Tessellating Stencils

Liang Yuan[†], Yunquan Zhang[†], Peng Guo[†‡], Shan Huang[†‡]
†State Key Lab. of Computer Architecture, Institute of Computing Technology, CAS
‡University of Chinese Academy of Sciences
yuanliang,zyq,guopeng,huangshan01@ict.ac.cn

ABSTRACT

Stencil computations represent a very common class of nested loops in scientific and engineering applications. The exhaustively studied tiling is one of the most powerful transformation techniques to explore the data locality and parallelism. Unlike previous work, which mostly blocks the iteration space of a stencil directly, this paper proposes a novel two-level tessellation scheme. A set of blocks are designed to tessellate the spatial space in various ways. The blocks can be processed in parallel without redundant computation. This corresponds to extending them along the time dimension and can form a tessellation of the iteration space. Experimental results show that our code performs up to 12% better than the existing highly concurrent schemes for the 3d27p stencil.

KEYWORDS

Stencil Computation, Tessellation, Tiling

ACM Reference format:
Liang Yuan[†], Yunquan Zhang[†], Peng Guo[†‡], Shan Huang[†‡]. 2017. Tessellating Stencils. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 13 pages.
DOI: 10.1145/3126908.3126920

1 INTRODUCTION

The growing gap between the processor speed and memory access time has been identified as the dominating bottleneck of computing devices. To alleviate the pressure on memory system, architecture designers employ increasing on-chip resources to form hardware- or software-managed caches on modern CPUs and GPUs and application developers maximize the in-cache data reuse.

Stencil computations represent a very common class of nested loops in scientific and engineering applications. They are the same as "structured mesh computations", used as one of the thirteen Berkeley motifs. Many algorithms in other fields, such as dynamic programming and image processing, also involve a similar dependence pattern. A stencil is a pre-defined pattern of neighbor points used for updating a given point. The stencil computation involves time-iterated updates on a regular d-dimensional grid, called the *data space* or *spatial space*. The data space is updated along the time dimension, generating a $(d+1)$-dimensional space referred as the *iteration space*. The stencils can be classified from various perspectives, such as the grid dimensions (1D, 2D, ...), number of neighbors (3-point, 5-point, ...), shapes (box, star, ...), dependence types (Gauss-Seidel, Jacobi) and boundary conditions (constant, periodic, ...).

The stencil computation is characterized by its apparently low arithmetic intensity. The naive implementation for a d-dimensional stencil is comprised of $(d+1)$ loops where the outmost loop traverses the time dimension and the inner loops update all grid points in the d-dimensional spatial space. It exhibits poor data reuse and is a typical bandwidth-bound kernel.

As one of the most powerful transformation techniques to explore the data locality and parallelism of multiple loop nests, tiling (also known as blocking) is exhaustively studied for stencil computations. The space blocking algorithms promote data reuse in a single time step for 2D and higher dimension stencils by changing the data traversal pattern in a specific order. An in-cache grid point may be reused to update all its neighbors before evicted from cache. However, the locality exploited by space blocking is limited by the neighbor pattern size of a stencil. To increase data reuse, the time dimension is often taken into consideration simultaneously with spatial dimensions. Almost all existing techniques work on the $(d+1)$-dimensional iteration space of a d-dimensional stencil.

As a representative loop type, stencils have been exhaustively studied by the compiler community in terms of the auto-generation of tiles and codes. One prevalent approach is the widely used polyhedral model, which deals with affine loop nests where lower bounds, upper bounds and array subscripts are all affine functions of surrounding loop indices and predefined constant parameters. The existing tiling methods find specific tile shape and tile size. For example, time skewed tiling (also known parallelogram in 2D, parallelepiped in 3D and parallelotope in higher dimensions) [27, 54, 68] allows to update one tile atomically. However, most of the methods fill the space with only a single tile shape and often enforce a pipelined startup and provide limited concurrency.

A number of tiling techniques have been proposed to improve the overhead of pipelined start-up [3, 6, 10, 16, 18, 22, 32, 45, 56]. Several authors [17, 22, 57] allow for cutting over an arbitrary number of spatial dimensions. A common significant problem of these compiler-generated codes is the requirement of the fixed tile size at compile time. Since run-time parametrization (symbolic tile sizes) enables efficient auto-tuning and performance portability, techniques have been developed to allow parametric tilings for the rectangle [4, 20, 21, 31, 50, 51] and the 2D diamond [5, 25].

Compilers alone often fail to full utilize hardware resources. Automatically generated or manually written codes with optional optimizations and empirical auto-tuned parameters have been demonstrated to be the effective ways to achieve high performance for a number of scientific kernels. For example, ATLAS [63] employs

SC17, Denver, CO, USA

978-1-4503-5114-0/17/11…$15.00
DOI: 10.1145/3126908.3126920

sophisticated auto-tuning techniques to generate high-performance BLAS libraries. For stencil computations, a number of auto-tuning frameworks [8, 19, 29, 35, 36, 39, 53, 72], programming models [40, 60] and hand-tuned implementations on CPUs [11, 43, 47, 61, 64], GPUs [9, 33, 44, 46, 49] and Phi [23, 71] have been proposed. However, most of them employ simple hyper-rectangular tile shapes [12, 43, 48, 52] and involve redundant computations to resolve the dependence between tiles.

We argue that, like other numerical kernels including matrix multiplication and FFT, the definition of a stencil is clear and simple. Therefore, there should be a general succinct mathematical parallelization framework for stencils of various dimensions, shapes, orders and boundary conditions. However, high-performance community often adopts hyper-rectangular tile with redundant computations, while the compiler community often focuses on the traditional approaches, such as the polyhedral model. We borrow the idea of OpenBLAS [15, 62, 70] in that we provide a simple and clear parallel framework of lightweight loop conditions to allow fine in-core optimizations.

To achieve this goal, we design a novel two-level tessellation scheme for stencil computations. Unlike the previous techniques, which mostly block the iteration space of a stencil directly, we devise a set of blocks to tessellate the spatial space in various ways. Then their extensions along the time dimension are able to form a tessellation of the iteration space. The blocks of the data space are able to executes in parallel. The scheme reveals the relation between the coordinate of a grid point and its update time in data space blocks. The proposed tessellation framework enables concurrent start, executes without redundant computation, and incurs light loop overhead.

The remainder of this paper is organized as follows. Section 2 overviews related work and discusses their deficiencies. The algorithm of the proposed tessellation scheme is described in Section 3. The implementation and optimization are presented in Section 4. We present the performance results in Section 5. Section 6 concludes the paper.

2 RELATED WORK

2.1 Tiling

Tiling [26, 34, 41, 65, 66] is one of the most powerful transformation techniques to explore the data locality and parallelism of multiple loop nests. Wonnacott and Strout present a comparison on the scalability of many existing tiling schemes [69]. We now summarize some work related to stencil computations.

Overlapped tiling. Hyper-rectangle tiling [12, 43, 48, 52] is often used for hand-optimized implementations in the high performance community. To execute more than one time step, the dependences between tiles are resolved by redundant computations [42], which is also known as overlapped tiling [24, 32]. Philips and Fatica [46] present a handwritten 3.5d blocking implementation with hand-tuned optmizations on GPUs. The 3.5D blocking method enhances the spatial 2.5D tiling scheme [52] with temporal tiling. The regularity of the hyper-rectangle shape enables high concurrency and allows to develop more fine optimizations. However, the redundant operations may outweigh the performance improvement. Hence in this work we concentrate on the redundancy-free scheme.

Time Skewing. Time skewed tiling (the resulting block shapes are parallelogram in 2D, parallelepiped in 3D and parallelotope in higher dimensions) [27, 54, 68] eliminates the redundant computations. Andonov et al. [1, 2] address the optimal tile size selection for time skewing in a 2D iteration space. However, most of the methods often enforce a pipelined startup and provide limited concurrency. Resembling the tiling techniques discussed next, our tessellation scheme enables fully parallelism such that all data blocks between synchronizations can execute concurrently.

Diamond tiling. The Pluto [7] is able to generate the diamond tiling for 1D stencil. Bandishti et al. [3, 45] extended it to higher dimension stencils. Grosser et al. [18] coarsened the apex of a diamond to form a hexagon in 2D and a truncated octahedron in 3D. Essentially, the diamond tiling finds a parallelepiped partition. The key advantage of the diamond tiling is the ability to concurrent start. A hyperplane whose orthogonal vector usually parallels with the time dimension consists of the wavefront of tiles are executed in parallel.

Cache oblivious Tiling. To exploit the data locality without explicitly knowing the memory hierarchy parameters, many cache oblivious algorithms for stencils are proposed. Frigo and Strumpen proposed the first serial [13] and parallel [14] cache oblivious stencil algorithms. The cache oblivious parallelograms method [55] splits the space-time simultaneously and can be viewed as a cache oblivious version of the time skewing algorithm with the wavefront parallelization. The Pochoir system [57] implements the hyperspace cut which allows to cut all spatial dimensions simultaneously. Comparing to the serial algorithm [13], it improves the parallelism while remains the same cache complexity.

Split tiling. To avoid the overhead of the pipelined start-up in wavefront parallelization, the split tiling method identifies and executes the independent sub-tiles in each tile concurrently and then sends them to its successors to allow the rest region executes concurrently too [10, 32, 67, 73]. Grosser et al. [17] proposed another split tiling method resembling the hyperspace cut of the cache oblivious paradigm. The nested split-tiling [22] allows for recursive split along all spatial dimensions.

Hybrid tiling. Strzodka et al. [56] and Malas et al. [37] proposed the CATS and MWD algorithms which combine the diamond tiling, parallelograms tiling and pipelined processing. Grosser et al. [16] introduced the hybrid hexagonal and parallelogram tiling algorithm. These algorithms decompose the iteration space combining hexagonal tiles or diamond tiles along the time and one spatial dimension with time skewed tiles along the remaining spatial dimensions. The hexagonal tiling extends the classic diamond tiling by stretching the tiles along the space dimension. This guarantees each tile depends on at most three predecessor tiles for high-order stencils. The hybrid split-tiling [22] combines the nested split-tiling and time skewed tiling.

2.2 Limitations

The above mentioned diamond tiling [3], hyperspace cut (cache oblivious tiling) [57] and nested split-tiling [17, 22] are three recently developed approaches to enable maximal concurrency. We identify several drawbacks of them as follows. Our tessellation scheme can overcome these limitations since it comes from a mathematical framework that exhibits a clear graphical blocking scheme.

The diamond tiling is a complier transformation technique based on the polyhedral model. The conventional code generation method requires a fixed tile size at compile time restricting the efficient auto-tuning and performance portability. Bertolacci et al. developed a parameterized version of the diamond tiling for 2D stencils [5]. But it did not resolve the automatic code generation problem [25]. Andonov et al. [1, 2] proposed a mechanism to select the optimal tile sizes based on an analytical model.

Another weakness of the diamond tiling is that it needs to process blocks of small size at the apex of a diamond. Grosser et al. [18] exhibited the relationship between the 2D diamond and the extended hexagonal tile. and artificially selected the truncated octahedron tiling as the extension of 3D diamond. However, there is no such simple illustration on coarsening the blocks for 3D or higher dimension stencils.

Finally, the diamond tiling fills the entire iteration space with skewed hyper-rectangles. It's hard to choose the proper tile sizes to ensure the concurrent start [18]. Though it can be combined with other hyper parallelepiped tiling to account for hierarchical memory systems, it's difficult to develop a multilevel diamond tiling directly.

The common cache oblivious implementation suffers the overhead of recursion and achieves limited speedups [30, 71]. The hyperspace cut performs the interleaved temporal and spatial cuttings. The recursive divide-and-conquer implementation of these algorithms introduces artificial dependencies between sub-blocks [58]. Tang et. al. proposed the Cache Oblivious Wavefront [58, 59] to eliminate these false dependencies and improve the parallelism and reduce the critical path of algorithms. However, it will incur runtime overhead of checking the wavefront data structure [28].

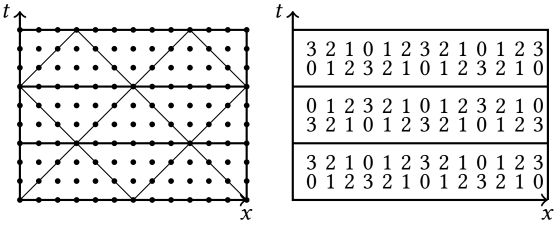
The major problem of the nested split-tiling is the high synchronization overhead, 2^d synchronizations for d-dimensional stencil. It should note that Tang et al. [57] correctly proves the same synchronization overhead. However, after careful examination to their codes, we found that the implementation based on the common cache-oblivious recursive framework adopts the same strategy as the one used in nested split-tiling, which incurs 2^d synchronizations for d-dimensional stencil. Though it can utilize dynamic queues to improve the synchronization overhead, the cost of the runtime scheduling may negate the benefit.

3 ALGORITHM

We use 1-order star stencils (1D 3-point, 2D 5-point and 3D 7-point) as our examples. However, we will show that our scheme is able to adapt to any kinds of stencils. The Gauss-Seidel stencil enforces a point-wise 45° pipeline startup and it cannot be improved to a fully concurrent start schedule with tiling [32]. Thus we only discuss Jacobi stencils in this work.

3.1 Reformulating the 2D Diamond tiling

We first present a new approach to analyzing the 2D diamond tiling, which inspires our new two-level tiling scheme for n-dimensional stencils. The new explanation can be viewed as different space tessellations with various tile shapes.



(a) 2D Representation **(b) A New Representation**

Figure 1: Diamond tiling of 1D Stencil

Figure 1a shows the diamond tiling of a 1D 3-point stencil. To provide a new explanation, each diamond is further split into two sub-blocks of equal size by a line parallel to the x axis and a synchronization is associated with each line. As a result, the iteration space is filled by triangles and inverted triangles and the execution consists of interleaved computations of them. Each grid point on a splitting line stays in a same time step. Thus the computations between two splitting lines can be view as a 2D *time tile*, in which every grid point starts in a same time dimension and is updated by identical steps.

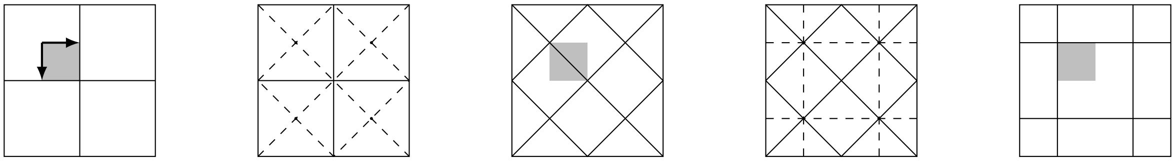
A triangle or inverted triangle, i.e, a block in the iteration space, can be represented by associating each grid point with its number of updates along the time dimension. For the example in Figure 1a, the block (the projection of a triangle on the data space) contains 7 grid points and the new representation is $(0, 1, 2, 3, 2, 1, 0)$, where the centerpoint is updated 3 times and its neighbors are updated less steps proportional to their distances to the centerpoint. To form a time tile where all grid point are updated 3 times, every block is split into two sub-blocks of equal size. Two sub-blocks from two adjacent triangle blocks are combined to form a new block whose time dimension values are $(3, 2, 1, 0, 1, 2, 3)$. Then every point can be updated to 3 times by adding the projection of a inverted triangle $(0, 1, 2, 3, 2, 1, 0)$ on the new block. Figure 1b illustrates the full corresponding execution.

Two issues should be noted when extending the scheme to higher-dimensional stencils. First, though the triangle and inverted triangle generate the same new representation, their executions are different. The grid points in the triangle start the computation simultaneously from the beginning, while end the computation at the same time in the inverted triangle. Second, the synchronizations added on splitting lines can be eliminated by reusing the block arrangement between time tiles.

3.2 Overview

The whole scheme for high dimension stencils is similar to the reformulation of the 2D diamond tiling described above. It consists of identical phases (time tiles). All grid points are in a same time dimension before a phase and updated same time steps after.

A phase contains $d + 1$ stages for a d-dimensional stencil. Specifically, in the first stage the d-dimensional data space is tessellated by d-dimensional hypercubes of size $2b + 1$ on each dimension. All hypercube blocks are then processed in parallel following the



(a) B_0 **(b)** $B_0 \to B_1$ **(c)** B_1 **(d)** $B_1 \to B_2$ **(e)** B_2

Figure 2: Tessellating Data Space of 2D Stencil

maximal updating scheme. This corresponds to extending the d-dimensional blocks to $(d+1)$-dimensional blocks, which are used for tessellating the iteration space. The remaining d stages repeat this process with different tiling blocks for the data space. After $d+1$ stages, all the extended $(d+1)$-dimensional blocks can form a time tile where every grid point is updated b time steps.

3.3 Tessellating Data Space

We now describe the $d+1$ blocks, B_i $(0 \leqslant i \leqslant d)$, used for the d-dimensional data space tessellation. In the stage i, the spatial space is tessellated by B_i. Similar to the 1D stencil, the first block B_0 is the hypercube (line segment in 1D, square in 2D, cube in 3D, tesseract in 4D, ...). B_{i+1} is constructed by splitting B_i into subblocks along some dimensions and then gluing some neighbor subblocks from different B_i blocks.

Figure 2 (ignore the gray square) describes the three block shapes for the 2D stencil and how they are transformed from their predecessors. The first block B_0 is a square. To form B_1, every B_0 is split (dashed lines) into four subblocks induced by the faces of B_0 that are orthogonal to the x and y axes. Every interior point of B_0 is assigned to the subblock of the side to which it is closest. Every two subblocks from adjacent B_0 blocks that share the same side are glued to form a B_1. Notice that though all B_1 blocks have the same diamond shape, they are built from two different directions, △ + ▽ → ⟠ and ◁ + ▷ → ◇⃒. To form B_2, each B_1 block is split along the dimension orthogonal to the dimension along which it's glued, ◇⃒ → △ + ▽ and ⟠ → ◁ + ▷. Then four neighbor subblocks are combined to form B_2, ▵ + ▿ + ◃ + ▹ → ⊠.

To present a formal description of the blocks for the d-dimensional stencil, a Cartesian coordinate system is introduced to the spatial space. Consider a B_0 block whose centerpoint is the origin $(0, 0, \ldots, 0)$. Other B_0 blocks' centerpoints are $(2k_0 b, 2k_1 b, \ldots, 2k_{d-1} b)$ where k_i is an integer. Since we are discussing a tessellation, there is no loss of generality in assuming $k_i = 0$.

Given a B_i $(0 \leqslant i < d)$ block, it's split by the faces which are orthogonal to the dimensions in which the coordinates of its centerpoints are $0 \pmod{2b}$. Then a B_{i+1} block is formed by combining all subblocks along dimensions in which its centerpoint is $b \pmod{2b}$. The centerpoints of all B_1 blocks which contain at least one point of the B_0 block are $(\pm b, 0, \ldots, 0), \ldots, (0, 0, \ldots, \pm b)$. Similar, all points that have value $\pm b$ in i dimensions and 0 in the rest $d-i$ dimensions are centerpoints of B_i $(i \geqslant 1)$ blocks. Thus, a B_i $(0 \leqslant i < d)$ block is split into $2(d-i)$ sublocks and a B_j $(0 < j \leqslant d)$ block is formed from $2j$ sublocks.

There exists another easier explanation for the transformation from B_i to B_{i+1}: connect the centerpoint of a B_i block and all the boundaries ($(d-2)$-dimensional) of its $2(d-i)$ splitting sides ($(d-1)$-dimensional) (dash lines in Figure 2) and then remove all the sides of B_i (solid lines in Figure 2).

Table 1: Properties of d-dimensional Stencil Tessellation

Stencil Dim	d
# stages in each phase (time tile)	$d+1$
Size of B_0	$(2b+1)^d$
# sub-blocks from B_i splitting	$2(d-i)$
# sub-blocks to combine B_i	$2i$
# B_i $(i > 0)$ centerpoints on surface of B_0	$2^i C_d^i$
# B_i $(i \geqslant 0)$ centerpoints on surface of B_0^+	C_d^i
# kinds of shapes for data space tessellation	$\lceil (d+1)/2 \rceil$

All the center points of B_j are on the surface of a B_i $(j > i)$ block and there are total $2(d-i)$ B_{i+1} center points on the surface of B_i. There are $2^i C_d^i$ centerpoints of B_i $(i > 0)$ on the surface of a B_0 block and each centerpoint is shared by 2^i B_0 blocks. Thus a B_i consists of elements from $2i$ adjacent B_{i-1} blocks or 2^i adjacent B_0 blocks. The number of B_i blocks is C_d^i times larger than the number of B_0 blocks. Consequentially, the volume of the B_i block is C_d^i times smaller than the B_0 block.

Several properties can be obtained from the data space tessellation scheme. First, every point belong to only one B_i except the points on the boundaries and B_i with its rotations can form a tessellation of the spatial space. These statements are trivial according to the transformation and their proofs are omitted here. However, we will present a simple proof in the iteration space tessellation scheme.

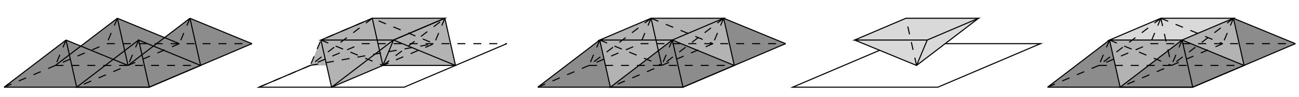
Second, according to the description, it's easy to prove that B_0 and B_d have the same shape. Notice that $B_i \to B_{i+1}$ is similar to $B_{d-i} \to B_{d-i-1}$. Combination of these leads to the following lemma, which shows that there are $\lceil (d+1)/2 \rceil$ kinds of blocks for tessellating a d-dimensional spatial space.

LEMMA 3.1. *$B_i = B_{d-i}$ $(0 \leqslant i \leqslant d)$.*

Table 1 summarizes the properties of the data space tessellation for a d-dimensional stencil.

3.4 Maximal Updating in Time Dimension

The fundamental idea of the maximal updating is presented in this subsection, while the specific formulation is discussed in the next subsection. Given a block of grid points, the maximal updating forwards each point along the time dimension until the dependence defined by the stencil can not be satisfied. Formally, the updating



(a) $\mathbb{B}_0$ (b) $\mathbb{B}_1$ (c) $\mathbb{B}_0$+$\mathbb{B}_1$ (d) $\mathbb{B}_2$ (e) $\mathbb{B}_0$+$\mathbb{B}_1$+$\mathbb{B}_2$

Figure 3: Tessellating Iteration Space of 2D Stencils

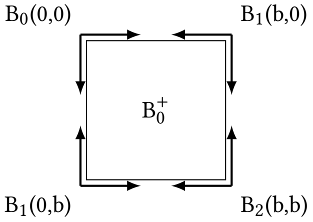


Figure 4: Cartesian Coordinate Systems

stops when there exist at least one neighbor $a' \in neighbor(a)$ for every grid point a such that the time dimension of a' is strict less than a, $time[a'] < time[a]$ or $a' \notin \mathrm{B}_i$. We call this the *correctness condition.*

The maximal updating scheme allows us to ignore the time dimension when performing the tiling scheme. That is to say, given a block B_i of the d-dimensional spatial space, an extended $(d+1)$-dimensional block, denoted as $\mathbb{B}_i$, can be generated by associating every point with the number of updated time steps. Furthermore, the maximal updating scheme allows all the spatial block to execute concurrently since there is no dependence between blocks.

3.5 Tessellating Iteration Space

Since the blocks are defined with respect to a center point, we exploit symmetry and concentrate on a sub-block of B_0 for the discussion of the iteration space tessellation. Specifically, consider a B_0 block whose centerpoint is the origin $(0, 0, \dots, 0)$ of a Cartesian coordinate system, only the grid points in the nonnegative orthant are targeted, i.e., each Cartesian coordinate of the point is nonnegative. This sub-block is denoted as B_0^+ or $\mathbb{B}_i^+$ after the maximal updating in stage i.

For the 2D stencil, the targeted B_0 is the top left square in Figure 2a. As shown in Figure 2a, the positive directions of the two Cartesian coordinate axes are downward and rightward respectively. The resulting B_0^+ is the gray region in Figure 2.

As mentioned above, there are C_d^i centerpoints of B_i blocks on the surface of a B_0^+ and totally $\mathrm{C}_d^0 + \dots + \mathrm{C}_d^d = 2^d$ centerpoints which are the 2^d vertices of the B_0^+ (a d-dimensional hypercube). To formalize the maximal updating and the iteration space tessellation, each centerpoint is associated with a Cartesian coordinate system in a way such that every point of B_0^+ is in its nonnegative orthant. Figure 4 illustrates the case for the 2D stencil.

Consider a point a in B_0^+, whose coordinate in the $\mathrm{B}_0(0, \dots, 0)$ coordinate system is $(a_0, \dots, a_{d-1})$. Its maximal updated time in stage i is denoted as $T_i(a_0, \dots, a_{d-1})$. Without loss of generality, assume the centerpoint of the B_i block containing a is $(b, \dots, b, 0, \dots, 0)$ where the first i coordinates are b. We call the dimensions of coordinate 0 *starting dimensions* of B_i and refer to the rest dimensions as *ending dimensions.* The point a's new coordinate in B_i's coordinate system is $(b - a_0, \dots, b - a_{i-1}, a_i, \dots, a_{d-1})$. In stage i, a starts to update at the time equal to its maximal distance to the centerpoint in the starting dimensions,

$$T_i^s(a_0, \dots, a_{d-1}) = \max(b - a_0, \dots, b - a_{i-1})$$

and ends before the time equal to its maximal distance to the centerpoint in the ending dimensions.

$$T_i^e(a_0, \dots, a_{d-1}) = b - \max(a_i, \dots, a_{d-1})$$

Since each block B_i updates b time steps, we obtain

$$\begin{aligned} & T_i(a_0, \dots, a_{d-1}) \\ = \; & T_i^e(a_0, \dots, a_{d-1}) - T_i^s(a_0, \dots, a_{d-1}) \\ = \; & b - \max(a_i, \dots, a_{d-1}) - \max(b - a_0, \dots, b - a_{i-1}) \end{aligned}$$

Table 2: Tessellating Iteration Space of 2D Stencil

	$\mathbb{B}_0^+$				$\mathbb{B}_1^+$				$\mathbb{B}_2^+$			
T_i^s	0	0	0	–	–	2	1	0	–	–	–	–
	0	0	0	–	2	–	1	0	–	2	2	2
	0	0	0	–	1	1	–	0	–	2	1	1
	–	–	–	–	0	0	0	–	–	2	1	0
T_i^e	3	2	1	–	–	3	3	3	–	–	–	–
	2	2	1	–	3	–	2	2	–	3	3	3
	1	1	1	–	3	2	–	1	–	3	3	3
	–	–	–	–	3	2	1	–	–	3	3	3
T_i	3	2	1	–	–	1	2	3	–	–	–	–
	2	2	1	–	1	–	1	2	–	1	1	1
	1	1	1	–	2	1	–	1	–	1	2	2
	–	–	–	–	3	2	1	–	–	1	2	3

Table 2 shows the corresponding maximal updating and the iteration space tessellation of 2D stencils. Notice that the '–'s correspond to the boundary between B_i blocks. Thus we can see that all grid points in B_0^+ belong to two adjacent B_1 blocks at stage 1 and a same B_2 block at stage 2. One can easily examine that all the updates respect dependences define by the stencil. The summation $\mathbb{B}_0 + \mathbb{B}_1 + \mathbb{B}_2$ constitutes a time tile $\mathbb{D}_2$, where each grid point of B_0 is updated b time steps. Table 3 shows the T of the tessellation of a B_0^+ block for a 3-D stencil. k is the Z dimension and each matrix in a table cell is a block in XY dimension. There are three $\mathbb{B}_1$ and $\mathbb{B}_2$ blocks, which are separately shown in the last three rows.

Figure 3 shows the iteration space tessellation of the 2D stencil pictorially. Notice that B_1 is a tetrahedron and combination of these

Table 3: Tessellating Iteration Space of 3D Stencil

	$k = 0$				$k = 1$				$k = 2$				$k = 3$					$k = 0$				$k = 1$				$k = 2$				$k = 3$			
$\mathbb{B}_0^+$	–	–	–	–	1	1	1	–	2	2	1	–	3	2	1	–	$\mathbb{B}_3^+$	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	–	–	–	–	1	1	1	–	2	2	1	–	2	2	1	–		–	1	1	1	–	1	1	1	–	1	1	1	–	–	–	–
	–	–	–	–	1	1	1	–	1	1	1	–	1	1	1	–		–	1	2	2	–	1	2	2	–	1	1	1	–	–	–	–
	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–	1	2	3	–	1	2	2	–	1	1	1	–	–	–	–
$\mathbb{B}_1^+$	3	2	1	–	2	1	–	1	1	–	1	2	–	1	2	3	$\mathbb{B}_2^+$	–	1	2	3	–	1	2	2	–	1	1	1	–	–	–	–
	2	2	1	–	1	1	–	1	–	–	1	2	1	–	1	2		1	–	1	2	1	–	1	1	1	–	–	–	–	1	1	1
	1	1	1	–	–	–	–	1	1	1	–	1	2	1	–	1		2	1	–	1	2	1	–	–	1	–	1	1	–	1	2	2
	–	–	–	–	1	1	1	–	2	2	1	–	3	2	1	–		3	2	1	–	2	1	–	1	1	–	1	2	–	1	2	3
$\mathbb{B}_1^+(0,0,b)$	3	2	1	–	2	1	–		1	–			–				$\mathbb{B}_2^+(b,b,0)$													–	–	–	–
	2	2	1	–	1	1	–		–	–																	–	–	–	–	1	1	1
	1	1	1	–	–	–	–																	–	–		–	1	1	–	1	2	2
	–	–	–	–																	–			–	1		–	1	2	–	1	2	3
$\mathbb{B}_1^+(0,b,0)$													–				$\mathbb{B}_2^+(b,0,b)$	–	1	2	3	–	1	2	2	–	1	1	1	–	–	–	–
									–	–			1	–					–	1	2		–	1	1		–	–	–				
					–	–	–		1	1	–		2	1	–					–	1			–	–								
	–	–	–	–	1	1	1	–	2	2	1	–	3	2	1	–					–												
$\mathbb{B}_1^+(b,0,0)$				–			–	1		–	1	2	–	1	2	3	$\mathbb{B}_2^+(0,b,b)$	–				–				–				–			
				–			–	1		–	1	2		–	1	2		1	–			1	–			1	–			–			
				–			–	1			–	1			–	1		2	1	–		2	1	–		1	–			–			
				–				–				–				–		3	2	1	–	2	1	–		1	–			–			

blocks can form a time tile of size b. For 3D stencils, the B_1 and B_2 are octahedrons.

The following Lemma 3.2 states a unified form of T_i.

Lemma 3.2.
$$T_i(a_0, \ldots, a_{d-1}) = \min(b, a_i, \ldots, a_{d-1}) - \max(0, a_0, \ldots, a_{i-1})$$

Proof. From the definition we obtain the following formulations for T_0, T_d and T_i $(0 < i < d)$, whose combination completes the proof.

$$\begin{aligned} T_0(a_0, \ldots, a_{d-1}) &= b - \max(a_0, \ldots, a_{d-1}) \\ T_d(a_0, \ldots, a_{d-1}) &= b - \max(b - a_0, \ldots, b - a_{d-1}) \\ &= \min(a_0, \ldots, a_{d-1}) \\ T_i(a_0, \ldots, a_{d-1}) &= \min(a_i, \ldots, a_{d-1}) - \max(a_0, \ldots, a_{i-1}) \end{aligned}$$

□

Lemma 3.2 can be combined with Lemma 3.1 to yield the following lemma, whose proof is omitted. Lemma 3.3 states that after assigning the number of updated time steps to each grid point in block B, there are still equivalences between $\mathbb{B}$ blocks of different stages.

Lemma 3.3. $\mathbb{B}_i = \mathbb{B}_{d-i}$ *$(0 \leqslant i \leqslant d)$.*

There are C_d^i centerpoints on the surface of B_0^+, The following lemma, whose proof is omitted, can be used to deduce that a grid point of B_0^+ belongs to only one B_i block except points on the boundaries. The reason is that, when assigning the updated time to these C_d^i B_i block according to Lemma 3.2, only one is strict larger than 0.

Lemma 3.4. *Consider a set A of d numbers, $\{a_0, \ldots, a_{d-1}\}$, where, without loss of generality, $a_0 \geqslant \cdots \geqslant a_{d-1}$. There are C_d^i ways to split A into two disjoint subsets A_1 and A_2 such that $|A_1| = i$ and $|A_2| = d - i$ $(0 < i < d)$. We have*

$$\min(A_1) - \max(A_2) = a_{i-1} - a_i \geqslant 0$$

when A_1 contains the i largest numbers $a_0, \ldots, a_{i-1}$. For all other $C_d^i - 1$ cases, we have

$$\min(A_1) - \max(A_2) \leqslant 0$$

Thus, it's also easy to see that before stage i, the accumulated update time of the former stages of a grid point whose coordinates in the starting dimensions are 0 is 0. After stage i, the accumulated update time of the former stages of a grid point whose coordinates in the ending dimensions are 0 is b.

The following theorems are the main contributions of this work, which show the correctness of our tessellation scheme. Theorem 3.5 states that the summation of all $\mathbb{B}_i^+$ blocks corresponds to update every grid point the same time steps b, namely forming a time tile $\mathbb{D}_d^+$.

Theorem 3.5. $\mathbb{D}_d^+ = \sum_{i=0}^{d} \mathbb{B}_i^+$.

Proof. Without loss of generality, consider a point $(a_0, \ldots, a_{d-1})$ where $a_k \geqslant a_{k+1}$, according to Lemma 3.2, we have

$$\begin{aligned} & \sum_{i=0}^{d} T_i(a_0, \ldots, a_{d-1}) \\ = \; & (b - a_0) + (a_0 - a_1) + \cdots + (a_{d-2} - a_{d-1}) + a_{d-1} \\ = \; & b \end{aligned}$$

□

We next show that the updating satisfies the dependence defined by a stencil between stages. Theorem 3.6 states that after each stage, the accumulated time of each grid point obeys the *correctness condition* mentioned above.

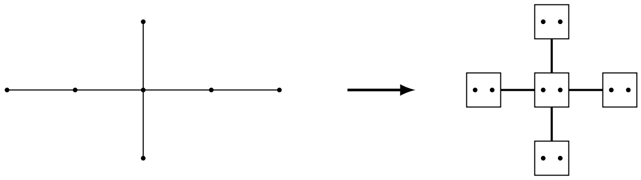


Figure 5: Handling High-order Stencil

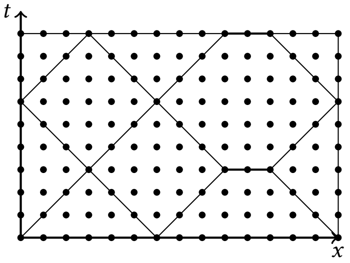


Figure 6: Handling Periodic Boundary Condition

Theorem 3.6. *After stage i, every point satisfies the dependences defined by the stencil.*

Proof. Without loss of generality, consider a point $(a_0, \ldots, a_{d-1})$ where $a_i \geqslant a_{i+1}$, according to Lemma 3.2, after stage k its accumulated time is:

$$\sum_{i=0}^{k} T_i(a_0, \ldots, a_{d-1}) = b - a_k$$

All its neighbor only have a 1 larger or smaller in one dimension. Thus after each stage the accumulated time of its neighbor is in the range $[T_i(a_0, \ldots, a_{d-1}) - 1, T_i(a_0, \ldots, a_{d-1}) + 1]$. This equals to the correctness condition. □

To complete the correctness statement, the last step is to prove that the computations inside each block respect the dependence. We omit the proof as it shares the same idea of the ones used in the above theorems.

3.6 Extensions

The discussion in previous subsections targets 1-order star stencils. However, the framework also works on box stencils i.e., $(\pm 1/0, \ldots, \pm 1/0)$ (1D 3-point, 2D 9-point, 3D 27-point, 3^d points in d-dimensional stencil) in the previous time constitute the stencil for updating $(0, \ldots, 0)$ in the current time. One can verify that all the above lemmas and theorems still apply to box stencils.

High-order stencils are common in real-world applications. If the order in one dimension of a stencil is m and the dependence pattern is symmetric, we can combine every m points along that dimension, termed as a supernode. This reduces the m-order dependence to a 1-order one between supernodes. Figure 5 shows an example where $m = 2$ along the x dimension. We can perform this combination for all other dimensions with high order dependence. Consequently, the tessellation with B_i can be applied to the combined 1-order stencil with supernodes and all grid points in a supernode are updated the same number of time steps in each $\mathbb{B}_i$.

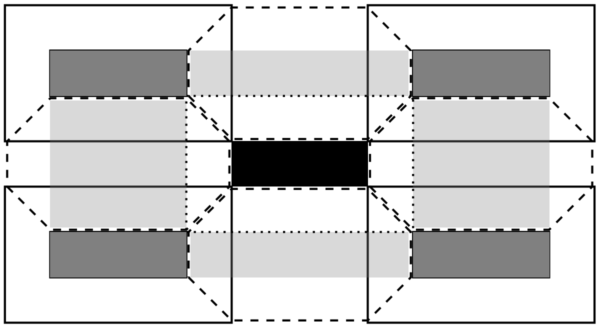


Figure 7: Coarsening Tessellation for 2D stencil

To handle periodic boundary conditions, it may need to stretch a block for 1-D or a set of blocks for higher-dimensional stencil. Figure 6 illustrates an example where the input size is not a multiple of the block size. Yielded by stretching a diamond block along the spatial dimension, one hexagonal block is included to guarantee that the two sub blocks on the two sides can be combined to a diamond block.

With these properties, we state that the proposed tessellation framework applies to all kinds of Jacobi stencils.

4 IMPLEMENTATION

With the mathematical description of the proposed framework, it is easy to implement the code manually. In the following section we will present the performance results of our hand-written implementations. An automatically code generating tool is left as future work.

4.1 Parallelization

It is easy to parallelize the tessellation scheme as all the blocks in the same stage can execute concurrently. Hence, we simply use the OpenMP pragma *parallel for* on shared memory machines.

For distributed memory computers, the clear tessellation scheme also enables us to generate a simple data/computation distribution and an efficient data communication plan. However, this is beyond the scope of this paper.

4.2 Coarsening

For 3D and higher-dimensional stencils, the size in each dimension is often smaller than 1000. To utilize the hardware prefetch effect, existing methods often choose to leave the unit-stride dimension uncut. However, this would limit the concurrency. Moreover, the time-dimensional block size b (produced by maximal updating) is limited by the smallest one among block sizes in all spatial dimensions. Thus the unequal partition of spatial space may yield a limited data reuse in time dimension.

Another driver of coarsening is that our tessellation scheme will incur ineffective data access patterns. For the 2D case, the two tetrahedron B_1 blocks, as shown in Table 2, will yield the updating of the elements in a column of the row-major array during the first or last time step. This inefficient spatial locality usage may degrade the performance.

Our new framework is highly modifiable, allowing to cut each dimension in different sizes and then coarsen the block by stretching it along each dimension. For easy explanation, Figure 7 shows the coarsening of a 2D stencil. The four rectangles correspond to B_0 blocks which have different size in the two spatial dimensions. The maximal updated time is limited by the half of the smaller size. The region of points which are updated at the first (last) time step of each $\mathbb{B}_i$ is termed as *starting (ending) region*. To improve the spatial locality, the ending region (the dark gray regions in Figure 7) of $\mathbb{B}_0$ and the starting region (the black region in Figure 7) of $\mathbb{B}_d$ are coarsened to a 2D block of the same size. Thus, the coarsened implementation allows to process a d-dimensional block for every time step in every stage. For example, the gray regions are the ending regions of $\mathbb{B}_1$ blocks rather than a column.

4.3 Merging B_d and B_0

As mentioned above, the blocks in the last and first stage share the same shape in the spatial space. Merging B_d and B_0 between phases can reduce one synchronization and improve the data reuse.

Careful examination of Figure 7 leads the condition that the distance between two B_0 along a dimension should equal the ending block size of B_0 in that dimension. This guarantees that the starting block of B_0 is identical to the ending block of B_d.

5 EXPERIMENTAL RESULTS

5.1 Setup

Our experiments were conducted on a machine made of two Intel Xeon E5-2670 processors with 2.70 GHz clock speed. We scaled the codes from uni-core to all the 24 cores of the two sockets. Each core owns a 32KB private L1 cache, 256KB private L2 cache and a unified 30MB L3 cache shared by 12 cores. We complied the program with the ICC complier version 16.0.1, using the optimization flag '-O3 -openmp'.

We compared our scheme with two of the three above mentioned highly concurrent approaches: Pluto and Pochoir. It's unfair to compare with the split tiling code since it incorporates an efficient vectorization method. In contrast, the evaluated schemes only involve the same naive in-core codes with pragmas, which is vectorized by the compiler. We also compared with Girih [38] for the 3d7p stencil.

Table 4: Problem Sizes for the Benchmarks

Benchmark	Problem Size	our blocking	Pluto blocking
Heat-1D	12000000 × 4000	2000 × 1000	2000 × 2000
1d5p	12000000 × 4000	2000 × 500	2000 × 2000
Heat-2D	6000^2 × 2000	128 × 256 × 64	64 × 64 × 64
2d9p	6000^2 × 2000	128 × 256 × 64	64 × 64 × 64
Game of life	6000^2 × 2000	128 × 256 × 64	128 × 128 × 128
Heat-3D	256^3 × 1000	24 × 24 × 12	12 × 12 × 12
3d27p	256^3 × 1000	24 × 24 × 12	12 × 12 × 12

Our test applications include four star stencils (Heat-1D 3-point, 1D 5-point, Heat-2D 5-point and Heat-3D 7-point) and three box stencils (2D 9-point, game of life and 3D 27-point) with non-periodic boundary conditions. All these kernels are included in the Pluto and Pochoir package. The problem sizes for various benchmarks are listed in Table 4. Pluto used the same blocking sizes to the ones mentioned in [3]. We also used the same sizes for the corresponding parameters. Since our codes contain more parameters, we chose to set other parameters to the half or double of the blocking size. However, we observed that the performance is very sensitive to the tile sizes, but this requires significant effort in auto tuning and should be done separately. The Pochoir was unchanged in the comparison and the default cutoff sizes are 100 × 100 × 5 for 2d benchmarks and 1000 × 3 × 3 × 3 for 3d benchmarks [57].

5.2 Analysis

Overall, our scheme achieves comparable performance on star stencils and better performance on box stencils. Pluto and Pochoir exhibit the similar performance trends as shown in [3]. Pochoir exhibits better scalabilities than Pluto while Pluto often obtains better performance. Our code achieves better speedups and scalabilities.

Figure 8 shows the results of 1D stencils. All these three schemes achieve linear scaling on both stencils. The block size is 2000 for our code and Pluto. Our performance is comparable to Pluto and better than Pochoir. The reason is that our scheme and PluTo produce the same diamond tiling codes, while Pochoir utilizes a dynamic blocking method which generates trapezoidal blocks of different sizes.

Figure 9 shows the results of the game of life benchmark. The Pochoir beats Pluto with less than 12 cores, which exhibits the same results in [3]. However, for more than 13 cores, Pluto obtains much higher speedups than Pochoir. Our code achieves the higher performance than the other two schemes and obtains an ideal scalability.

Figure 10 shows the results of 2D stencils. For the Heat-2D 5-point stencil, both our code and Pochoir exhibit a similar performance and scalability. Pluto suffers the load imbalance penalty as explained in [3]. Pluto only outperforms the other two less than 5% with 24 cores. For the 2D 9-point stencil, Pochoir and Pluto yield the similar trend of the 5-point stencil. However, our code outperforms them by 14% and 20% on average.

The results of 3D stencils are presented in Figure 11b and Figure 11a. Notice that codes of Pluto, Pochoir and ours leave the unit-stride dimension uncut. Our code and Pochoir still exhibit better scalability than Pluto. Similar to the 2D star case, Pluto obtains better performance with more than 20 cores than Pluto and our code for the 3D 7-point stencil. For 3D 27-point box stencil, Girih and Pochoir acheive the similar performance. Our code significantly outperforms Pluto and Pochoir by a maximum of 74% and 100%, and by 30% and 99% on average.

Figure 12 shows the memory transfer volume and memory bandwidth of the Heat-3D kernel. Our code and Pluto exhibit the similar cache complexity. Since Girih aims at maximal utilizing the last level shared cache, its requirement on memory access is the best.

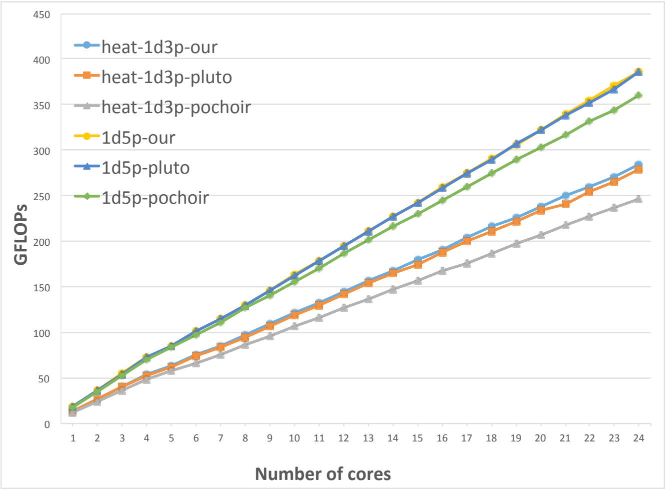


Figure 8: 1D results

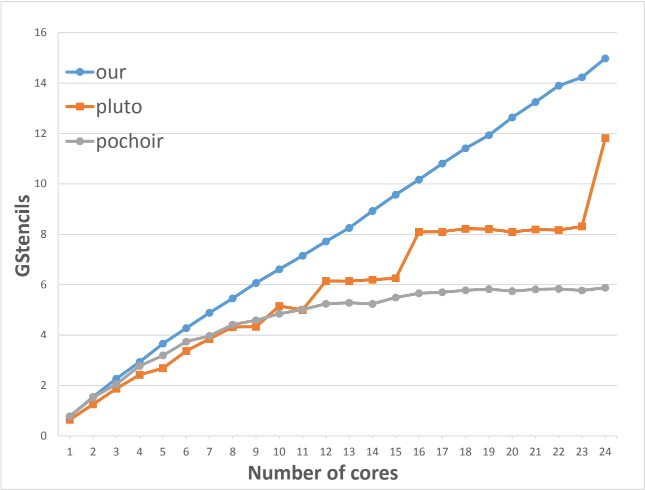
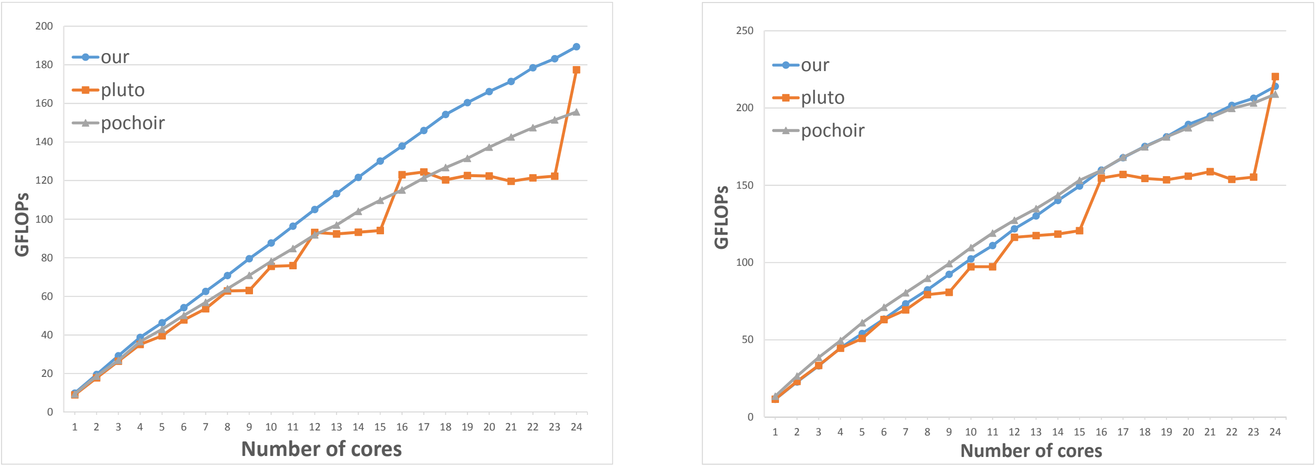


Figure 9: Game of life



(a) 2d9p

(b) Heat-2D

Figure 10: 2D results

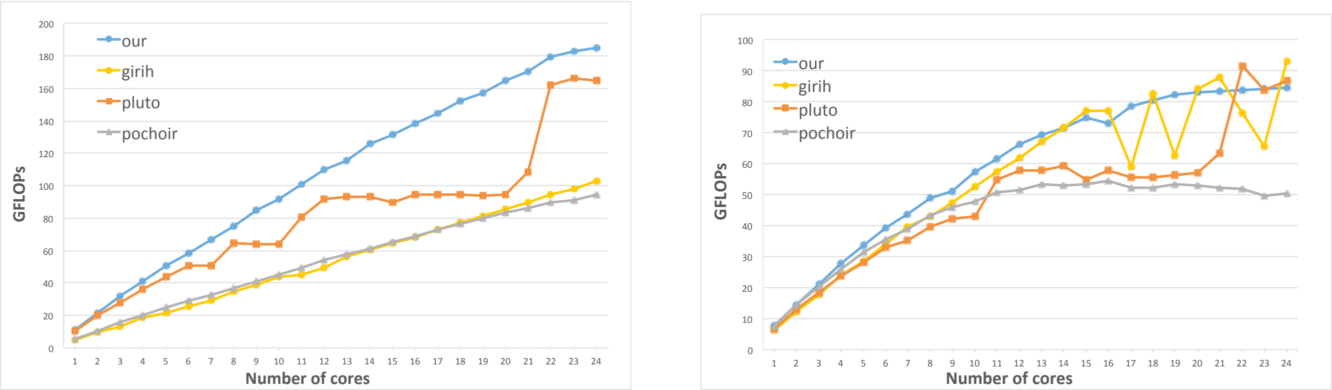
6 CONCLUSION

We have presented a new parallelization framework with highly concurrent execution for stencils. It is based on a novel two-level tessellation scheme. A set a blocks is developed to tessellate the space dimensions and their extensions can form a tessellation for the iteration space. Our scheme leads to a succinct parallelization framework and reveals the relationship between the coordinates and the updating time steps of a grid point. Experimental results shows that our scheme achieved comparable performance on star stencils and better performance on box stencils. For the 3d27p box stencil, our code performs up to 12% better than Pluto.

Future work will design an tool to automatically generate the stencil codes based on the proposed framework. Since it contains more parameters than other approaches, our ongoing work focuses on the auto-tuning method to efficiently search the best block sizes.

ACKNOWLEDGMENTS

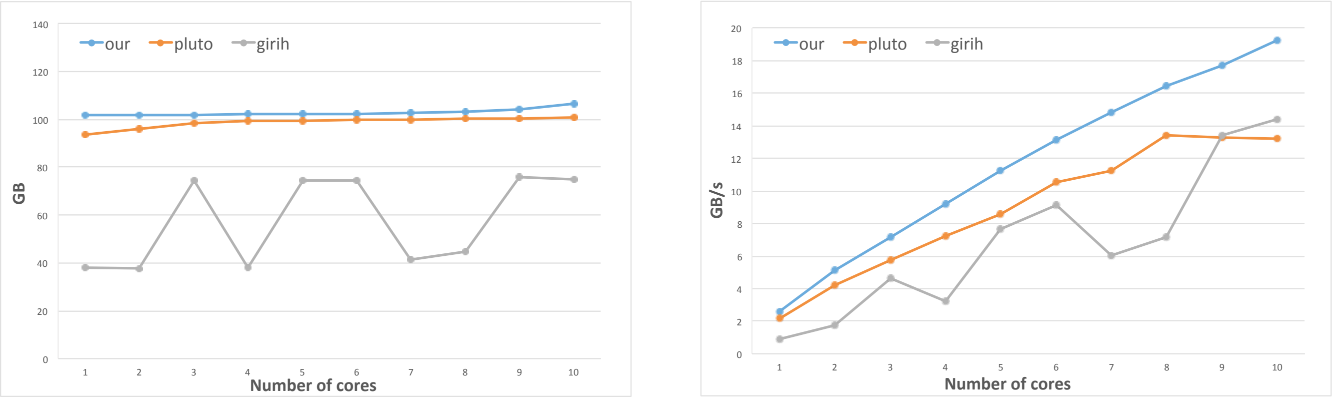
The authors would like to thank our paper shepherd and the reviewers for their valuable comments and helpful suggestions. This work is supported by the National Key Research and Development Program of China under Grant No.2016YFB0200803; the National Natural Science Foundation of China under Grant No.61432018, No.61402441, No.61272136 and No. 61521092; the National High Technology Research and Development Program of China under Grant No.2015AA01A303 and No.2015AA011505.



(a) 3d27p

(b) Head-3D

Figure 11: 3D results



(a) Heat-3D memory transfer volume

(b) Heat-3D memory bandwidth

Figure 12: Heat-3D memory performance

REFERENCES

[1] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. 2001. Optimal Semi-oblique Tiling. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01)*. 153–162.

[2] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. 2003. Optimal semi-oblique tiling. *IEEE Transactions on Parallel and Distributed Systems* 14, 9 (2003), 944–960.

[3] V. Bandishti, I. Pananilath, and U. Bondhugula. 2012. Tiling stencil computations to maximize parallelism *(SC '12)*. 1–11.

[4] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. 2010. Parameterized Tiling Revisited *(CGO '10)*. 200–209.

[5] 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 *(ICS '15)*. 197–206.

[6] W. Bielecki and P. Skotnicki. 2015. Concurrent Start Tiling of Stencil Computations based on the Transitive Closure of a Data Dependence Graph. *Przeglad Elektrotechniczny* R. 91, nr 11 (2015), 167–170.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer *(PLDI '08)*. 101–113.

[8] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures *(IPDPS '11)*. 676–687.

[9] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. 2009. Parallel data-locality aware stencil computations on modern micro-architectures *(IPDPS '09)*. 1–10.

[10] Hui-Min Cui, Lei Wang, Dong-Rui Fan, and Xiao-Bing Feng. 2010. Landing Stencil Code on Godson-T. *J. Comput. Sci. Technol.* 25, 4 (July 2010), 886–894.

[11] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures *(SC '08)*. Article 4, 12 pages.

[12] Chris Ding and Yun He. 2001. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems *(SC '01)*. 50–50.

[13] Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations *(ICS '05)*. 361–366.

[14] Matteo Frigo and Volker Strumpen. 2006. The cache complexity of multithreaded cache oblivious algorithms *(SPAA '06)*. 271–280.

[15] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* (May 2008), 1–25.

[16] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs *(CGO '14)*. 66–75.

[17] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles *(GPGPU-6)*. 24–31.

[18] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P. Sadayappan. 2014. The Relation Between Diamond Tiling and Hexagonal Tiling. *Parallel Processing Letters* 24, 03 (2014).

[19] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. 2015. MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures *(ICS '15)*. 177–186.

[20] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. 2009. Parametric Multi-level Tiling of Imperfectly Nested Loops *(ICS '09)*. 147–157.

[21] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan. 2010. DynTile: Parametric tiled loop generation for parallel execution on multicore processors *(IPDPS '10)*. 1–12.

[22] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A Stencil Compiler for Short-vector SIMD Architectures *(ICS '13)*. 13–24.

[23] S. Heybrock, B. JoÃş, D. D. Kalamkar, M. Smelyanskiy, K. Vaidyanathan, T. Wettig, and P. Dubey. 2014. Lattice QCD with Domain Decomposition on Intel Xeon Phi Co-Processors *(SC '14)*. 69–80.

[24] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance Code Generation for Stencil Computations on GPU Architectures *(ICS '12)*. 311–320.

[25] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. 2015. *Mono-parametric Tiling is a Polyhedral Transformation.* Research Report.

[26] F. Irigoin and R. Triolet. 1988. Supernode Partitioning *(POPL '88)*. 319–329.

[27] Guohua Jin, John Mellor-Crummey, and Robert Fowler. 2001. Increasing Temporal Locality with Skewing and Recursive Blocking *(SC '01)*. 43–43.

[28] Tian Jin, Nirmal Prajapati, Waruna Ranasinghe, Guillaume Iooss, Yun Zou, Sanjay Rajopadhye, and David G. Wonnacott. 2016. Hybrid Static/Dynamic Schedules for Tiled Polyhedral Programs. *CoRR* abs/1610.07236 (2016).

[29] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. 2010. An auto-tuning framework for parallel multicore stencil computations *(IPDPS '10)*. 1–12.

[30] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2006. Implicit and Explicit Optimizations for Stencil Computations *(MSPC '06)*. 51–60.

[31] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Multi-level Tiling: M for the Price of One *(SC '07)*. Article 51, 12 pages.

[32] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations *(PLDI '07)*. 235–244.

[33] Marcin Krotkiewski and Marcin Dabrowski. 2013. Efficient 3D Stencil Computations Using CUDA. *Parallel Comput.* 39, 10 (Oct. 2013), 533–548.

[34] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms *(ASPLOS IV)*. 63–74.

[35] Yulong Luo, Guangming Tan, Zeyao Mo, and Ninghui Sun. 2015. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-solution Space Model *(ICS '15)*. 187–196.

[36] Thibaut Lutz, Christian Fensch, and Murray Cole. 2013. PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems. *ACM Trans. Archit. Code Optim.* 9, 4, Article 59 (Jan. 2013), 24 pages.

[37] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes. 2015. Multicore-Optimized Wavefront Diamond Blocking for Optimizing Stencil Updates. *SIAM Journal on Scientific Computing* 37, 4 (2015), C439–C464.

[38] Tareq M. Malas, Georg Hager, Hatem Ltaief, and David E. Keyes. 2015. Multi-dimensional intra-tile parallelization for memory-starved stencil computations. *CoRR* abs/1510.04995 (2015). http://arxiv.org/abs/1510.04995

[39] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. 2012. Autotuning Stencil-Based Computations on GPUs *(CLUSTER '12)*. 266–274.

[40] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. 2011. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers *(SC '11)*. 1–12.

[41] A. C. McKellar and E. G. Coffman, Jr. 1969. Organizing Matrices and Matrix Operations for Paged Memory Systems. *Commun. ACM* 12, 3 (1969), 153–165.

[42] Jiayuan Meng and Kevin Skadron. 2009. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs *(ICS '09)*. 256–265.

[43] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 2010. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs *(SC '10)*. 1–13.

[44] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. 2014. A Study on Balancing Parallelism, Data Locality, and Recomputation in Existing PDE Solvers *(SC '14)*. 793–804.

[45] Irshad Pananilath, Aravind Acharya, Vinay Vasista, and Uday Bondhugula. 2015. An Optimizing Code Generator for a Class of Lattice-Boltzmann Computations. *ACM Trans. Archit. Code Optim.* 12, 2, Article 14 (May 2015), 23 pages.

[46] E. H. Phillips and M. Fatica. 2010. Implementing the Himeno benchmark with CUDA on GPU clusters *(IPDPS '10)*. 1–10.

[47] Manuel Prieto, Ignacio M. Llorente, and Francisco Tirado. 2000. Data Locality Exploitation in the Decomposition of Regular Domain Problems. *IEEE Trans. Parallel Distrib. Syst.* 11, 11 (Nov. 2000), 1141–1150.

[48] Fabrice Rastello and Thierry Dauxois. 2002. Efficient Tiling for an ODE Discrete Integration Program: Redundant Tasks Instead of Trapezoidal Shaped-Tiles *(IPDPS '02)*. 138–.

[49] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, and P. Sadayappan. Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs *(GPGPU '16)*. 92–102.

[50] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized Tiled Loops for Free *(PLDI '07)*. 405–414.

[51] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout, and Sanjay Rajopadhye. 2012. Parameterized Loop Tiling. *ACM Trans. Program. Lang. Syst.* 34, 1, Article 3 (May 2012), 41 pages.

[52] Gabriel Rivera and Chau-Wen Tseng. 2000. Tiling Optimizations for 3D Scientific Computations *(SC '00)*. Article 32.

[53] Rodrigo C. O. Rocha, Alyson D. Pereira, Luiz Ramos, and LuÃŋs F. W. GÃşes. 2017. TOAST: Automatic tiling for iterative stencil computations on GPUs. *Concurrency and Computation: Practice and Experience* (2017).

[54] Yonghong Song and Zhiyuan Li. 1999. New Tiling Techniques to Improve Cache Temporal Locality *(PLDI '99)*. 215–228.

[55] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. 2010. Cache oblivious parallelograms in iterative stencil computations *(ICS '01)*. ACM, 49–59.

[56] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. 2011. Cache Accurate Time Skewing in Iterative Stencil Computations *(ICPP '11)*. 571–581.

[57] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler *(SPAA '11)*. 117–128.

[58] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul A. Chowdhury. 2014. Improving Parallelism of Recursive Stencil Computations Without Sacrificing Cache Performance *(WOSC '14)*. 1–7.

[59] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul A. Chowdhury. 2015. Cache-oblivious Wavefront: Improving Parallelism of Recursive Dynamic Programming Algorithms Without Losing Cache-efficiency *(PPoPP 2015)*. 205–214.

[60] Didem Unat, Xing Cai, and Scott B. Baden. 2011. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C *(ICS '11)*. 214–224.

[61] Sundaresan Venkatasubramanian, Richard W. Vuduc, and none none. 2009. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems *(ICS '09)*. 244–255.

[62] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs *(SC '13)*. Article 25, 12 pages.

[63] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 12 (2001), 3 – 35. New Trends in High Performance Computing.

[64] Samuel Williams, Leonid Oliker, Jonathan Carter, and John Shalf. 2011. Extracting Ultra-scale Lattice Boltzmann Performance via Hierarchical and Distributed Auto-tuning *(SC '11)*. Article 55, 12 pages.

[65] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm *(PLDI '91)*. 30–44.

[66] M. Wolfe. 1989. More Iteration Space Tiling *(Supercomputing '89)*. 655–664.

[67] D. Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations *(IPDPS '00)*. 171–180.

[68] David Wonnacott. 2002. Achieving Scalable Locality with Time Skewing. *Int. J. Parallel Program.* 30, 3 (June 2002), 181–221.

[69] David G Wonnacott and Michelle Mills Strout. 2013. On the scalability of loop tiling techniques. *IMPACT 2013* (2013).

[70] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor *(ICPADS '12)*. 684–691.

[71] C. Yount and A. Duran. Effective Use of Large High-Bandwidth Memory Caches in HPC Stencil Computation via Temporal Wave-Front Tiling. In *(PMBS '16)*. 65–75.

[72] Yongpeng Zhang and Frank Mueller. 2012. Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters *(CGO '12)*. 155–164.

[73] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. 2012. Hierarchical Overlapped Tiling *(CGO '12)*. 207–218.

Artifact Description

We now provide and describe our codes. Our scheme is clear since it is based on a simple mathematical formulation. The following `kernel` can be any star or box stencil. `XSLOPE` and `YSLOPE` control the stencil size along each dimension. In this work we only implement the non-periodic boundary condition. However it's no hard to modify the code to handle stencils of periodic boundary conditions, which is our ongoing work. `level` controls the position of the two merged $(d+1)$-dimensional diamond, $\mathbb{B}_0 + \mathbb{B}_d$ from time tile of odd level to time tile of even level and $\mathbb{B}_d + \mathbb{B}_0$ from the opposite direction. `xright[level]` is the right of the leftmost $\mathbb{B}_0$ block in time tile of that level. `nb0[level]` is the number of $\mathbb{B}_0$ or $\mathbb{B}_d$ blocks.

Here is the 1D code based on our tessellation scheme.

```
int bx = atoi(argv[1]);
int bt = atoi(argv[2]);
int ix = bx + bx - 2 * bt * XSLOPE;
int xright[2] = {bx + XSLOPE,  bx + XSLOPE - ix / 2};
int nb0[2] = {(N + bx - (xright[0] - XSLOPE) - 1) / ix + 1,(N + bx - (xright[1] - XSLOPE) - 1) / ix + 1};

int level = 0;
int tt, n, t, x, xmin, xmax;

for (tt = -bt; tt < T ;  tt += bt ){
#pragma omp parallel for private(xmin,xmax,t,x)
for(n = 0; n < nb0[level]; n++) {
for(t = max(tt, 0) ; t < min( tt + 2 * bt,  T); t++){
xmin = max(     XSLOPE, xright[level] - bx + n * ix + myabs(t+1, tt+bt) * XSLOPE);
xmax = min( N + XSLOPE, xright[level]      + n * ix - myabs(t+1, tt+bt) * XSLOPE);
#pragma ivdep
for(x = xmin; x < xmax; x++){
update(t, x);}}}
level = 1 - level;}
```

Here is the 2D code based on our tessellation scheme.

```
int Bx = atoi(argv[1]);
int By = atoi(argv[2]);
int bt = atoi(argv[3]);
int bx = Bx-2*(bt*XSLOPE);
int by = By-2*(bt*YSLOPE);
int ix=Bx+bx;
int iy=By+by;
int xnb0=ceild(NX,ix);
int ynb0=ceild(NY,iy);
int xnb11=ceild(NX-ix/2+1,ix) +1;
int ynb11= ynb0;
int xnb12= xnb0;
int ynb12=1+ceild(NY-iy/2+1,iy);
int xnb2=max(xnb11,xnb0);
int ynb2=max(ynb12,ynb0);
int nb1[2] = {xnb12 * ynb12, xnb11 * ynb11};
int nb02[2] = {xnb2 * ynb2, xnb0 * ynb0};// B_0 and B_2 are merged to a 3-d diamond
int xnb1[2] = {xnb12, xnb11};
int xnb02[2] = {xnb2, xnb0};
int xleft02[2] = {XSLOPE-bx, XSLOPE+(Bx-bx)/2};
int ybottom02[2] = {YSLOPE-by, YSLOPE+(By-by)/2};
int xleft11[2] = {XSLOPE+(Bx-bx)/2, XSLOPE - bx};
int ybottom11[2] = {YSLOPE-(By+by)/2, YSLOPE};
int xleft12[2] = {XSLOPE-(Bx+bx)/2, XSLOPE};
int ybottom12[2] = {YSLOPE+(By-by)/2, YSLOPE-by};

int level = 1;
int tt, n, t, x, y, xmin, xmax, ymin, ymax;
```

```
for(tt = -bt; tt < T; tt += bt){
#pragma omp parallel for private(xmin,xmax,ymin,ymax,t,x,y)
for(n = 0; n < nb02[level]; n++){
for(t = max(tt,0); t < min( tt + 2*bt,  T); t++){
xmin = max(     XSLOPE,   xleft02[level] + (n%xnb02[level]) * ix      - bt*XSLOPE + abs(t+1, tt+bt) * XSLOPE);
xmax = min(NX + XSLOPE,   xleft02[level] + (n%xnb02[level]) * ix + bx + bt*XSLOPE - abs(t+1, tt+bt) * XSLOPE);
ymin = max(     YSLOPE, ybottom02[level] + (n/xnb02[level]) * iy      - bt*YSLOPE + abs(t+1, tt+bt) * YSLOPE);
ymax = min(NY + YSLOPE, ybottom02[level] + (n/xnb02[level]) * iy + by + bt*YSLOPE - abs(t+1, tt+bt) * YSLOPE);
for(x = xmin; x < xmax; x++) {
#pragma ivdep
for(y = ymin; y < ymax; y++){
update(t, x, y); }}}}
#pragma omp parallel for private(xmin,xmax,ymin,ymax,t,x,y)
for(n = 0; n < nb1[0] + nb1[1]; n++){
for(t = tt+bt ; t < min( tt + 2*bt,  T); t++) {
if(n<nb1[level]){
xmin = max(     XSLOPE,   xleft11[level] +                (n%xnb1[level]) * ix      - (t+1-tt-bt) * XSLOPE);
xmax = min(NX + XSLOPE,   xleft11[level] +                (n%xnb1[level]) * ix + bx + (t+1-tt-bt) * XSLOPE);
ymin = max(     YSLOPE, ybottom11[level] +                (n/xnb1[level]) * iy      + (t+1-tt-bt) * YSLOPE);
ymax = min(NY + YSLOPE, ybottom11[level] +                (n/xnb1[level]) * iy + By - (t+1-tt-bt) * YSLOPE); }
else{
xmin = max(     XSLOPE,   xleft12[level] + ((n-nb1[level])%xnb1[1-level]) * ix      + (t+1-tt-bt) * XSLOPE);
xmax = min(NX + XSLOPE,   xleft12[level] + ((n-nb1[level])%xnb1[1-level]) * ix + Bx - (t+1-tt-bt) * XSLOPE);
ymin = max(     YSLOPE, ybottom12[level] + ((n-nb1[level])/xnb1[1-level]) * iy      - (t+1-tt-bt) * YSLOPE);
ymax = min(NY + YSLOPE, ybottom12[level] + ((n-nb1[level])/xnb1[1-level]) * iy + by + (t+1-tt-bt) * YSLOPE); }
for(x = xmin; x < xmax; x++) {
#pragma ivdep
for(y = ymin; y < ymax; y++){
update(t, x, y); }}}}
level = 1 - level;}
```