Reducing Redundancy in Data Organization and Arithmetic Calculation for Stencil Computations

Kun Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
School of Computer Science and Technology, University of Chinese Academy of Sciences
Beijing, China
likungw@gmail.com

Liang Yuan*
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
yuanliang@ict.ac.cn

Yunquan Zhang
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
zyq@ict.ac.cn

Yue Yue
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
School of Computer Science and Technology, University of Chinese Academy of Sciences
Beijing, China
yyue1998@gmail.com

ABSTRACT

Stencil computation is one of the most important kernels in various scientific and engineering applications. A variety of work has focused on vectorization techniques, aiming at exploiting the in-core data parallelism. However, they either incur spatial data conflicts or hurt the data locality when integrated with tiling. In this paper, a novel spatial computation folding is devised to reduce the data reorganization overhead for vectorization and preserve the data locality for tiling in the data space simultaneously. We then propose an approach of temporal computation folding enhanced with shifts reusing, tessellate tiling, and semi-automatic code generation. It aims to further reduce the redundancy of arithmetic calculations and exploit the register reuse along the time dimension. Experimental results on the AVX2 and AVX-512 CPUs show that our approach obtains significant performance improvements compared with state-of-the-art techniques.

CCS CONCEPTS

• **Computing methodologies** → **Vector / streaming algorithms**;
• **Theory of computation** → **Vector / streaming algorithms**.

KEYWORDS

Stencil, Vectorization, Register reuse, Data locality

*Corresponding author.

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.
SC '21, November 14–19, 2021, St. Louis, MO, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476154

ACM Reference Format:
Kun Li, Liang Yuan, Yunquan Zhang, and Yue Yue. 2021. Reducing Redundancy in Data Organization and Arithmetic Calculation for Stencil Computations. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, St. Louis, MO, 13 pages. https://doi.org/10.1145/3458817.3476154

1 INTRODUCTION

Stencil is one of the most important kernels widely used across a set of scientific and engineering applications. It is extensively involved in various domains from physical simulations to machine learning [23, 33]. Stencil is also one of the seven computational motifs presented in the Berkeley View [3, 4, 48] and arises as a principal class of floating-point kernels in high-performance computing.

A stencil contains a pre-defined pattern that updates each point in d-dimensional spatial grids iteratively along the time dimension. The value of one point at time t is a weighted sum of itself and neighboring points at the previous time [37]. The naive implementation for a d-dimensional stencil contains $d + 1$ loops where the time dimension is traversed in the outmost loop and all grid points are updated in inner loops. Since stencil is characterized by this regular computational structure, it is inherently a bandwidth-bound kernel with a low arithmetic intensity and poor data reuse [21, 48].

Performance optimization of stencils has been exhaustively investigated in the literature. Traditional approaches have mainly focused on either vectorization or tiling schemes, aiming at improving the data parallelism and locality respectively. These two approaches are often regarded as two orthogonal methods working at different levels. Vectorization seeks to utilize the SIMD facilities in CPU to perform multiple data processing in parallel, while tiling tries to increase the reuse of a small set of data fit in cache.

Prior work on vectorization of stencil computations primarily falls into two categories. The first one is based on the associativity of the weighted sums of neighboring points. Specifically, the

execution order of one stencil computation can be rearranged to exploit common subexpressions [6, 10, 30, 31, 50]. Consequently, the number of load/store operations can be reduced and the bandwidth usage is alleviated in an optimized execution order. The second one attempts to deal with the spatial data conflict [15, 16], which is the main performance-limiting factor. The spatial data conflict is a problem caused by vectorization, where the neighbors for a grid point appear in the same vector register but at different positions. One milestone approach is the DLT method (Dimension-Lifting Transpose) [15], and it performs a global matrix transformation to address the spatial data conflict.

To address the problem of spatial conflicts, two common implementations are often adopted. The first one loads all the needed elements from memory in a vector form straightforward. Due to the low operational intensity, the stencil computation is often regarded as a memory-starving application. Compared with the scalar code, this multiple load vectorization method further increases the data transfer volume. Moreover, in each iteration of this code, it has at least two unaligned memory references where the first data address is not at a 32-byte boundary. Since CPU implementations favor aligned data loads and stores, these unaligned memory references will degrade the performance considerably.

The second solution is similar to the scalar code in terms of the CPU-memory data transfer. It loads each input element to vector register only once and assembles the required vectors via data permutation instructions. Compared with the multiple load method, this data permutation method reduces the memory bandwidth usage and takes advantage of the rich set of data-reordering instructions supported by most SIMD architectures. However, the execution unit for data permutations inside the CPU may become the bottleneck.

One milestone approach to address the spatial data conflicts is the DLT method [15]. In DLT the original one-dimensional array of length N is viewed as a matrix of size $vl^*(N/vl)$, where vl=4 for double-precision floats in a 256-bit vector. It then performs a global matrix transformation and assembles input vectors for calculating output vectors at the boundary. The DLT layout overcomes the input spatial data conflicts.

The following are some drawbacks of DLT. First, if we disregard the boundary processing, DLT can be considered as vl independent stencils. As a result, when combined with blocking frameworks, data reuse is reduced by vl times. The reason is that the vl independent stencils do not share data. Second, explicit transformation operations add overhead to DLT, which is particularly noticeable in stencils with higher orders or dimensions. The number of time loops in 1D stencils in scientific applications is frequently large enough to amortize the transpose overhead. However, the time size for 3D and high-dimensional stencils in other applications such as image processing is small, thus the overhead for global matrix transformation is unignorable. Finally, it's difficult to implement the DLT transpose in-place, so it's common to store the transposed data in a separate array. This increases the code's space complexity.

As one of the crucial techniques to exploit the parallelization and data locality for stencils, tiling, also known as blocking, has been widely studied for decades. Since the size of the working sets is generally larger than the cache capacity on a processor [24], the spatial tiling algorithms are proposed to explore the data reuse by changing the traversal pattern of grid points in one time step. However, such tiling techniques are restricted to the size of the neighbor pattern [21, 47]. Temporal tiling techniques have been developed to allow more in-cache data reuse across the time dimension [48].

The two aforementioned approaches of stencil computation optimizations often have no influence on the implementation of each other. However, the data organization overhead for vectorization may degrade the data locality. Moreover, most of the prior work only focuses on temporal tiling on the cache level. This only reduces the data transfer volume between cache and memory, and the high bandwidth demands of CPU-cache communication are still unaddressed or even worse with vectorization. Thus, the redundant calculation is performed on the same grid point iteratively due to massive CPU-cache transfers along the time dimension.

In this paper, we first design a novel computation folding strategy to overcome the input spatial data conflicts of vectorization and preserve the data locality for tiling simultaneously. The new vectorization scheme is formed with an improved fast in-CPU matrix transpose, which achieves the lower bounds both on the total number of data organization operations and the whole latency. Compared with conventional methods, the corresponding computation scheme for the new strategy requires no additional data organization operations, and the whole vectorized process is executed efficiently under considerable loads.

Based on the proposed strategy, a temporal computation folding approach is devised to reduce the redundancy of arithmetic calculations. We perform a deep analysis of the expansion for multiple time steps, fold the redundant operations on the same point, and reassign a new weight for it to achieve a multi-step update directly. An improved in-CPU flops/byte ratio is obtained by reusing registers, and the calculation of intermediate time steps is skipped over to alleviate the increased register pressure. The temporal computation folding approach could also be generalized for arbitrary stencil patterns. Furthermore, we utilize a shifts reusing technique to decrease the redundant computation within the innermost loops, integrate the proposed approach with a tiling framework to preserve the data locality, and design a semi-automatic code generator to simplify the parallel programming.

The proposed scheme is evaluated with AVX2 and AVX-512 instructions for 1D, 2D, and 3D stencils. The results show that our approach is obviously competitive with the classic vectorization methods (Auto Vectorization [38] and Data Reorganization [48]), state-of-the-art compilers (Pluto [5, 7] and SDSL [15, 16, 48]) and existing highly-optimized work (YASK[46] and Tessellation [47]).

This paper makes the following contributions:

- We propose an efficient computation folding strategy and corresponding vectorization scheme for stencil computation. The strategy utilizes a fast in-register transpose to eliminate spatial conflicts.
- Based upon the new proposed strategy, we design a temporal computation folding approach enhanced with shifts reusing, tessellate tiling and semi-automatic code generation. It aims to reduce the redundancy of arithmetic calculation in time iteration space.

- We generalize our approach on various kernels, and demonstrate that it could achieve superior performance compared to different highly-optimized work[5, 15, 16, 38, 46, 48] on multi-core processors.

The paper is organized as follows. First, Table 1 presents a brief sketch of the related terms used in this paper. Next, Section 2 introduces the relevant background and presents the existing work. Section 3 elaborates on the motivation of reducing spatial conflicts and formally describes the proposed computation folding strategy. Then the computation folding strategy is extended by exploiting register reuse to eliminate the arithmetical redundancy in time iteration space in Section 4. In Section 5, techniques for efficient implementation are discussed. Section 6 evaluate the performance exhaustively and Section 7 concludes the paper.

2 RELATED WORK

Research on optimizing stencil computation has been intensively studied [9, 20, 26, 36, 39], and it can be broadly classified as optimization methods to boost the computation performance, enhance the data reuse, and improve the data locality.

Vectorization by using SIMD instructions is an effective way to improve computation performance for stencils. Henretty proposes a new method DLT [15, 16] to overcome input spatial data conflicts at the expense of a dimension-lifting transpose, which makes it infeasible to perfectly utilize the tiling technique as a result of its spatially separated data elements [21]. Essentially DLT can be viewed as the combination of strip-mining (1-dimensional tiling) and out-loop vectorization [16]. In DLT the loop is transformed to a depth-2 loop nest where the size of the outer loop equals the vector length vl and the inner loop processes each subsequence of length N/vl. Note that the strip-mining was also introduced for vectorization [2]. However, the conventional usage is to make the size of the innermost loop be the vector length and substitute it by a vector code.

Data reuse has also been extensively recognized and exploited. Prior work [30, 31, 35, 50] on optimizing the order of execution instructions could decrease loads/stores operations to relieve the register pressure, while only the individual element in each vector could be reused. Basu designs a vector code generation scheme to reuse several vectors in the computation process, and it is constrained to constant-coefficient and isotropic stencils [6]. YASK [46] could improve data reuse by using common expression elimination and unrolling based on their vector-folding methods with fine-grained blocks [45], which is optimized only for high-order 3-dimensional stencils [50]. Zhao [50] designs a greedy algorithm to decide the part of the computation with high reuse, and groups reordered operators by using the same part as inputs to be computed with scatter operations. For other parts not identified by the algorithm, they still utilize the original computation by gather operations. Rawat [30] utilizes a DAG of trees with shared leaves to describe the stencil computation, and devises a scheduling algorithm to minimize register usage by reordering instructions on GPUs. Stock [35] proposes a framework to analyze the execution schedule of inputs in stencil computation. Reordering operations by using the associativity and commutativity are performed to relieve the increased register pressure. Nevertheless, the floating-point computations are not reduced by these approaches. Common subexpression elimination (CSE) [1] is presented to reduce the redundant computation in successive iterations of the same loop by reusing partial sums of a subexpression. This method relies heavily on loop unrolling to find specific expressions. Deitz extends the CSE method as Array Subexpression Elimination (ASE) [11] by creating an abstraction called a neighborhood tablet. Since the ASE reuses partial sums by subtablets via temporary variables, scalar dependences are newly introduced and hinder the instruction-level parallelization by compilers.

Tiling [17, 22, 25, 41, 42] is one of the most powerful transformation techniques to explore the data locality of multiple loop nests. Notably work for stencil computations includes hyper-rectangle tiling [12, 27, 29, 32], time skewed tiling [18, 34, 43], diamond tiling [5, 7], cache oblivious Tiling [13, 36, 37], split-tiling [16] and tessellating [47]. Wonnacott and Strout present a comparison on the scalability of many existing tiling schemes [44]. Most of these techniques are compiler transformation techniques and this paper integrated the new proposed layout with the tessellation scheme for simplifying the implementation. For stencil computations, a variety of auto-tuning frameworks [8, 14, 19, 49] have been presented by using varied hyper-rectangular tiles to exploit data reuse alone. However, redundant computations are involved in these work to resolve the introduced inter-tile dependencies that hinder the concurrent execution of shaped tiles on different cores.

Table 1: Terms

N	problem size of one-dimensional stencils
vl	maximum number of *double* elements a register can hold
VS	vector set, the building block composed of vl vectors
m	temporal unrolling factor
t	stamp at this time step
E	scalar arithmetic expression for a stencil
$C(E)$	the number of arithmetical instructions used in E
P	fraction of the cardinalities on $C(E)$ and $\mathbb{E}_\Lambda$
w	original pre-defined stencil weights
λ	reassigned stencil weights
v	vector register variable

3 SPATIAL COMPUTATION FOLDING

In this section, we first discuss the drawbacks of existing methods. Then we present a computation folding strategy and its corresponding vectorized process to eliminate the spatial conflicts when loading data.

3.1 Motivation

Normal vectorization leads to two drawbacks for stencil computations. First, there will be redundant references in one single vector computation. As shown in Figure 1 (a), to compute the next time step of (H, I, J, K), it requires (A, B, C, D), (B, C, D, E) and (C, D, E, F) first, which have common elements appearing at different positions. Thus, it will incur at least two unaligned vector loads.

Second, normal vectorization prevents the data reuse between continuous computations. In the scalar execution, two continuous

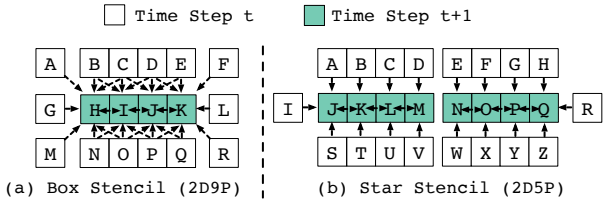

Figure 1: Spatial data conflicts in vectorization of 2D stencils.

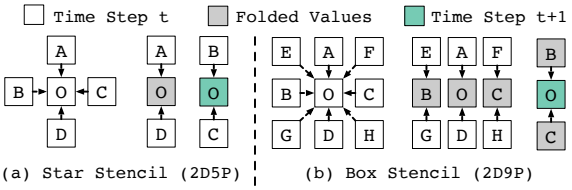

Figure 2: Illustration of computation folding strategy for scalar 2D stencils.

computations share 6 points and each new computation only require 3 loads and 1 store. The vectorized code has data conflicts and incurs 9 vector loads and 1 vector store.

Furthermore, these disadvantages worsen as the order or dimensionality of stencil increases. For example, each scalar point computation reuses 20 data points, loads 5 data points and stores 1 data point for a 2D25P stencil, while the vectorized code is able to reuse only 5 vectors and incurs 20 vector loads and 1 vector store. Thus the data redundancy is worse than that of the 2D9P stencil, i.e. $20/5 > 9/3$.

3.2 Scalar Folding

To preserve the data locality and reduce the number of data organization operations, we propose a folding strategy to address spatial conflicts. The idea is based on the observation that the data conflicts only appear in the innermost spatial loops for star stencils. Figure 1 (b) shows a 2D5P stencil. The two continuous vector computations of (J, K, L, M) and (N, O, P, Q) cause no data conflicts in the outer spatial dimension, i.e. (A, B, C, D) and (E, F, G, H) do not share data.

Our idea is to first calculate the stencil along outer spatial dimensions and perform the unit-stride loop computation after a data layout transformation that eliminates the data conflicts. We call the calculation along one dimension a *folding*. Figure 2 (a) illustrates the folding scheme for the 2D5P stencil with scalar computation. The vectorization implementation will be described in the next subsection. To update O, it first folds the neighbors A and D along the non-unit stride dimension (the column). The new O which is called a folded value is colored gray. The folded O is then updated with neighbors B and C in the unit-stride dimension (the row). Note that the last folding is performed with a transposed data layout where B and C can be viewed as neighbors in the non-unit stride dimension. This final result, O at the time step $t+1$ is colored green.

For a box stencil, there are data conflicts in all spatial dimensions as shown in Figure 2 (b). Nevertheless, the folding strategy is similar to that of star stencils. Figure 2 (b) illustrates the folding scheme for the 2D9P stencil. Additionally, four neighbors at corners are also folded vertically, such as E, G folded to B and F, H folded to C respectively. Then the three folded values B, O and C are merged to the final result. Our folding strategy is easily extended to higher-dimensional stencils. For a one-dimensional stencil, we view the array of N elements as a two-dimensional one of size $(vl) * (N/vl)$ and there is no data dependence along the outermost dimension. Therefore it only requires one folding operation with the transposed layout.

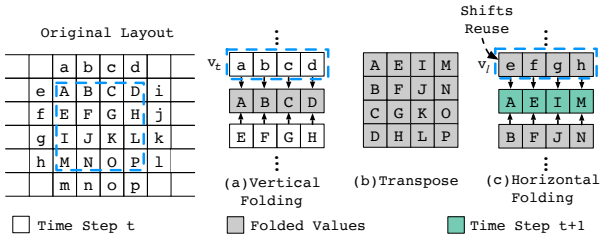

Figure 3: Vectorized process for 2D9P box stencil.

3.3 Vectorization

To adopt the folding strategy for vectorization, we group vl vectors as a *vector set*. The vector set is the basic processing granularity. Algorithm 1 presents the pseudo-code of the vectorization for 2D9P stencil with the folding strategy.

The program entry is contained in STENCIL function and it traverses the time loop. Indices of the two inner spatial loops are stepped by $vl = 4$. Thus, each iteration of x loop processes a row block and y loop splits each row block into vector sets. Line 10 to Line 15 performs on the first vector set VS_1 of each row block for boundary compuation. At first, VS_1 is loaded into registers by LoadVS (Line 10). The 16 elements A to P surrounded by a dashed line in the left of Figure 3 illustrate an example. Then the top and bottom vectors for this vector set are loaded to $v_t = (a, b, c, d)$ (Line 11) and $v_b = (m, n, o, p)$ (Line 12) by LoadVec, respectively. Since the original data layout is characterized by redundancy-free vertically, a set of *vertical folding* $VFoldVS$ is performed on VS_1 (Line 13) by VFOLDVS function, such as $(a, b, c, d) + (A, B, C, D) + (E, F, G, H) \rightarrow (A, B, C, D)$ as shown in Figure 3 (a). Now the vector set contains all four folded vectors updated with all neighbors in the same column and is transposed (Line 14) for horizontal folding in Figure 3 (b). Before entering the innermost loop, the left vector $v_l = (e, f, g, h)$ must be vertically folded in scalar style (Line 15).

The innermost loop pipelines the stencil compuation. Each iteration of the y loop loads a new vector set VS_2 and Lines 17 to 21 are identical to Lines 10 to 14 for the new vector set. The final update of the previous vector set VS_1 requires a *horizontal folding* $HFoldVS$ operation to gather the folded values based on the transposed layout in registers, such as the $(e, f, g, h) + (A, E, I, M) + (B, F, J, N) \rightarrow (A, E, I, M)$ as shown in Figure 3 (c). Though the computation physically resembles the $VFoldVS$, logically the elements are collected along each row. Thus we refer it as $HFoldVS$

Table 2: Analytical register behaviours for different methods on Jacobi stencils (per vector)

Kernel	1D5P				2D9P				3D27P				1D-Heat				2D-Heat				3D-Heat			
Operation[1]	C.	L.	S.	P.	C.	L.	S.	P.	C.	L.	S.	P.	C.	L.	S.	P.	C.	L.	S.	P.	C.	L.	S.	P.
AutoVec.[2]	5	5	1	0	9	9	1	0	27	27	1	0	3	3	1	0	5	5	1	0	7	7	1	0
Reorg.	5	1	1	4	9	3	1	6	27	9	1	18	3	1	1	2	5	3	1	2	7	5	1	2
S-Fold	5	1	1	0	5	1.5	1	1	15	4.5	1	3	3	1	1	0	5	1.5	1	1	7	2	1	1
S&T-Fold	4.5	0.5	0.5	0	5	1	0.5	0.5	10	2.5	0.5	1.5	2.5	0.5	0.5	0	5	1	0.5	1	10	1.5	0.5	2

[1] For better clarity, Computation, Load, Store, and Permute operations are abbreviated with C., L., S., and P. respectively.
[2] Methods for Auto Vectorization, Data Reorganization, Spatial Folding, and Spatial&Temporal Folding are also abbreviated accordingly (similarly hereinafter).

(Line 22). Note that the right vector to the $HFoldVS$ of VS_1 is the first vector in the transposed VS_2. Symmetrically, we reuse the last vector of VS_1 (Line 24) as the left vector for the computation of the next vector set.

Each vector set is stored with the transposed layout (Line 23 and 29) to the x-y swapped address. Thus each iteration of the outermost time loop actually performs a transpose to the whole data space if we ignore the computations in Algorithm 1. Figure 4 illustrates an example. The next iteration of the time loop works on the transposed array whose pointer and sizes are swapped (Line 31 and 32) and the vertical folding and horizontal folding are physically perform opposite ones. However, the code is still correct if the stencil pattern is symmetric along the diagonals. Otherwise one can simply add a code copy that manually swaps the $HFoldVS$ and $VFoldVS$ for even time iterations. In sum, the number of transpose operations is decreased by half with our folding strategy since it does not require the same data layout after each time iteration.

The spatial folding could also be applied to higher-order and higher-dimensional stencils in the same manner easily. The proposed vectorization scheme avoids data reloads compared with the multiple load method and fewer inter/intra-vector permutations compared with the data reorganization method. Generally, the in-register transpose only requires $vl \log(vl)$ in-lane instructions for a $vl \times vl$ matrix. Thus each vector computation incurs $\log(vl)$ permutations and the overhead is irrelevant to the order or dimensionality of stencils.

Table 2 lists the numbers of arithmetic, load, store and data reorganization operations. For the spatial computation folding (S-Fold) on the 2D9P kernel, Algorithm 1 loads 6 vectors and stores 4 vectors for updating a vector set in one iteration of the innermost loop. The total arithmetic operations are 5 per vector. For other stencil kernels, we can easily obtain similar results. Compared with the auto vectorization and data reorganization methods, spatial folding achieves fewer data accesses for all stencils and fewer computation instructions for higher-dimensional box stencils.

Algorithm 1 Vectorization with Folding Strategy for the 2D9P stencil. $T\%2 = 0$, $NX\%4 = 0$ and $NY\%4 = 0$

```
1: function VFoldVS(v0, v1, v2, v3, v4, v5)
2:     for i = 1 → 4 do
3:         v_{i-1} ← VFold(v_{i-1}, v_i, v_{i+1})
4:     end for
5:     v1, v2, v3, v4 ← v0, v1, v2, v3
6: end function
7: function Stencil( )
8:     for t = 1 → T do
9:         for x = 1 → NX by 4 do
10:            VS1 ← LoadVS(A, x, 1)
11:            v_t ← LoadVec(A, x − 1, 1)
12:            v_b ← LoadVec(A, x + 1, 1)
13:            VFoldVS(v_t, VS1, v_b)
14:            Transpose(VS1)
15:            v_l ← FoldandSetVec(A, x, 0)
16:            for y = 5 → NY by 4 do
17:                VS2 ← LoadVS(A, x, y)
18:                v_t ← LoadVec(A, x − 1, y)
19:                v_b ← LoadVec(A, x + 1, y)
20:                VFoldVS(v_t, VS2, v_b)
21:                Transpose(VS2)
22:                HFoldVS(v_l, VS1, VS2[0])
23:                Store(B, VS1, y − 4, x)
24:                v_l ← VS1[3]
25:                VS1 ← VS2
26:            end for
27:            VS2[0] ← FoldandSetVec(A, x,y)
28:            HFoldVS(v_l, VS1, VS2[0])
29:            Store(B, VS1, y − 4, x)
30:        end for
31:        A ↔ B
32:        NX ↔ NY
33:    end for
34: end function
```

4 TEMPORAL COMPUTATION FOLDING

4.1 Overview

In general, all grid points are only updated once before the round starts for the next time step in stencil computation. Although most

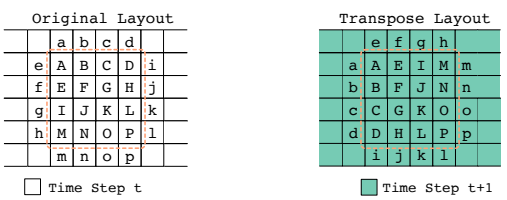

Figure 4: Optimized order of storage for 2D stencils.

of the existing work utilizes blocking technique [5, 47, 48] to decrease the data transfers between main memory and cache, there is no in-register data reuse between m successive time loops, where m is called the temporal unrolling factor. On the contrary, the straightforward implementation of reusing registers along the time dimension produces massive intermediate results at the time step t to $t + m$, which exacerbates excessive register spilling.

The existing work and straightforward implementation discussed above represent opposite extremes of register reusing in time iteration space. Our approach is to seek a balance that the redundancy of arithmetic calculation is eliminated along the time dimension, and register pressure is alleviated simultaneously. To facilitate the process of reducing redundant calculation in time iteration space, we extend the proposed spatial folding approach to update the grid points for m time steps directly in registers. The temporal computation folding is elaborated thoroughly based upon a profitability analysis, and then it is optimized with shifts reusing to obtain further performance gains.

4.2 Scalar Profitability Analysis

Figure 5 shows a scalar arithmetic expression for a representative 9-point box stencil with unrolling factor $m = 2$ on the center grid point. A collect $C(E)$ in Equation 1 is defined to describe the number of arithmetical instructions (add, multiply, multiply-add, etc.) used for 2-step updates in the expression E.

$$C(E) = \bigcup_{s} \{\langle g, w_g\rangle | \langle g, w_g\rangle \in C(E_s)\} \tag{1}$$

For the original expression in Figure 5(a), the center point with eight neighboring grid points are all updated to the state $t + 1$ first, and then these updates are swept from registers to memory. When the next iteration for $t + 2$ begins, these grid points of $t + 1$ are reloaded again. To obtain a 2-step update on the center point, the computing instructions of ten subexpressions are all counted into the collect $C(E)$. In each subexpression E_s, a pre-defined weight w_g is assigned on each grid point and then a 9-way addition result is obtained. Since nine distinct point references are engaged for each subexpression, we obtain a $|C(E)| = 10 \times |C(E_s)| = 90$ for the expression E. It is worth noting that redundant arithmetic operations are performed iteratively on the same point in different subexpressions, and store/reload operations incur additional interrupts during the computation process.

For the optimized expression in Figure 5(b), weights are all reassigned based on the m-step expansion. A new arithmetical expression $\mathbb{E}_\Lambda$ is determined by the folding matrix comprised of new weights λ. The five associative grid points of the same column are folded with λ first, and then a Horizontal Folding is performed to gather the obtained five folded values. Thus, the new collect in Equation 2 is 25, which is obtained from the computation folding on this point set with each grid point folded by λ_g.

$$C(\mathbb{E}_\Lambda) = \{\langle g, \lambda_g\rangle | \text{ grid } g \text{ used in } \mathbb{E}_\Lambda \text{ weighted with } \lambda_g\} \tag{2}$$

The profitable index is defined in Equation 3, and a profitable folding means the fraction of the cardinalities on two sets at least exceeds a threshold $\theta \geq 1$. In this case, it gives a net profitable index of $P(E, \mathbb{E}_\Lambda) = 90/25 = 3.6$ from Equation 3. Moreover, the interrupt cost of store/reload operations is also eliminated completely in $\mathbb{E}_\Lambda$.

$$P(E, \mathbb{E}_\Lambda) = \frac{|C(E)|}{|C(\mathbb{E}_\Lambda)|} \geq \theta \tag{3}$$

4.3 Vectorized Multi-step Computation

In this subsection, we take a 2-step 2D9P box stencil [47, 48] as an example to illustrate the details on our temporal folding approach in Figure 6, which is a vectorized process of the optimized arithmetical expression discussed in Section 4.2.

Data Preparation. Figure 6 depicts the codes and data preparation for the example stencils. Based on the previous strategy in Section 3, the basic granularity of temporal folding is also constructed as a 4×4 square of grid points denoted as s_o. They are loaded from cache to four registers as v_0 to v_3 respectively at time step t.

Vertical Folding. Similar to the manipulation in Section 3.3, Vertical Folding is performed first to collect neighbor points in the same column. A new square of grid points derived from s_o with Vertical Folding is called a counterpart. Typically a m-step update contains $m + 1$ counterparts at most. Equation 4 describes how each counterpart is obtained by Vertical Folding, where λ is the reassigned weight; superscripted n is the counterpart number.

$$v_i^{(n)} = \sum_{t=-m}^{m} \lambda_t^{(n)} \cdot v_{i+t} \tag{4}$$

For example, the weights for the first counterpart are reassigned as $\boldsymbol{\lambda}^{(1)}$={1, 2, 3, 2, 1} by the folding matrix shown in Figure 6. According to Equation 4, each $v_i^{(1)}$ in the first counterpart c_1 is calculated by performing a sum on v_{i-2}, $2v_{i-1}$, $3v_i$, $2v_{i+1}$, and v_{i+2}.

Horizontal Folding. With Vertical Folding completed, a local transpose is performed subsequently for further Horizontal Folding to collect the folded values in the same row:

$$v_i' = \sum_{t=-m}^{m} v_{i+t}^{(c-|t|)} \tag{5}$$

where c is the total number of counterparts. Since reassigned weights for the other two counterparts c_2 and c_3 are represented by $\boldsymbol{\lambda}^{(2)} = 2\boldsymbol{\lambda}^{(1)}$={2, 4, 6, 4, 2} and $\boldsymbol{\lambda}^{(3)} = 3\boldsymbol{\lambda}^{(1)}$={3, 6, 9, 6, 3}, the Equation 5 could be expanded as:

$$\begin{aligned} v_i' &= v_{i-2}^{(1)} + v_{i-1}^{(2)} + v_i^{(3)} + v_{i+1}^{(2)} + v_{i+2}^{(1)} \\ &= v_{i-2}^{(1)} + 2v_{i-1}^{(1)} + 3v_i^{(1)} + 2v_{i+1}^{(1)} + v_{i+2}^{(1)}. \end{aligned} \tag{6}$$

Thus, a coarse result s_c for 2-step updates on a point square s_o is obtained by only utilizing the square c_1.

Weighted Transpose. Horizontal folding is followed by a weighted transpose at last. Conventionally the stencil of Jacobi style is implemented with two arrays [7, 48], storing the value at odd and even time respectively. Therefore, the local transpose can be also optimized away here, and the result s_r could be organized to the original layout by the transpose in Horizontal Folding alternately. The $|C(\mathbb{E}_\Lambda)|$ in Equation 3 is further decreased to 9, and we obtain a profitable index $P(E, \mathbb{E}_\Lambda) = 10$ theoretically.

Table 2 also lists the analytical register behaviours for the spatial-temporal folding (S&T-Fold) in the last row. For the 2D9P kernel, it loads 9 vectors and stores 4 vectors for updating a vector set two

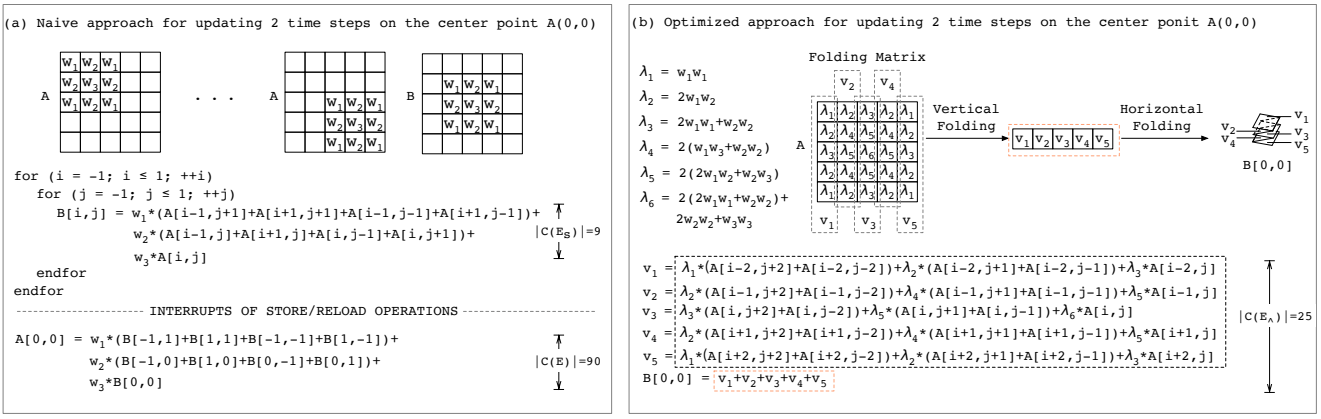

Figure 5: Illustration of scalar arithmetic expression for the 9-point box stencils with $m = 2$. The approach is used to try to minimize the collect $C(E)$ during the computation process.

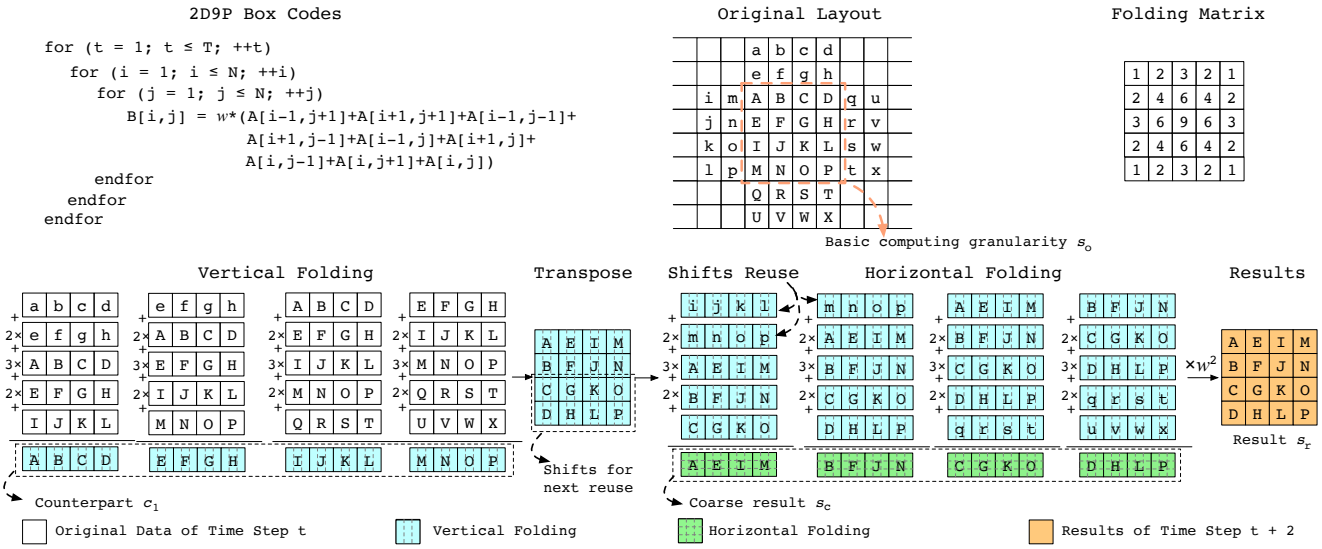

Figure 6: Vectorized process of computation folding approach for 9-point box stencils with unrolling factor $m = 2$.

time steps. Thus the total arithmetic operations are 4.5 per vector. The store instruction numbers are all decreased by half and the load operations are also diminished.

5 IMPLEMENTATION

5.1 Shifts Reusing

Figure 7 depicts a brief sketch of the scalar 1-step stencil computations between two adjacent grid points F and G in data space. It can be recognized from Figure 7 that there is potential for reusing shifts within the successive stencil computation from grid point F to G, and this gives us another reuse profitability of 2.25 by Equation 3. For our approach in Figure 6, the last two vectors of transposed counterpart c_1 in each iteration can be reused as shifts between computing squares. Therefore, the optimization of reducing reloads across squares is enabled by utilizing the same data collected in the last round as input to be computed together, which contributes to further performance gains. Similarly in Figure 3, the folded values in the last step can also reside in registers temporarily to avoid redundant arithmetical calculations and repetitive data transfers.

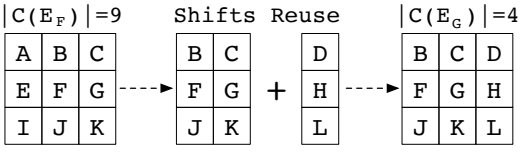

Figure 7: Illustration of shifts reusing.

5.2 Tessellate Tiling

Vectorization and tiling are two orthogonal methods and they target at different levels. Tiling serves to exploit the data reuse at cache levels, while vectorization boosts the computation using the data parallelism at the execution level. In our work, we choose the tessellation framework [47] to exploit the data reuse at cache levels. It provides a simple and clear parallel framework with lightweight loop conditions to allow fine in-core optimizations, which makes the integration easier than the compiler approaches. The benefits of our methods with tiling and parallelization scheme are demonstrated in Section 6.4.

5.3 Code Generation

Provided with a mathematical look into various cases, it is observed that all stencil kernels require the executions of identical and independent operations on vector registers, which poses challenges to efficient implementation on different architectures. Thus we notice that some loops can be created by a generator to reduce the code size and achieve performance portability. We abstract the algorithmic skeletons of the outer loops as a semi-automatic code generator. The recurring patterns of stencil expression are wrapped and generated by the code generator. For the characteristic descriptions of a stencil kernel, they are manually defined in a script file and exposed as parameters to the code generator. Then the output of the code generator is integrated into C code to simplify the parallel programming of stencils.

6 EVALUATION

In this section, we evaluate our proposed scheme on varied stencils used for real-world applications with AVX2 and AVX-512 instructions.

6.1 Setup

Machines. Experimental results presented in this paper have been obtained using two different machines. The first machine is composed of two Intel Xeon Gold 6140 processors with 2.30 GHz clock speed, which owns 36 physical cores organized into two sockets. DDR4 DRAM and 6 memory channels are supported, which yields a peak memory bandwidth of 127.5 GB/s. Each core contains a 32KB private L1 cache, a 1 MB private L2 cache, and a unified 24.75MB L3 cache. AVX-512 instruction set extension is supported and it's able to conduct operations for 8 double-precision floating-point data in a SIMD manner, which yields a theoretical peak performance of 73.6 GFlops/core (2649.6 GFlops in aggregate).

Table 3: Configuration for stencils used in experiments

Type	Pts	Problem Size	Blocking Size
1D-Heat	3	10240000×1000	2000×1000
1D5P	5	10240000×1000	2000×500
APOP	6	10240000×1000	2000×500
2D-Heat	5	5000×5000×1000	200×200×50
2D9P	9	5000×5000×1000	120×128×60
Game of Life	8	5000×5000×1000	200×200×50
3D-Heat	7	400×400×400×1000	20×20×10
3D27P	27	400×400×400×1000	20×20×10

The other machine consists of two AMD EPYC 7452 32-core processors running at 2.35GHz. Each core has a private 32KB L1 cache, a 512KB L2 cache, and a shared 16MB L3 cache. It features the AVX2 SIMD instruction set and a theoretical peak performance of 4812.8 GFLOPs. Concurrently, 8 DDR4 memory controllers provide a memory bandwidth of 204.8 GB/s.

Benchmarks. We first performed the sequential block-free experiments with three classic vectorization methods (Auto Vectorization [38], Data Reorganization [48], and DLT [15]) to investigate the absolute performance on a single process in Section 6.3. Then the newly related work (SDSL [16], Pluto [5], YASK[46] and Tessellation [47]) was employed for further comparison on multicore architecture in Sention 6.4. It is worth noting that SDSL and Tessellation are two related work extended with cache-blocking techniques on DLT and Auto Vectorization methods repectively. At last, the scalability was evaluated thoroughly compared with highly-optimized work and state-of-the-art compilers (SDSL [16], Pluto [5], YASK[46] and Tessellation [47]) in Section 6.5. The parallelization was inherently supported by OpenMP scheme in all above benchmarks, thus we utilized the OpenMP pragma *parallel for* on shared memory machines. All programs were compiled using the ICC compiler version 19.0.3, with the '-O3 -xHost -qopenmp -ipo' optimization flags.

Kernels. The detailed parameters for stencils used in experiments are described in Table 3, which consists of three star kernels (1D-Heat, 2D-Heat, and 3D-Heat) and three box kernels (1D5P, 2D9P, and 3D27P) corresponding to the references [16, 48]. Star and box equations are symmetric examples that can represent a wide variety of stencil kernels. Moreover, we also collect a series of classic kernels used in real-world applications [5, 28, 48]:

- APOP is a 1D3P stencil from two different input arrays to calculate the American put stock option pricing.
- The Game of Life is a cellular automaton proposed by Conway, and the update of each grid depends on all 8 of its neighbors.

The default value of total time steps ranges from 200 to 1000 in the references. Thus, we use a larger value of 1000 in our experiments, and the influence of it will be discussed in Section 6.3. Other parameters of each stencil are also fine-tuned based on reference work to guarantee that the peak performance for all methods could be reached exactly.

6.2 Impact of Data Preparation

In this subsection, the influence brought by data preparation is investigated first along the time dimension on Jacobi stencils. For DLT method, the major data preparation is the global dimension-lifted transformation, while in our computation folding strategy the in-register transpose substitutes. We employ the configurations in Table 3 and perform stencil computations with or without data preparations. Figure 8 shows the percentages of stencil computations ranging from 1 to 1,024 time steps.

As illustrated in Figure 8, our method could obtain a high computing density continuously. The in-register transpose makes little difference on the overall performance, which only decreases by 0.04% negligibly in 1D5P and 6.54% tolerably in 2D5P. However,

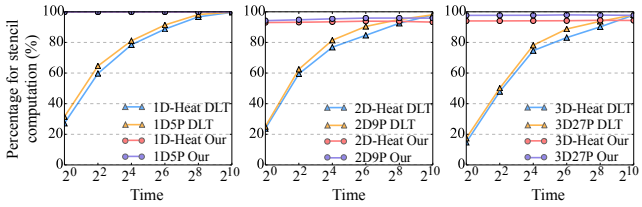

Figure 8: The percentages of stencil computations for Jacobi stencils in single-thread blocking-free experiments.

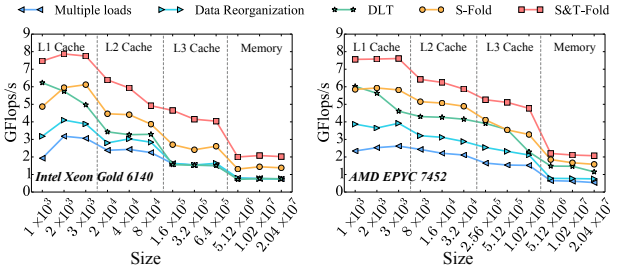

Figure 9: Absolute performance comparison for tested methods in single-thread blocking-free experiments. The results are shown separately with different machines.

DLT method is sensitive to dimension and time size distinctly. With lower dimensions or higher time steps, the percentage of stencil computation could reach a better result. This is primarily due to the data layout is relatively simple in lower dimensions and transformation cost is also diluted by long time size. The results also corroborate the fact that the number of time loops should be large enough to amortize the data preparation overhead in DLT method, which has been discussed in Section 1 previously. Therefore, a larger time size of 1,000 is used in our following experiments to leverage the full strengths of DLT. Besides the requirements of time size, an additional array is also required to store the transposed data by DLT, which increases the storage pressure.

6.3 Sequential Block-free Results

Then we present the performance results of varied methods across problem sizes ranging from L1 cache to main memory with a single thread. The cache-blocking technique is not applied for investigating the pure improvements brought by vectorization on various storage levels. The multiple loads and data reorganization methods represent a class of auto-vectorization in modern compilers and recent work [47, 48]. DLT is the dimension-lifting transpose approach designed by Henretty [15]. All the methods are implemented by hand-written codes optimized with the appropriate strategies to ensure fairness. First, we make the memory access aligned to 256-bit boundary. This ensures that the access will not cross cache lines, leading to performance reduction. Second, the loop unrolling is performed by four steps in innermost loops, which makes the most of available registers. Furthermore, though inplace implementation is supported by our computation folding strategy, two arrays are still used for storing the value at odd and even time respectively like other methods.

Figure 9 shows the performance comparison of our methods with the others. The results are illustrated separately in two subfigures by two machines. It can be seen that our S&T-Fold method outperforms others apparently in both machines, which demonstrates the effectiveness of the improvement of the flop/byte ratio. The S-Fold also achieves better performance than the DLT in most cases. The multiple loads method exhibits the worst performance among them due to the overhead caused by redundant loads. Furthermore, the performance drops apparently as the problem size moves from L1 cache to the memory hierarchy, which is mainly caused by the cost of data transfers.

Then we report the detailed results on the relative improvements of absolute performance on different storage levels in Table 4. The performance improvement of our methods is the largest one in each case, which is unconstrained to the storage level. This reflects the best performance again and corresponds to the results of Figure 9.

Table 4: Performance improvements on different storage level in single-thread blocking-free experiments

Methods	AutoVec.		Reorg.		DLT		S-Fold		S&T-Fold	
Machines	I.[1]	A.	I.	A.	I.	A.	I.	A.	I.	A.
L1 Cache	1.0x	1.0x	13x	1.4x	2.1x	2.2x	2.2x	2.2x	2.8x	3.0x
L2 Cache	1.0x	1.0x	1.1x	1.2x	1.4x	1.7x	1.8x	2.1x	2.5x	2.9x
L3 Cache	1.0x	1.0x	1.0x	1.3x	1.0x	1.8x	1.7x	2.1xx	3.0x	3.1x
Memory	1.0x	1.0x	1.0x	1.3x	1.0x	2.1x	1.8x	2.6x	2.7x	3.3x
Mean	1.0x	1.0x	1.1x	1.4x	1.4x	1.9x	2.0x	2.4x	2.8x	3.0x

[1] For better clarity, two targeted machines Intel Xeon Gold 6140 and AMD EPYC 7452 are abbreviated with I. and A. respectively.

6.4 Multicore Cache-blocking Experiments

In this subsection, we present the experiments that exhibit the benefits of our methods with cache-blocking techniques and parallelization scheme. We combine our vectorization scheme with tessellate tiling [48] and compare with the SDSL [16], Pluto [5], YASK[46] and Tessellation [47]. The techniques that benchmarks adopted for vectorization, cache-blocking, and parallelization are listed in Table 5.

Table 5: Techniques for vectorization, cache-blocking, and parallelization in benchmarks

Benchmarks	vectorization	Blocking	Parallelization
SDSL [16]	DLT [15]	Split tiling	OpenMP
Pluto [5]	AutoVec.	Diamond tiling [7]	OpenMP
Tesselation [47]	AutoVec.	Tessellate tiling [48]	OpenMP
YASK [46]	Vector Folding [45]	Loop tiling	OpenMP
Our	S&T-Fold	Tessellate tiling	OpenMP

Figure 10 and Figure 11 show the comparison for absolute performance and speedups of the different benchmarks optimized by

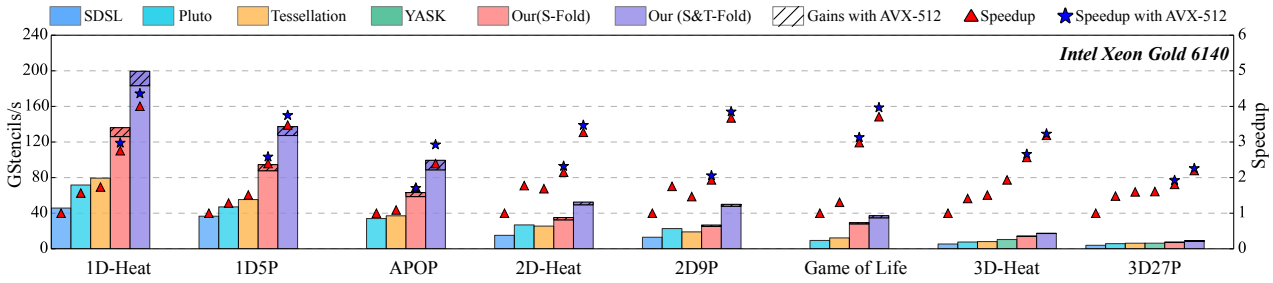

Figure 10: Performance and speedup comparison with cache-blocking on multicore Intel machine. The speedups of each group are compared to the lowest base which is annotated with the triangles by default value of 1.

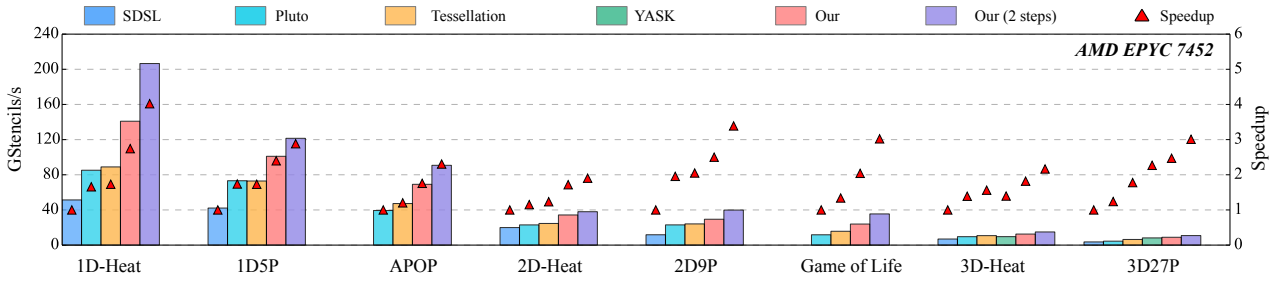

Figure 11: Performance and speedup comparison with cache-blocking on multicore AMD machine. The speedups of each group are compared to the lowest base which is annotated with the triangles by default value of 1.

the blocking techniques on two machines. Since some kernels are not supported by SDSL, the speedups of each group are relative to the base which is annotated with the speedup value of 1. Taking all stencils with AVX2 instructions into account, remarkable performance improvements are observed from our method with S&T-Fold, demonstrating that our vectorization scheme provides a significant benefit in a large problem size compared to the referenced work. Moreover, the optimization with AVX-512 instructions in Figure 10 could obtain further performance gains. The performance of SDSL is inferior to tessellation, which is resulted from the blocking technique constrained to its data layout. A closer look at Figure 10 and Figure 11 indicate that the performance is relative to the shape, dimension, and weight of the stencils. For star-shaped stencils, higher performance improvements are obtained compared to the box-shaped due to fewer neighbor points. For lower-dimensional stencils, much higher reuse is achieved on the loaded inputs, which exhibits better performance.

6.5 Scalability

We also evaluate the scalability of our schemes and benchmarks. The detailed parameters are given in Table 3, where all problem sizes exceed the L3 cache. As the experiments are performed across a broad variety of stencil kernels, some of them are not supported in all benchmarks. Since our tiling framework is the same as the tessellation, the performance improvements of our method with respect to it are fully derived from the vectorization.

It can be observed from Figure 12 and Figure 13 that our method could achieve the highest performance while the SDSL obtain the lowest performance. In 1D-Heat stencils, all these methods achieve nearly linear scaling on both instruction sets and the proposed temporal computation folding provides a significant improvement. With the increase of the problem dimension, the scalability for all methods drops as a result of the inherent complexity for multidimensional stencil computations. Similarly, the overall performance of high-order stencils also falls behind the corresponding low-order results, which is resulted from complex data access patterns in high-order stencils. Compared to the results implemented with AVX2 instructions, the performance of AVX-512 optimization on Intel machine shows a further increase.

6.6 Discussion

In this subsection, we provide an analysis of the performance on various configurations in previous experiments to tease out the contributions from different aspects of our proposed scheme.

We first investigate the impacts of data preparation on the DLT, which is a state-of-the-art vectorized method for stencils. Apparently, high performance is obtained by the cost of specific stencil parameters (low dimension or long time size), and that is not guaranteed in practical application. Our vectorized scheme, by contrast, could achieve data preparation more efficiently.

Sequential block-free experiments examine a variety of vectorization methods and demonstrate that S&T-Fold can achieve 2.8x and 3.0x improvements on Intel and AMD machines respectively compared with baseline method. Moreover, DLT method is more appropriate on the relatively small size, and this is partly explained

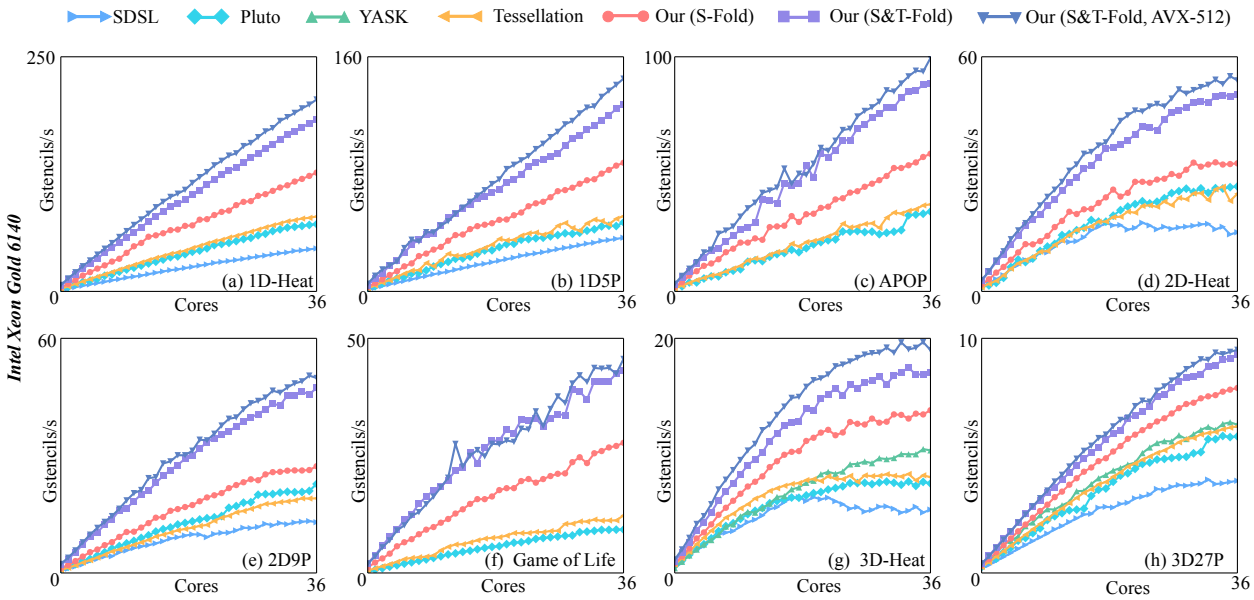

Figure 12: Scalability for stencils of various orders with different dimensions on multicore Intel machine.

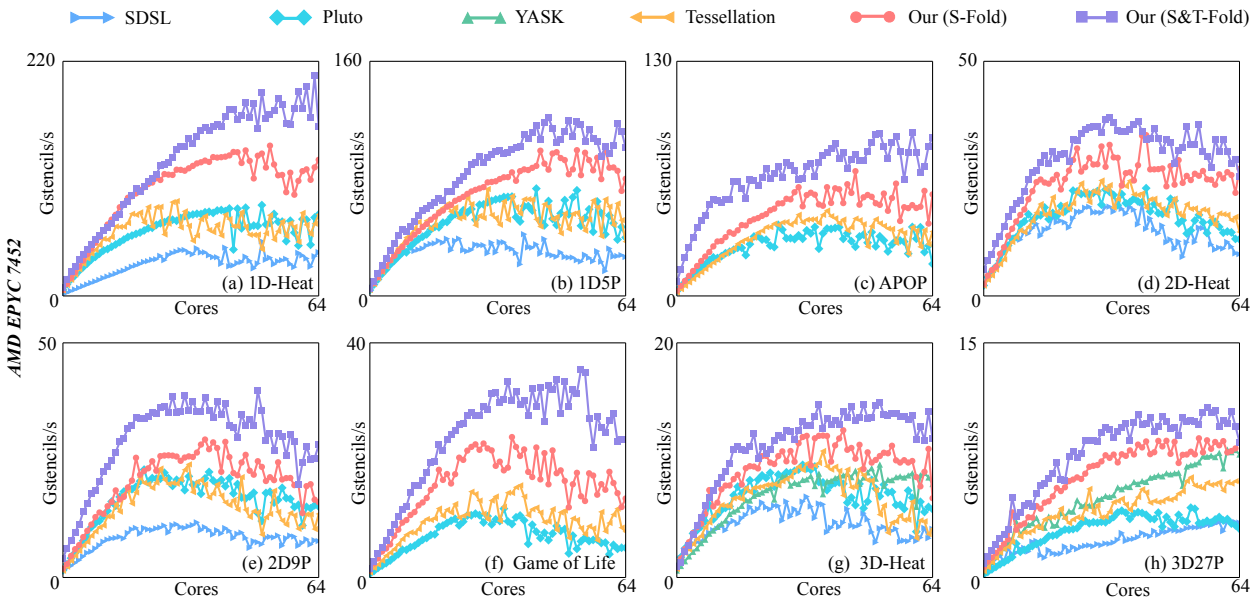

Figure 13: Scalability for stencils of various orders with different dimensions on multicore AMD machine.

by the less performance hit associated with additional dimension-lifting transpose in memory. With the problem size ranging from L1 cache to main memory, clear insights are provided that the overall performance trends drop consistently with the memory hierarchy on both machines. Furthermore, AMD machine obtains a slightly higher performance in the same memory hierarchy while Intel machine achieves better results in the same problem size. This is primarily contributed by the higher clock speed and smller cache size on AMD machine.

Multicore experiments conduct stencil cases on various benchmarks implemented with cache-blocking techniques.Typically, modern Intel machines contain AVX-512 instructions capable of performing on larger 512-bit registers, thus we also study the performance of our scheme on AVX-512 architectures. The results reveal

that they could contribute to better performance for our methods especially on 1D and 2D stencils. It is worth noting that the frequency reduction called throttling exists in CPUs when heavy AVX2 and AVX-512 extensions are involved. The slowdown is even worse with more cores employed. For example, the turbo frequency of the experimental Intel machine drops from 3.70 GHz to 3.00 GHz when active cores are expanded to full 18 on each processor [40]. The AVX-512 implementation has a further decrease to 2.10 GHz, which contributes to the mediocre performance in 3D stencils. The overall trends are in accord with the sequential block-free experiments, and our method with S&T-Fold outperforms others obviously.

The scalability experiments demonstrate that our vectorized scheme leveraging tessellate tiling outperforms the referenced benchmarks across a broad variety of configurations. Constrained to its specific data layout, SDSL is slower than other methods. Since multidimensional or high-order stencils are more compute-intensive, more dependency data are loaded into cache while they are not fully utilized to perform their own stencil computation. Thus, the overall performance for each method falls gradually with the increasing dimensions or orders. Furthermore, an interesting observation is the number of cores reaching peak performance on two machines. Intel machine obtains the best result with all cores active while AMD machine reach peak performance with slightly over half cores. Nevertheless, the real number of cores reaching the peak falls into the interval around 30 to 40 consistently on both two machines. As mentioned above, AVX throttling contributes to the frequency reduction with more active cores. Besides, more inter-core communication is also a performance-limiting factor as cores increase.

7 CONCLUSION

In this paper, we propose a novel spatial computation folding strategy to overcome the spatial data conflicts efficiently for vectorization in the data space. Then the temporal folding by reducing the redundancy of arithmetic calculations in time iteration space is presented on the basis of the proposed spatial approach. Furthermore, we describe how the proposed vectorization scheme is optimized with shifts reusing for enhancing data reuse and integrated with tessellate tiling for improving data locality. With the qualitative analysis and quantitative experiments, we demonstrate that significant performance improvements are achieved by our vectorization scheme over state-of-the-art products such as Intel's ICC and recent work [5, 15, 16, 38, 46, 48].

ACKNOWLEDGMENTS

The authors would like to thank all the reviewers for their insightful and valuable comments and suggestions. This work is supported by the the National Key Research & Development Program of China (2016YFB0200800), National Natural Science Foundation of China under Grant No. 61972376, No. 62072431 and No. 62032023, the Science Foundation of Beijing No. L182053.

REFERENCES

[1] Alfred V Aho, Stephen C Johnson, and Jeffrey D Ullman. 1976. Code generation for expressions with common subexpressions. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*. 19–31.

[2] Randy Allen and Ken Kennedy. 2002. *Optimizing compilers for modern architectures: a dependence-based approach*. Taylor & Francis US.

[3] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. The landscape of parallel computing research: A view from berkeley. (2006).

[4] Krste Asanovic, Ras Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John D Kubiatowicz, Edward A Lee, Nelson Morgan, George Necula, David A Patterson, et al. 2008. The parallel computing laboratory at UC Berkeley: A research agenda based on the Berkeley view. *EECS Department, University of California, Berkeley, Tech. Rep* (2008).

[5] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[6] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella. 2015. Compiler-Directed Transformation for Higher-Order Stencils. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 313–323.

[7] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.

[8] 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.

[9] 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. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–12.

[10] Raúl de la Cruz and Mauricio Araya-Polo. 2014. Algorithm 942: Semi-Stencil. *ACM Trans. Math. Softw.* 40, 3, Article 23 (April 2014), 39 pages. https://doi.org/10.1145/2591006

[11] Steven J Deitz, Bradford L Chamberlain, and Lawrence Snyder. 2001. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th international conference on Supercomputing*. 65–77.

[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] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. 2015. Modesto: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 177–186.

[15] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. 2011. Data layout transformation for stencil computations on short-vector simd architectures. In *International Conference on Compiler Construction*. Springer, 225–245.

[16] 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 (ICS '13)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2464996.2467268

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

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

[19] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

[20] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2006. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*. 51–60.

[21] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. *SIGPLAN Not.* 42, 6 (June 2007), 235–244. https://doi.org/10.1145/1273442.1250761

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

[23] Kun Li, Honghui Shang, Yunquan Zhang, Shigang Li, Baodong Wu, Dong Wang, Libo Zhang, Fang Li, Dexun Chen, and Zhiqiang Wei. 2019. OpenKMC: a KMC design for hundred-billion-atom simulation using millions of cores on Sunway Taihulight. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[24] Tareq M. Malas, Georg Hager, Hatem Ltaief, and David E. Keyes. 2017. Multidimensional Intratile Parallelization for Memory-Starved Stencil Computations. *ACM Trans. Parallel Comput.* 4, 3, Article Article 12 (Dec. 2017), 32 pages. https://doi.org/10.1145/3155290

[25] 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.

[26] Jiayuan Meng and Kevin Skadron. 2009. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*. 256–265.

[27] 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.

[28] C.S. Department of University of Oregon. 2014. Stencil Pattern. https://ipcc.cs.uoregon.edu/lectures/lecture-8-stencil.pdf [Online; accessed 29-July-2020].

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

[30] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 168–182.

[31] P. S. Rawat, A. Sukumaran-Rajam, A. Rountev, F. Rastello, L. Pouchet, and P. Sadayappan. 2018. Associative Instruction Reordering to Alleviate Register Pressure. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 590–602.

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

[33] Aaron Sawdey, Matthew O'Keefe, Rainer Bleck, and Robert W Numrich. 1995. The design, implementation, and performance of a parallel ocean circulation model. In *Proceedings of 6th ECMWF Workshop on the Use of Parallel Processors in Meteorology: Coming of Age*. 523–550.

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

[35] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 65–76.

[36] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. 2010. Cache oblivious parallelograms in iterative stencil computations. In *Proceedings of the 24th ACM International Conference on Supercomputing*. 49–59.

[37] 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 (SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1989493.1989508

[38] Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, and Ernesto Su. 2002. Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal* 6, 1 (2002).

[39] Sundaresan Venkatasubramanian, Richard W Vuduc, and none none. 2009. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd international conference on Supercomputing*. 244–255.

[40] Wikichip.org. 2019. Wikichip of Intel Xeon Gold 6140. https://en.wikichip.org/wiki/intel/xeon_gold/6140 [Online; accessed 29-July-2020].

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

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

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

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

[45] Charles Yount. 2015. Vector Folding: improving stencil performance via multi-dimensional SIMD-vector representation. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 865–870.

[46] 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.

[47] Liang Yuan, Shan Huang, Yunquan Zhang, and Hang Cao. 2019. Tessellating Star Stencils. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[48] Liang Yuan, Yunquan Zhang, Peng Guo, and Shan Huang. 2017. Tessellating Stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. Association for Computing Machinery, New York, NY, USA, Article Article 49, 13 pages. https://doi.org/10.1145/3126908.3126920

[49] Yongpeng Zhang and Frank Mueller. 2012. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 155–164.

[50] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. 2019. Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–44.