_inner \in [0, 32)$ split from $k \in [0, 256)$. Note that the size of inner loops i_inner, j_inner, and k_inner equals to the tile size τ_x, τ_y and τ_z, respectively.

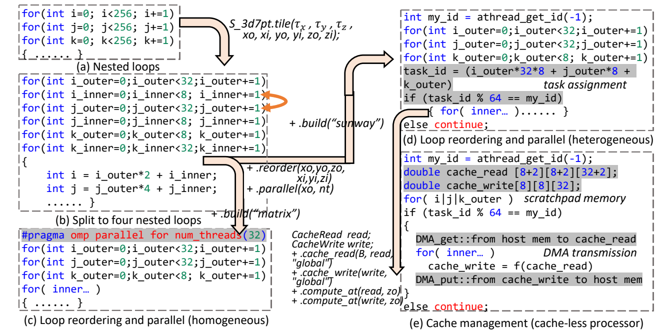

Figure 4: Illustration of *tile*, *reorder* and *parallel* primitives, where (a) original stencil kernel of three nested loops, (b) six nested loops split by *tile* primitive, (c) loops after *reorder* primitive and multi-threading *parallel* primitive on a homogeneous many-core processor, (d) loops after *reorder* primitive and multi-threading *parallel* primitive on a heterogeneous many-core processor, and (e) data caching management through *cache_read|write* and *compute_at* primitives on a cache-less processor.

Reorder primitive - After loop fission with *tile* primitive, the order of data access is not changed. Therefore, we introduce *reorder(<a list of axises>)* primitive to reorder the loops for better data access locality. The *reorder* primitive reorders the data access of the nested loops according to the order of the given list of

axises. As shown in Figure 4(b), the loops will achieve better data locality when the loops are reordered into the data access order of *xo* (denoted as i_outer), *yo* (j_outer), *zo* (k_outer), *xi* (i_inner), *yi* (j_inner), *zi* (k_inner). The reordered nested loops using *reorder(xo, yo, zo, xi, yi, zi)* are shown in Figure 4(c) and (d). The combination of *reorder* and *tile* primitives can split the stencil computation into a sequence of computation tasks on tiles. The tiles are assigned with overlapped halo regions to avoid computation dependencies. Therefore, each task can be calculated independently at per core basis of the many-core processors (e.g., CPEs of Sunway processor, and compute cores of Matrix processor).

Parallel primitive - The multi-threading parallelization is the common optimization for stencil computation adopted on many-core processors. Thus, MSC also provides *parallel(ax,* $N_{threads}$*)* primitive, where *ax* denotes the outer-most loop axis for parallelization and $N_{threads}$ indicates the thread number specified for parallelization. As shown in Figure 4(c), the *parallel* primitive optimizes the outer-most loop *xo* (denoted as i_outer) with multi-threading of 64 threads. Due to the implementation difference of multi-threading on emerging many-core processors, we currently provide two implementations of *parallel* primitive. For homogeneous many-core processors (e.g., Matrix), we choose OpenMP programming model for multi-threading. Specifically, we add *#pragma omp parallel* directives, as shown in Figure 4(c). For heterogeneous many-core processors (e.g., Sunway), we use the native multi-threading paradigms such as *athread* to enable parallelization. As shown in Figure 4(d), each CPE of Sunway processor has a unique my_id, and each computation task has its $task_id$ derived through the loop counters. Then the tasks whose $task_id$ satisfies $\mod(task_id, 64) == task_id$ are assigned to CPE my_id. With parallel primitive, the computation tasks generated by *tile* and *reorder* primitives can be mapped to the massive cores of the many-core processors conveniently.

Caching related primitives - For emerging many-core processors that adopt the cache-less architecture such as Sunway, the data access requires explicit control to utilize the local memory (usually implemented as scratchpad memory, SPM) for data reuse. To utilize the SPM for better data locality, we provide three primitives in MSC. The *CacheRead* and *CacheWrite* primitives define the read/write buffers allocated in SPM. The two primitives have two clauses *cache_read* and *cache_write* to bind the input/output tensor to the read/write buffer, and register these bindings to MSC. To utilize the direct memory access (DMA) mechanism available on the processors, we provide the *compute_at* primitive. This primitive dictates the DMA data transfer, which contains two parts: *1)* the data to be transferred, and *2)* the code position to invoke DMA. The MSC can perform the DMA transfer to *get/put* the required data from/to off-chip memory. Take Figure 4(e) for example, the *cache_read* primitive binds the input tensor *B* (*SpNode*) with the read buffer, *cache_write* primitive binds the temporary tensor (*TeNode*) with the write buffer, and the parameter *global* indicates the allocated buffers are in the scope of all loops to avoid frequent *malloc* and $free$ memory operations. The two *compute_at* primitives inform MSC to transfer the read/write buffers at the beginning/end of the *zo* (denoted as k_outer) loop respectively. Therefore, the caching related primitives tailored for cache-less many-core processors (e.g., Sunway) can control the allocation of local memory (e.g., SPM) for better data reuse, and manage the DMA transfer between local memory and main memory automatically.

Sliding time window - Due to the limited memory space, it is impossible to store all the intermediate data of a stencil kernel at each timestep, especially iterating over a large number of timesteps. In order to reduce the memory occupancy, we introduce the *sliding time window* to store the intermediate results from previous timesteps. As shown in Figure 5(a), we use a stencil on 2D grid to illustrate this optimization. The stencil computation at time t depends on the output tensors from time $(t-1)$ and $(t-2)$. Therefore, the width of the sliding time window is set to three. At the time t, since the output tensors from time $(t-1)$ and $(t-2)$ are within the sliding time window, they have been stored in the memory buffer for stencil computation at time t. At time $(t+1)$, the output tensor at time t replaces the output tensor at time $(t-2)$ in the memory buffer. In this way, the number of intermediate tensors kept in memory is restricted to three (Figure 5(c)), which avoids overwhelming the memory by continuously increasing the memory occupancy (Figure 5(b)).

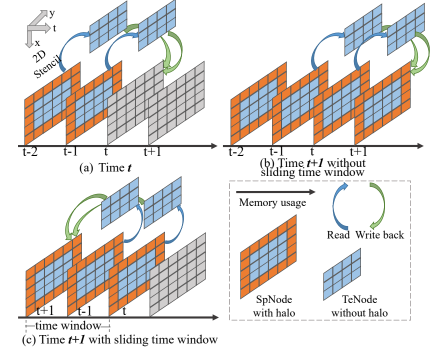

Figure 5: Sliding time window optimization along time.

In sum, the loop related primitives (*tile*, *reorder* and *parallel*) improve the processor utilization with better parallelism, and the caching related primitives improve the data locality during memory access. In addition, the sliding time window optimization mitigates the memory footprint during the computation. Together, MSC can effectively leverage the architecture features of many-core processors, and thus generate optimized stencil codes.

4.4 Communication Library

We design an efficient MPI communication library tailored for large-scale stencils, which works in synergy with MSC to generate stencil codes automatically. This library considers the stencil patterns and ensures computation correctness. Besides, it is asynchronous and topology-aware. The users only need to specify the size of the MPI grid (Line 15 of Listing 1) without concerning about the communication details across multiple nodes. And the MSC invokes the halo exchanging APIs provided by the library at the correct locations of the generated stencil codes. Generally, the communication library contains three parts: *1)* domain decomposition, *2)* halo exchanging, and *3)* performance auto-tuning. In Figure 6, we choose a *2d7pt* stencil on a 8×8 input grid with a 2×2 MPI grid to illustrate the implementation of the communication library in detail.

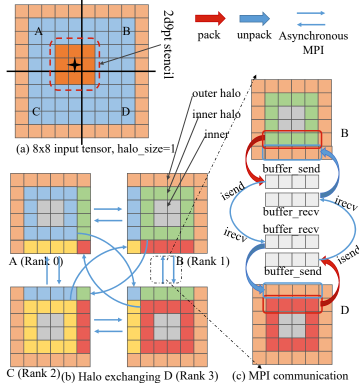

Figure 6: The implementation of the communication library, where (a) the domain decomposition and task assignment to MPI processes, (b) the halo exchange between MPI processes, and (c) the detailed MPI communication.

Domain decomposition - The input tensor is decomposed evenly among the number of MPI processes, and then each sub-tensor is assigned to an MPI process. In Figure 6(a) and (b), the sub-tensors A, B, C, D are assigned to MPI processes with rank 0, 1, 2, 3 respectively. Each sub-tensor also contains a halo region similar to the input tensor. Besides, we further dissect the sub-tensor into three parts: *1)* outer halo region (orange color), which holds the data to be received from neighboring sub-tensors, *2)* inner halo region (blue/green/yellow/red color), which holds the data to be sent to neighboring sub-tensors, and *3)* inner region (gray color), which does not participate in halo exchange. Notably, both the inner halo region and the inner region contain the valid data of a sub-tensor. As shown in Figure 6(b), we plot the inner halo regions of sub-tensors A, B, C, D with different colors to illustrate where the data in the outer halo regions comes from.

Halo exchange - As for halo exchange, a MPI process (rank 1) allocates the memory for the send buffer and the receive buffer, then packs the data of the inner halo region in the send buffer. After that, it calls *MPI_isend* function to send the packed data to the neighboring MPI process (rank 3) asynchronously. The neighboring MPI process (rank 3) calls *MPI_irecv* function to receive the packed data in the receive buffer, and then unpacks the data to update the outer halo region. Meanwhile, the neighboring process (rank 3) is also sending its inner halo region to its neighboring process (rank 1), as shown in Figure 6(c). Notably, all MPI processes are exchanging the halo region asynchronously (Figure 6(b)), which better utilizes the network bandwidth and reduces the communication overhead. After the halo exchange, all MPI processes have received the required data in the outer halo region for stencil computation.

Performance auto-tuning - To achieve optimal stencil codes, we need to adjust certain parameters to adapt to the input domains and the underlying hardwares. These parameters include the sub-domain size with domain decomposition, and the tile size with *tile* primitive. The settings of the above parameters could have a significant impact on performance. To alleviate the burden of parameter tuning by hand, we adopt the auto-tuning methods to determine the optimal parameter settings. Specifically, we build an analytical performance model using multivariable linear regression to predict the stencil kernel time. Based on the performance model, we adopt the simulated annealing algorithm to search for the optimal parameter settings for large-scale stencil codes. The searching algorithm considers the MPI initialization time, kernel computation time, data packing/unpacking time, and data transferring time. Due to the page constraint, we omit the detailed formulations of the performance model and searching algorithm.

Since the communication library works as a plugin to MSC, it is naturally separated from the stencil kernel optimizations. Therefore, various communication optimizations can be further implemented in this library without modifying MSC. Such advantage overcomes the drawback of tightly coupling stencil kernel optimizations with communication methods adopted in existing DSLs such as YASK [40] and Physis [27]. Therefore, users can easily plug in their own halo-exchanging libraries (e.g., GCL in STELLA) and seamlessly integrate with code generation of MSC. It also enables easy adaption to supercomputers or large clusters installed with exotic network topologies, requiring customization of the underlying communication APIs.

5 EVALUATION

5.1 Experimental setup

Our experiments are conducted on the three platforms, including Sunway TaihuLight supercomputer, the Prototype Tianhe-3 cluster, and the local CPU server. The detailed configurations of these platforms are listed in Table 3. Note that in the prototype cluster, the core resources assigned to the user are at the granularity of 32 cores (one SN) with other cores reserved [39].

Table 3: The hardware and software config.

Platform	Processor	Compiler	MPI	OpenMP
Sunway TaihuLight	SW26010 (65 cores*4)	gcc-8.3 sw5cc	mpich-3.0	None
Tianhe-3 Prototype	MT2000+ (32 cores)	gcc-8.2	mpich-3.2	4.5
Local CPU Server	E5-2680v4*2 (14 cores*2)	gcc-8.3	openmpi-3.1	4.5

We first evaluate the performance of MSC on a single many-core processor, including Sunway and Matrix. We then evaluate the scalability of MSC on multiple processor nodes on Sunway platform and Tianhe-3 platform, respectively. To compare the MSC with the state-of-the-art stencil DSLs such as Halide, Patus, and Physis, we provide the evaluation results on the CPU platform for single node runs. As listed in Table 4, we evaluate a set of stencil benchmarks with different shapes, input dimensions, and computation orders, which are representative of a wide range of scientific applications.

For performance evaluation on a single processor, we compare the codes generated by the MSC with the codes manually optimized using OpenACC (on Sunway) and OpenMP (on Matrix). The OpenACC and OpenMP baselines adopt the same optimizations as MSC for a fair comparison. For scalability evaluation, we measure the strong and weak scalability of the MSC on Sunway and Tianhe-3 platforms, respectively. Moreover, we evaluate the effectiveness of the auto-tuning method in MSC for determining the optimal parameter settings. Each implementation has been executed ten

times, with the average execution time reported. To ensure the correctness of MSC, we measure the relative errors between the generated codes and the serial codes. For all evaluation results, the relative errors of the single-precision (fp32) results and the double-precision (fp64) are less than 10^{-5} and 10^{-10}, respectively, which indicates our MSC does not affect the correctness [17].

Table 4: Stencil benchmarks used in the evaluation.

Benchmark	Read(Byte)	Write(Byte)	Ops(+-×...)	Time Dep.
2d9pt_star	72	8	17	2
2d9pt_box	72	8	17	2
2d121pt_box	968	8	231	2
2d169pt_box	1352	8	325	2
3d7pt_star	56	8	13	2
3d13pt_star	104	8	17	2
3d25pt_star	200	8	41	2
3d31pt_star	248	8	50	2

5.2 Performance comparison on a single many-core processor

On both Sunway and Tianhe-3 platforms, we execute the MSC on the login nodes for compilation and code generation, and evaluate the generated codes on their computation nodes. We use the same grid size (256^3) of 3D stencils in our experiments as Physis [27]. And for 2D stencils, we set the total number of points in the 2D grids to be equal to that of 3D grids ($4096^2 = 256^3$). Table 5 present the parameter settings of MSC across the benchmarks.

Table 5: The parameter settings of 2D/3D stencils using MSC on a single Sunway (a CG) / Matrix (32 cores) processor.

Stencil	Grid Size	Tile Size	Reorder Rule
2d9pt_star 2d9pt_box	$4,096^2$	(32,64) / (2,2048)	(xo,yo,xi,yi)
2d121pt_box 2d169pt_box	$4,096^2$	(16,32) / (2,2048)	(xo,yo,xi,yi)
3d7pt_star 3d13pt_star	256^3	(2,8,64) / (2,8,256)	(xo,yo,zo,xi,yi,zi)
3d25pt_star 3d31pt_star	256^3	(2,4,32) / (2,8,256)	(xo,yo,zo,xi,yi,zi)

5.2.1 Performance speedup. On Sunway, the users can leverage the directives (*#pragma acc ...*) provided by the OpenACC compiler to optimize stencil codes. And we select the directives of data caching (*acc copyin/copyout*), loop splitting (*acc tile*), and multi-threading (*acc parallel*) to accelerate the stencil codes, and use codes optimized by OpenACC as the baseline. Since the OpenACC focuses on program optimization on a single CG, we conduct experiments on a single CG for performance comparison (1 MPE + 64 CPEs). Figure 7 presents the execution time comparison of MSC and OpenACC codes to perform stencil computations under fp64 and fp32 precisions. It can be seen that MSC outperforms OpenACC in all cases, with the average speedup of 24.4× (fp64) and 20.7× (fp32). Although the OpenACC implementations adopt similar optimization techniques as MSC, they lack the fined-grained managements that adapt the stencil patterns to the Sunway architecture, especially on high-order stencils (e.g., *2d121pt_box* and *2d169pt_box*). Whereas the MSC can leverage the various optimization primitives to generate optimized codes exploiting the architectural features such as SPM and DMA for superior performance. Taking the *3d13pt_star* stencil for example, all 64 CPEs inside a CG have been fully utilized, with each CPE calculating 256 tiles. Through DMA, each tile has been loaded from main memory to SPM, and written back to main memory when updated. During the computation, the CPEs can access all required data resided in SPM without accessing the main memory. Specifically, the SPM utilization has reached more than 78%, with each data point reused about 13 times.

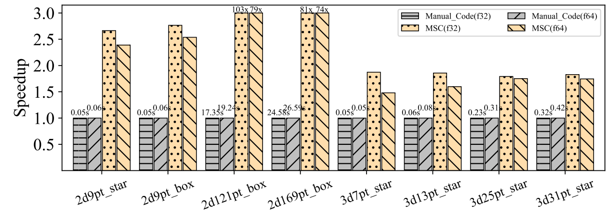

Figure 7: Performance comparison between MSC generated codes and OpenACC manually optimized codes on a Sunway CG, where OpenACC is set as the baseline.

We also compare the performance of MSC with manually optimized OpenMP codes on a single Matrix processor. Figure 8 presents the performance comparison of MSC and OpenMP codes to perform stencil computations under fp64 and fp32 precisions. The performance of MSC generated stencil codes is close to the manually optimized OpenMP codes. This is because the Matrix processor is an ARM-based homogeneous many-core processor, which is easier to optimize codes manually than the Sunway processor. In addition, the OpenMP programming paradigm provides efficient *pragmas* to manipulate the parallelism and data access. Specifically, MSC achieves 1.05× (fp64) and 1.03× (fp32) performance of the manually optimized codes on average. Note that although MSC achieves similar performance with manually optimized OpenMP codes, the productivity using MSC to write stencil codes is better than using OpenMP (byLoC in Section 5.2.3).

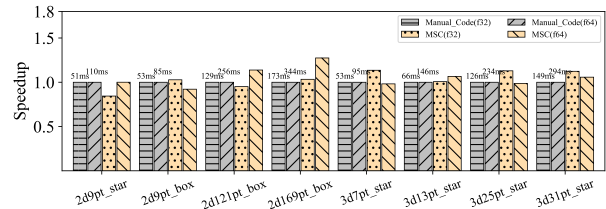

Figure 8: Performance comparison between MSC generated codes and OpenMP manually optimized codes on a Matrix processor, where OpenMP is set as the baseline.

5.2.2 Roofline model analysis. We use the roofline model analysis to better understand the performance of stencil codes generated by MSC. Due to the similar computation patterns under both fp64 and fp32 precision, we only provide the roofline model analysis under fp64 precision on both Sunway and Matrix processors. As

shown in Figure 9(a) and Figure 9(b), most stencil benchmarks are on the left of the ridge point, which means their performance is memory-bound. Whereas for *2d169pt* stencil on Sunway, it is compute-bound due to the large number of calculations required for applying the stencil computation. However, due to the limited bandwidth on Matrix processor, the *2d169pt* stencil is still memory-bound. The roofline results indicate the future direction of our MSC optimizations for generating high-performance stencil codes.

To further analyze the roofline results, we classify the stencil benchmarks into three categories based on their operational intensity on Sunway processor, including *1) 2d9pt_star*, *2d9pt_box*, *3d7pt_star*, *3d13pt_star*, *2) 3d25pt_star*, *3d31pt_star*, and *3) 2d121pt_box*, *2d169pt_box*. The stencil benchmarks in *category 3)* achieves better performance compared to other categories due to their intensive computation patterns, which densely access the neighboring elements in compact regions and thus improve the data locality. The above reason also applies to the observation that *2d9pt_box* outperforms *2d9pt_star* in *category 1)*. We also notice that the stencil benchmarks in *category 2)* have lower performance even than stencil benchmarks with lower operational intensity in *category 1)*. This is because the star stencils with more points have much larger neighboring regions, which leads to more discrete (input grid) and redundant (halo region) data accesses, and thus poorer data locality. Similar analysis can also be applied on Matrix processor.

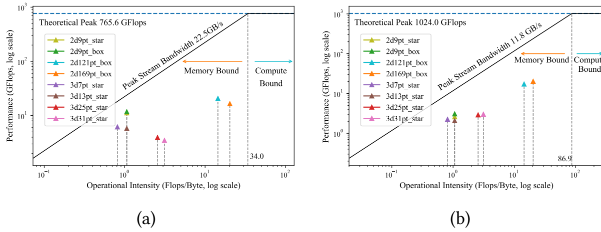

Figure 9: Roofline analysis of all stencil benchmarks on a CG of Sunway processor (a) / a Matrix processor (b).

5.2.3 LoC comparison. To measure the productivity of MSC, we compare the line of codes (LoC) between MSC DSL and manually optimized OpenACC/OpenMP codes on Sunway and Matrix processors, respectively. As shown in Table 6, the LoCs using MSC are less than that of the manually optimized codes, especially on Matrix processor. The average reduction of LoC is 27% and 74% on Sunway and Matrix processors, respectively. On Sunway processor, the LoCs using MSC are close to that of OpenACC. This is because the OpenACC provides limited primitives for optimizing stencil codes on Sunway. Therefore, the length of OpenACC code is usually quite constrained. On the contrary, the OpenMP offers a large number of *pragmas* for performance tuning. Therefore, it costs the users significant time for code optimizations with increasing code length. In sum, the results indicate that MSC can significantly reduce the programming efforts to write high-performance stencil codes.

5.3 Scalability

We evaluate the strong and weak scalability of MSC on Sunway TaihuLight supercomputer and prototype Tianhe-3 cluster. The

Table 6: LoC comparison between MSC and manually optimized codes on Sunway and Matrix processor.

Benchmark	Sunway		Matrix	
	MSC	OpenACC	MSC	OpenMP
2d9pt_star	33	45	27	95
2d9pt_box	32	45	26	95
2d121pt_box	50	55	44	207
2d169pt_box	54	57	48	255
3d7pt_star	36	45	28	101
3d13pt_star	33	51	27	98
3d25pt_star	35	65	29	102
3d31pt_star	37	72	31	103

configuration of the strong/weak scalability experiments on the two platforms is listed in Table 7. On Sunway platform, we scale the execution of MSC generated codes from 128 to 1,024 CGs. Since each CG contains 65 cores, the number of cores used in the stencil computation ranges from 8,320 to 66,560. On Tianhe-3 platform, we scale the execution from 32 to 256 processors. Since each Matrix processor contains 32 cores, the number of cores used in the stencil computation ranges from 1,024 to 8,192. Figure 10(a) and (b) presents the strong and weak scalability of MSC on Sunway platform and Tianhe-3 platform, respectively. The y-axis represents the absolute measured performance (GFlops), and the x-axis represents the number of computation cores. Note that on the x-axis, the number on the left and right of the separator line (e.g., 1,024|256) represents the number of computation cores on Sunway (e.g., 1,024) and Tianhe-3 (e.g., 256) platform, respectively. The dash and solid line represent the ideal and achieved performance.

Table 7: Configuration of the strong/weak scalability experiments of MSC on Sunway TaihuLight supercomputer (Left|) and prototype Tianhe-3 cluster (|Right).

Dim	Weak Scalability	Strong Scalability	MPI Grid	Processes
	Sub_grid per MPI	Sub_grid per MPI		
2D	$4,096^2$	$4,096 \times 4,096$	$16 \times 8 \mid 8 \times 4$	128 | 32
	$4,096^2$	$4,096 \times 2,048$	$16 \times 16 \mid 8 \times 8$	256 | 64
	$4,096^2$	$2,048 \times 2,048$	$32 \times 16 \mid 16 \times 8$	512 | 128
	$4,096^2$	$2,048 \times 1,024$	$32 \times 32 \mid 16 \times 16$	1024 | 256
3D	256^3	$256 \times 256 \times 256$	$8 \times 4 \times 4 \mid 4 \times 4 \times 2$	128 | 32
	256^3	$256 \times 256 \times 128$	$8 \times 8 \times 4 \mid 4 \times 4 \times 4$	256 | 64
	256^3	$256 \times 128 \times 128$	$8 \times 8 \times 8 \mid 4 \times 8 \times 4$	512 | 128
	256^3	$128 \times 128 \times 128$	$16 \times 8 \times 8 \mid 8 \times 8 \times 4$	1024 | 256

For strong scalability experiments, we double the number of computation cores each time, with the size of the input grid fixed. As shown in Figure 10(a), all stencil benchmarks achieve linear speedup on Sunway platform when scaling the number of cores. Therefore, the strong scalability of MSC on Sunway platform is almost ideal. For Tianhe-3 platform, the MSC achieves almost ideal strong scalability on the 3D stencils. Whereas for the 2D stencils, the strong scalability of MSC deviates from ideal as the number of cores increases. This is because the halo regions of 2D stencils are exchanged more frequently, which leads to network congestion on the Tianhe-3 platform. Particularly, when scaling to the maximum number of cores, the average speedup (compared to the performance at the minimum number of cores) achieved by MSC is 6.74× and 5.85× on Sunway and Tianhe-3 platforms, respectively.

For weak scalability experiments, we double the number of computation cores, with the size of the input grid also doubled. In

addition, we keep the sub-grid assigned to each core fixed. As shown in Figure 10(b), the stencil codes generated by MSC achieve linear speedup as the number of cores scales. Therefore, the weak scalability of MSC is almost ideal. Particularly, when scaling to the maximum number of cores, the average speedup (compared to the performance at the minimum number of cores) achieved by MSC is 7.85× and 7.38× on Sunway and Tianhe-3 platform, respectively.

5.4 Performance Auto-tuning

To evaluate the effectiveness of auto-tuning method in MSC, we measure the performance of *3d7pt_star* stencil in large-scale execution before and after auto-tuning. Specifically, we use the input domain size of 8192×128×128 and run the stencil codes on 128 CGs of Sunway platform. The parameters to be tuned include the tilling size in each spatial dimension and the shape of MPI process grid. We invoke the auto-tuning method twice to evaluate its stability. We omit the evaluation results on Tianhe-3 platform, which reveals a similar tendency.

Figure 11 presents the stencil performance as the auto-tuning method iterates. The x-axis indicates the number of iterations using the simulated annealing algorithm. The y-axis indicates the execution time of 100 timesteps. The execution time of both runs decreases rapidly as the number of iterations increases. The two runs identify the optimal parameters after 13,460,000 (around 13 minutes) and 19,670,000 (around 16 minutes) iterations, respectively. This converged iteration time across runs proves the stability of the auto-tuning method. Using the optimal parameters identified by the auto-tuning method, the performance of the stencil code can be improved by 3.28×. The results indicate that the auto-tuning method in MSC can effectively determine the optimal parameters for generated stencil code.

5.5 Performance Comparison with SOTA DSLs

Since no stencil DSLs support the Sunway and Matrix processors, we decide to compare MSC and SOTA stencil DSLse on the CPU platform. We choose Halide (v12.0.1), Patus, and Physis for comparison. Specifically, we use Halide and Patus with OpenMP parallelism since they do not support MPI. We use Physis with MPI parallelism. Note that Physis does not support hybrid parallelism with OpenMP and MPI, whereas MSC supports well (kernel optimization + communication library). The experiments are conducted on a two-socket Intel E5-2680v4 CPU with 28 cores in total.

The parameter settings of stencils are the same as in Table 5 when comparing MSC with Halide and Patus. In addition, we set the thread number to 28. And the configuration parameters of MSC, when compared to Physis, are shown in Table 8. Note that we only adjust the number of MPI processes and OpenMP threads, whereas the input grid and the program parallelism remain unchanged. The size of the 2D and 3D input grid used to compare with Physis is $16,384 \times 28,672$ and $512 \times 512 \times 1,792$, respectively.

The performance comparison between MSC and Halide (under both JIT and AOT settings) is shown in Figure 12. Compared with Halide-JIT (baseline), the average speedup of Halide-AOT and MSC is 2.92× and 3.33×, respectively. The poor performance of Halide-JIT can be attributed to the large overhead of JIT compilation. We also notice that Halide-AOT achieves better performance than MSC on small stencils (e.g., *2d9pt_star*, *2d9pt_box*, and *3d7pt_star*), whereas MSC performs better than Halide-AOT on large stencils. After careful investigation, we realize the performance difference can be attributed to data indexing codes generated by them. Specifically, Halide-AOT generates a large number of subscript expressions for data indexing, whereas MSC can directly index the data due to its design of tensor IR. Therefore, Halide-AOT requires more computation for evaluating subscript expressions as the stencil order increases, and thus is inferior to MSC with large stencils.

Table 8: Configuration of MSC when compared with Physis on CPU platform.

Dim	Sub_grid	MPI Grid	MPI Processes	OMP Threads
2D	4096×4096	4×7	28	1
	8192×4096	2×7	14	2
	16384×4096	1×7	7	4
3D	256×256×256	2×2×7	28	1
	512×256×256	1×2×7	14	2
	512×512×256	1×1×7	7	4

The performance comparison between MSC and Patus (baseline) is shown in Figure 13. The performance of MSC is better than Patus for all stencil benchmarks, and the average speedup is 5.94×. This is because the evaluated stencil benchmarks are already bounded by memory bandwidth. However, Patus applies aggressive SIMD vectorization with SSE intrinsics, which leads to more unaligned memory accesses and thus exacerbates the memory-bound problem. In addition, the 3D star stencils require more data elements (e.g., *3d25pt_star*, *3d31pt_star*) for updating the input grid, which suffers more from discrete memory accesses, and further deteriorates the performance of Patus.

The performance comparison between MSC and Physis under different configurations is shown in Figure 14. For all stencil benchmarks, the performance of MSC is better than that of Physis, with an average speedup of 9.88×. Especially on stencil benchmarks with higher orders (e.g., *2d121pt_box*, *2d169pt_box*, *3d25pt_star*, and *3d31pt_star*), the performance of MSC is much better than Physis. This is because there is a large amount of halo exchange for the above stencils. In Physis, the halo exchange relies on the RPC runtime that coordinates the communication among all processes with a master process, which soon becomes the bottleneck as the amount of halo exchange increases. In contrast, the communication library in MSC supports asynchronous halo exchange, which is efficient for large-scale execution of high-order stencils.

5.6 Discussion

We are interested to apply MSC to real-world applications such as weather forecasting (WRF) and ocean modeling (POP2), whose performance critical kernels consist of stencil computation. For example, the *advect_mono* and *advect* subroutines from *advect_em* module of WRF, as well as the *hdifft* and *vdifft* subroutines from *baroclinic* module of POP2. The above stencils commonly require more than one input grid, along with their coefficient grids. These grids can be too large to fit in the limited local memory of the many-core processors. Therefore, MSC should manage the large input data in a streaming and pipelined manner so that it can overlap the data access and computation within the limited local memory. In addition, since WRF and POP2 suffer from serious load imbalance [41]

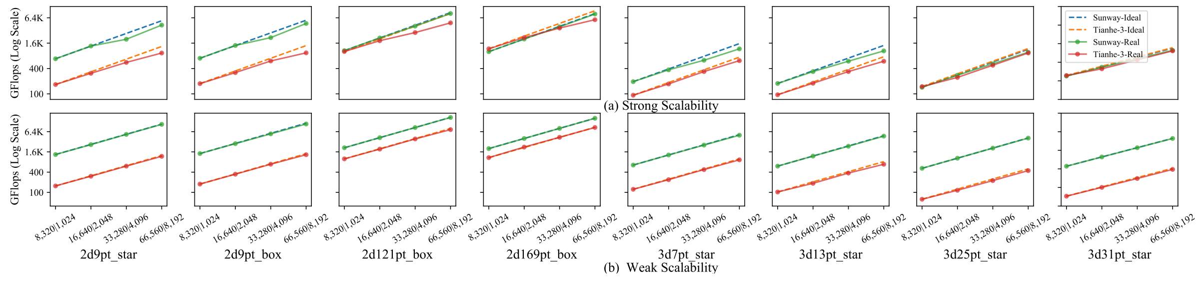

Figure 10: Strong (a)/weak (b) scalability of MSC on Sunway TaihuLight supercomputer and prototype Tianhe-3 cluster.

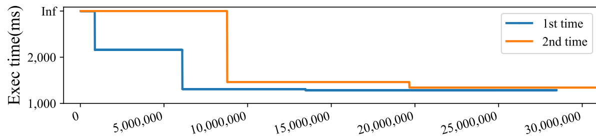

Figure 11: Auto-tuning results of *3d7pt_star* stencil in large-scale execution on Sunway platform.

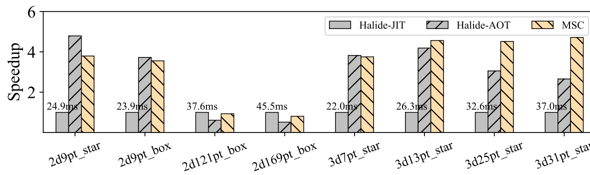

Figure 12: Performance comparison of Halide-JIT, Halide-AOT, and MSC, where the performance of Halide-JIT is chosen as the baseline.

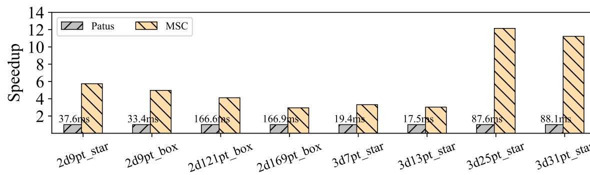

Figure 13: Performance comparison of Patus and MSC, where the performance of Patus is chosen as the baseline.

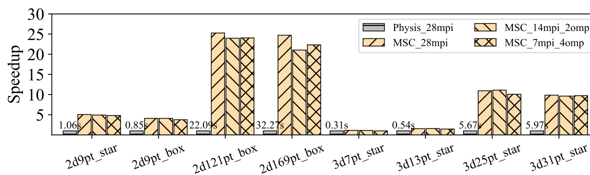

Figure 14: Performance comparison of Physis and MSC, where Physis running on 28 cores is chosen as the baseline.

in large-scale execution, the subgrids assigned to different processors may require diverging compilation optimizations. Therefore, we plan to adopt the *inspector-executor* method [6] in MSC, which analyzes the subgrids and generates the corresponding optimization schedules in the *inspector phase*, and performs compilation and code generation in the *executor phase*.

6 CONCLUSION

In this paper, we propose MSC, a new stencil DSL that generates optimized stencil codes targeting many-core processors such as Sunway and Matrix. MSC supports expressing stencil computation with multiple time dependencies and optimizes the stencil codes from both spatial and temporal dimensions. It also implements various optimization primitives to exploit the parallelism and data locality across the computation and memory hierarchies. Moreover, we design an efficient communication library to support asynchronous halo region exchange in large-scale stencil codes. This library is integrated into MSC as a plugin, which decouples from the stencil kernel optimizations and can be easily adapted to customized network topologies. The experiment results with representative stencils demonstrate that the MSC can generate optimized codes with similar or better performance than manually optimized codes on Sunway and Matrix many-core processors. In addition, MSC achieves better performance compared to existing stencil DSLs such as Patus and Physis on CPU. The MSC is open-sourced at https://github.com/buaa-hipo/MSC-stencil-compiler.

ACKNOWLEDGMENTS

This work was supported by National Key Research and Development Program of China (No. 2020YFB1506703), National Natural Science Foundation of China (No. 62072018) and State Key Laboratory of Software Development Environment (No. SKLSDE-2021ZX-06). Hailong Yang is the corresponding author.

REFERENCES

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[2] Y. Ao, C. Yang, X. Wang, W. Xue, H. Fu, F. Liu, L. Gan, P. Xu, and W. Ma. 2017. 26 PFLOPS Stencil Computations for Atmospheric Modeling on Sunway TaihuLight. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 535–544.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and

Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205.

[4] Ian J Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L Chamberlain, David G Wonnacott, and Michelle Mills Strout. 2015. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 197–206.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113.

[6] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 779–793.

[7] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 676–687.

[8] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed Halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages.

[9] Jack Dongarra, Gregory Peterson, Stanimir Tomov, Jeff Allred, Vincent Natoli, and David Richie. 2008. Exploring new architectures in accelerating CFD for Air Force applications. In *2008 DoD HPCMP Users Group Conference*. IEEE, 472–478.

[10] Jiarui Fang, Haohuan Fu, Wenlai Zhao, Bingwei Chen, Weijie Zheng, and Guangwen Yang. 2017. swdnn: A library for accelerating deep learning applications on sunway taihulight. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 615–624.

[11] Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*. 361–366.

[12] Haohuan Fu, Conghui He, Bingwei Chen, Zekun Yin, Zhenguo Zhang, Wenqiang Zhang, Tingjian Zhang, Wei Xue, Weiguo Liu, Wanwang Yin, et al. 2017. 9-Pflops nonlinear earthquake simulation on Sunway TaihuLight: enabling depiction of 18-Hz and 8-meter scenarios. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[13] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. 2016. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences* 59, 7 (2016), 1–16.

[14] Tobias Grosser, Albert Cohen, Justin Holewinski, Ponuswamy Sadayappan, and Sven Verdoolaege. 2014. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 66–75.

[15] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.

[16] Jia Guo, Ganesh Bikshandi, Basilio B Fraguela, and David Padua. 2009. Writing productive stencil codes with overlapped tiling. *Concurrency and Computation: Practice and Experience* 21, 1 (2009), 25–39.

[17] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2020. Domain-specific Multi-Level IR rewriting for GPU. *arXiv preprint arXiv:2005.13014* (2020).

[18] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. 2015. STELLA: A Domain-Specific Tool for Structured Grid Methods in Weather and Climate Models *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 41, 12 pages.

[19] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 100–112.

[20] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A Stencil Compiler for Short-Vector SIMD Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. Association for Computing Machinery, New York, NY, USA, 13–24.

[21] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*. 311–320.

[22] Yongmin Hu, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. Massively Scaling Seismic Processing on Sunway TaihuLight Supercomputer. *IEEE Transactions on Parallel and Distributed Systems* 31, 5 (2020), 1194–1208.

[23] Andreas Klöckner. 2014. Loo.Py: Transformation-Based Code Generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Edinburgh, United Kingdom) *(ARRAY'14)*. Association for Computing Machinery, New York, NY, USA, 82–87.

[24] Mingzhen Li, Yi Liu, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2019. Accelerating sparse cholesky factorization on sunway manycore architecture. *IEEE Transactions on Parallel and Distributed Systems* 31, 7 (2019), 1636–1650.

[25] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. 2020. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.* 46, 1, Article 6 (April 2020), 28 pages.

[26] Tareq Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and David Keyes. 2015. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing* 37, 4 (2015), C439–C464.

[27] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. 2011. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[28] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. 2020. AN5D: Automated Stencil Framework for High-Degree Temporal Blocking on GPUs. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 199–211.

[29] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[31] Prashant Singh Rawat. 2018. *Optimization of stencil computations on GPUs*. Ph.D. Dissertation. The Ohio State University.

[32] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2019. On Optimizing Complex Stencils on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 641–652.

[33] István Z. Reguly, Gihan R. Mudalige, and Michael B. Giles. 2018. Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 873–886.

[34] Gabriel Rivera and Chau-Wen Tseng. 2000. Tiling optimizations for 3D scientific computations. In *SC'00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE, 32–32.

[35] Allen Taflove and Susan C Hagness. 2005. *Computational electrodynamics: the finite-difference time-domain method*. Artech house.

[36] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 117–128.

[37] M.E. Wolf, D.E. Maydan, and Ding-Kai Chen. 1996. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. 274–286.

[38] Chao Yang, Wei Xue, Haohuan Fu, Hongtao You, Xinliang Wang, Yulong Ao, Fangfang Liu, Lin Gan, Ping Xu, Lanning Wang, et al. 2016. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 57–68.

[39] Xin You, Hailong Yang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2019. Performance evaluation and analysis of linear algebra kernels in the prototype tianhe-3 cluster. In *Asian Conference on Supercomputing Frontiers*. Springer, 86–105.

[40] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK—Yet another stencil kernel: A framework for HPC stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE, 30–39.

[41] Shaoqing Zhang, Haohuan Fu, Lixin Wu, Yuxuan Li, Hong Wang, Yunhui Zeng, Xiaohui Duan, Wubing Wan, Li Wang, Yuan Zhuang, et al. 2020. Optimizing high-resolution Community Earth System Model on a heterogeneous many-core supercomputing platform. *Geoscientific Model Development* 13, 10 (2020), 4809–4829.