An Efficient Vectorization Scheme for Stencil Computation

***Abstract*—Stencil computation is one of the most important kernels in various scientific and engineering applications. A variety of work has focused on vectorization and tiling techniques, aiming at exploiting the in-core data parallelism and data locality respectively. In this paper, the downsides of existing vectorization schemes are analyzed. Briefly, they either incur data alignment conflicts or hurt the data locality when integrated with tiling. Then we propose a novel transpose layout to preserve the data locality for tiling and reduce the data reorganization overhead for vectorization simultaneously. To further improve the data reuse at the register level, a time loop unroll-and-jam strategy is designed to perform multistep stencil computation along the time dimension. Experimental results on the AVX2 and AVX-512 CPUs show that our approach obtains a competitive performance with the classic vectorization methods (Auto Vectorization and Data Reorganization), state-of-the-art compilers (Pluto and SDSL), and highly-optimized work (DLT and Tessellation).**

***Index Terms*—Stencil, Vectorization, Data locality, Data alignment conflict**

I. 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 [22]. Stencil is also included as one of the seven computational motifs presented in the Berkeley View [3], [36] 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 a d-dimensional spatial grid iteratively along the time dimension. The stencil's order [36], [38] defines the dependent relationship in a certain direction. If the order of a symmetric stencil in one dimension is r, the value of one point at time t is a weighted sum of $(2r+1)$ points at the previous time [30]. 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], [35].

Performance optimizations of stencils have been exhaustively investigated in the literature [10], [11], [23]. Traditional approaches have mainly focused on either vectorization or tiling schemes, aiming at improving the in-core data parallelism and the data locality in cache 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. They actually complement each other and can be subtly combined.

Prior work on vectorization of stencil computation 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 sub-expression or data reuse at register or cache level [6], [25], [27], [38]. Consequently, the number of load/store operations can be reduced and the bandwidth usage is alleviated in optimized execution order. The second one attempts to deal with the data alignment conflicts [16], [17], which is the main performance-limiting factor. The data alignment conflict is a problem caused by vectorization. Since the data elements are stored contiguously in memory, the neighbors for each element are loaded into different slots in the same register by using vector operations. Thus, stencil computation for each element requires the use of either redundant and unaligned load operations from memory, or frequent inter-register and intra-register data permutations, to make adjacent elements remapped to the same slots in different registers. One milestone approach to address the data alignment conflict is the Dimension-Lifting Transpose (DLT) method [16]. We will present a deep discussion on them in the next section.

As one of the crucial transformation 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 working sets in stencil-based applications is generally larger than the cache capacity on a processor, the spatial tiling algorithms are proposed to explore the data reuse by changing the traversal pattern of grid points in one time step. Generally, a grid point in cache is utilized to perform stencil computation for all its neighbors before swapped out cache. Thus, the data transfers between the cache and main memory could be reduced. However, the improvement of such tiling techniques is restricted to the size of the neighbor pattern [21], [36]. Temporal tiling techniques have been developed to allow more in-cache data reuse across the time dimension.

The aforementioned two approaches of stencil computation optimizations often have no influence on the implementation of each other. The fundamental reason is that the vectorization typically applies to the innermost loop. Therefore, integrating one technique of vectorization with another tiling scheme is often straightforward. However, the data organization overhead for vectorization may degrade the data locality. Furthermore, to the best of our knowledge, most of the prior work only focuses on temporal tiling technique on the cache level. This only opti-

mizes the data transfer volume between cache and memory and the high bandwidth demands of CPU-cache communication is still unaddressed or even worse with vectorization. We will present a deep discussion of these two problems in the next section.

In this paper, we first design a novel transpose layout to overcome the input data alignment conflicts of vectorization. The new layout is formed with an improved in-CPU matrix transpose scheme, 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 layout requires less data organization operations, whose cost can be further overlapped by arithmetic calculations. To enhance the data reuse on the register level, we then propose an approach to perform multiple time steps for stencil computations. The in-register data can be reused to perform successive updates along the time dimension, which has not explored in existing work. Finally, we integrated the proposed layout with a tiling framework. It only requires a slight modification of the new vectorization scheme to preserve the data reuse ability of tiling. The proposed vectorization 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 [31] and Data Reorganization [36]), state-of-the-art compilers (Pluto [5], [7] and SDSL [17]) and highly-optimized work [16], [35].

This paper makes the following contributions:

- We propose an efficient transpose layout and corresponding vectorization scheme for stencil computation. The layout transformation utilizes an improved matrix transpose of the lowest latency.
- We exploit the in-register data reuse by performing multiple time step computation based upon the new proposed transpose layout.
- An integrated approach is proposed to perform a tiling framework in conjunction with the vectorization scheme.
- We demonstrate that the proposed approach could achieve superior performance compared to several highly-optimized stencil benchmarks on multi-core processors.

The paper is organized as follows. Section 2 presents the relevant background and elaborates on the addressed problem. Section 3 introduces the proposed vectorization scheme and the tiling technique. Section 4 provides experimental results that demonstrate our approach produces a higher performance compared to the benchmarks. In Section 5, we present the related work and Section 6 concludes the paper.

II. Background

A. Data Alignment Conflicts of Vectorization

As stated earlier in Section I, the input data alignment conflict incurred by vectorization is a crucial performance-limiting factor in stencil computation. In this subsection, we take the 1D3P stencil as an example to illustrate this fundamental problem caused by vectorization. Since in most

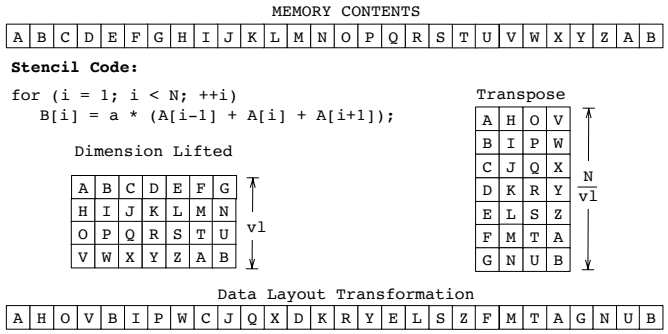

Fig. 1. Data alignment conflicts handling in DLT.

existing work vectorization is restricted to innermost loops [9], the codes shown in Figure 1 only illustrates the 1D3P stencil of one time step.

In the i-th iteration of this scalar code execution, it loads $A[i+1]$ and $B[i]$ to registers and reuses register data $A[i-1]$ and $A[i]$ referenced by the previous calculation of $B[i-1]$. Observing the CPU-memory data transfer, this code is exactly similar to a common array copy code, i.e. the `memcpy` function [12]. The computation implementation inside CPU is straightforward. Loop optimizations like loop unrolling also preserve these properties.

The vectorization groups a set of data in a vector register and processes them in parallel. The naive vectorization of the 1D3P stencil code computes contiguous elements in the output array B. Assume the vector register holds 4 elements (vector length $vl = 4$), the vectorization code performs the calculation using vector operations and output $(B[1], B[2], B[3], B[4])$ with one vector register.

A well-known problem incurred by the vectorization of stencil codes is the input data alignment conflicts. For example, to compute $(B[1], B[2], B[3], B[4])$, it requires three vectors: $(A[0], A[1], A[2], A[3])$, $(A[1], A[2], A[3], A[4])$ and $(A[2], A[3], A[4], A[5])$. The element $A[2]$ appears in all these vector registers but at different positions. We call this a data alignment conflict. Thus there is no corresponding simple execution as the scalar code.

To address the data alignment problem, two common implementations are often adopted. The first one loads all the needed elements from memory in a vector form straightforward, which is adopted as Auto Vectorization by modern compilers. 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 inter-register and intra-register data permutations instructions. Compared with the multiple load method, this data permutations method reduces the memory bandwidth usage and takes the advantages 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.

B. Dimension-Lifting Transpose (DLT)

One milestone approach to address the data alignment conflict is the DLT method [16]. In DLT the original one-dimensional array of length N is viewed as a matrix of size vl*(N/vl), where vl is the vector length in vector elements. For example, vl=4 for double-precision floats in a 256-bit vector. It then performs a global transpose. Figure 1 illustrates the DLT method for a one-dimensional array of 28 elements. The DLT layout overcomes the input data alignment conflicts. For instance, the second $vl = 4$ elements in the transformed layout are formed into one vector (B[1], B[8], B[15],B[22] and all the three required input vectors: left vector (A[0], A[7], A[14], A[21]), center vector (A[1],A[8],A[15],A[23]) and right vector (A[2], A[9], A[15], A[23]) are free of data sharing and stored contiguous in memory. DLT needs to assemble input vectors for calculating output vectors at boundary.

DLT has the following disadvantages. First, DLT can be viewed as vl independent stencils if we ignore the boundary processing. Therefore when incorporated with blocking frameworks, the data reuse decreases vl times. The reason is that there is no data reuse among the vl independent stencils. Second, DLT suffers from the overhead of explicit transpose operations executed before and after the stencil computation. For 1D and 2D stencils in scientific applications, the number of time loops is often large enough to amortize the transpose overhead. But for 3D and higher-dimensional stencils in other applications like image processing, the time size is small that makes the transpose overhead unignorable. Finally, it's hard to implement the DLT transpose in-place and it often chooses to use an additional array to store the transposed data. This increases the space complexity of the code.

III. The Transpose Layout

In this section, we first discuss the drawbacks of existing vectorization methods and explain the distinctiveness of our methods. Then we present a new transpose layout and its corresponding stencil computation scheme. Next, we present several further optimizations on the transpose layout including the extension to multiple time steps, integration with a tiling framework and an improved matrix transpose algorithm for double precision floating-point numbers.

A. Motivation

From the hardware perspective, the critical approach to boost performance is to fully utilize the execution units that perform the arithmetic instructions. Since there is no data dependence in one time-step iteration of stencil computations, the only bottleneck is data preparation. Equivalently, the key technique of vectorization is to address the data alignment conflicts.

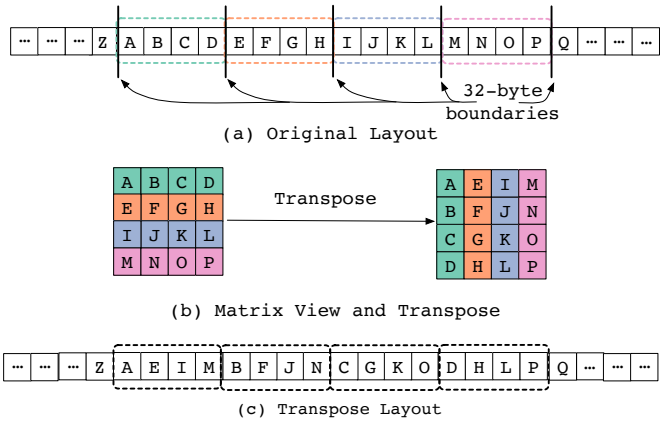

Fig. 2. Register Transpose Layout for SIMD vector length of 4.

Our starting point is the observation of the disadvantages of existing methods. The DLT is a promising method that extremely reduces the data reorganization operations. However, essentially the DLT vectorization format hurts the locality properties as mentioned above. In particular, the elements in one vector are distant, thus there is no data reuse among them. On the contrary, the straightforward multiple load and data reorganization methods load contiguous element in one vector. They lead to the optimal data locality when integrated with a temporal tiling scheme. These two methods seem to be at two extreme ends of a balance between the number of reorganization operations of data in CPU and the reuse ability of data in cache. Our scheme proposed in Section III-B seeks to preserve the "in-register" data locality by taking the advantages of the rich register assembling instruction set and efficient implementations in modern CPUs.

Then the temporal tiling on register level in Section III-C is devised based on our proposed scheme. To the best of our knowledge, in-core exploitation of multiple time steps of stencil computations hasn't been considered in existing work. Experimental results also show a significant improvement with this strategy.

Moreover, it's inefficient to integrate DLT with a temporal tiling technique, especially for high-dimensional stencils as exhibited in SDSL [17]. Besides the data locality damage, DLT is inappropriate to temporal tiling due to the boundary processing and transformation layout. Our method reserves the original data layout for the natural data locality of stencil computations with an integrated tiling framework in Section III-D.

The existing work has shown that the global matrix transformations, which are required by DLT, are time-consuming and demand extra memory quota. We propose an efficient in-core transpose scheme in Section III-E and the reorganizations are executed on-the-fly with stencil computations. Consequently, the cost is insignificant.

B. Locally Transpose

To preserve the data locality and reduce the number of data organization operations, we apply a matrix transpose to a

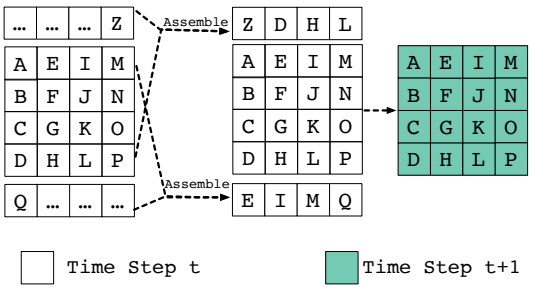

Fig. 3. Illustration of stencil computation for transpose layout.

small sub-sequence of contiguous elements. Specifically, like the dimension-lifting approach in DLT, the one-dimensional view to the sub-sequence is substituted by a two-dimensional matrix view. To perform vectorization after a matrix transpose, the column size of the matrix should be equal to the vector length vl. Let the row size be m, the size of the matrix is then $vl * m$. Our locally transpose layout is equivalent to the DLT when $m = N/vl$ and the original data layout when $m = 1$.

After the matrix transpose, it still requires some data reorganizations for computing the first and last one of the m vectors. For example, if $m = 1$, the original data preparations of the left and right vectors must be done for computing each output vector. This is the trade-off between data locality and the number of data preparations explained above.

There are several considerations for deciding the size m. First, m should be large enough to hide the overhead of the data reorganizations for the first and last vectors by the actually arithmetic operations of the middle vectors. Assume the order of a stencil is r, then the number of arithmetic operations of the middle vectors is $(2r+1)*(m-1)+1$. The number of data operations is $4r$ since the first and last vectors need $2r$ vectors and assembling each of them requires two reorganization instructions as will be explained shortly. Thus m should be at least 3. Notice that this limitation is irrelevant to the order r. Second, to avoid an additional array that is needed to store the transposed data as in the DLT format, it's desirable to complete the matrix transpose in CPU. Thus the m input vectors and additional auxiliary vectors must be kept in the CPU vector register file. In this work we always set $m = vl$. The final reason is that transposing a matrix of size $vl * vl$ is easier to implement on modern CPU products. We will present a highly efficient algorithm for matrix transpose of size $vl * vl$ later.

Figure 2 illustrates the transpose layout for a one-dimensional stencil with a vector length of four. The matrix transposes of every sub-sequence of $vl*vl$ length is performed before and after the stencil computation. In the rest of the paper, we also refer to the vl vectors as a **vector set** (VS). Note that in the implementation a vector set is always aligned to a 32-Byte boundary.

The update of one vector set of the 1D3P stencil requires two assembled vectors. One is the left dependent vector of its first vector and the other is the right dependent vector of its last vector. Figure 3 describes the data reorganization of these two vectors. The first vector is (A, E, I, M) and its left dependent vector is (Z, D, H, L) which is stored in two distant vectors in the transpose layout, $(*, *, *, Z)$ and $(D, H, L, *)$. These two vectors are combined by a blend instruction followed by a permute operation to shift the components to the right circularly.

The stencil computations of the vector set are straightforward as shown in Figure 3. We then achieve an efficient vectorization scheme by performing lower-overhead matrix transpose and two data operations per vector set. Moreover, the proposed vectorization scheme avoids data reloads compared with the multiple load method and frequent inter-vector permutations compared with the data reorganization method. The transpose layout could also be applied to higher-order and multidimensional stencils in the same manner.

C. Unroll-and-jam the Time Loop

In general, stencil computation is restricted to its input data alignment conflicts, and all elements are only updated once before the round starts in the next time step. Although blocking technique [5], [35], [36] can be utilized to decrease the data transfers between main memory and cache, there is no in-register data reuse between successive time loops. Therefore the in-CPU flops/byte ratio is limited by the stencil pattern. To the best of our knowledge, computation for multiple time steps in registers is not explored in existing work.

We develop a in-register unroll-and-jam strategy of time loops based on locally transpose in Section III-B. It loads one element at time dimension t and updates it k time steps before storing it to memory. k is called the unrolling factor. The normal execution corresponds to the case of $k = 1$. If $k > 1$ the execution is equivalent to unroll the time loop k times and jam them. Consequently, it improves the in-CPU flops/byte ratio by k times. A one-dimensional example is illustrated in this subsection and the strategy is also applicable to multidimensional stencils.

Overall the algorithm is straightforward. After updating one vector set, we keep the result in registers and process the next neighbor vector set. Then the current vector set can be forwarded along time dimension one more step using the new value of the right neighbor.

Algorithm 1 shows the pseudo-code of our multiple time steps updating scheme. The COMPUTE function receives a set of vl vectors and their dependent vectors that are assembled by the ASSEMBLE function. It computes the elements in the vector set by one time step. Notice that this is an in-place updating that the value of last time will be overwritten.

The main function traverses the time loop stepped by the unrolling factor k. For simplicity, we assume T is divisible by k. In each iteration of the while loop, every element is forwarded k steps along the time dimension. The booting computation prepares the data at head needed by the following pipelined updating. The top part of Figure 4 illustrates the case of $k = 2$ after a booting computation. The vector sets $\mathbf{VS}_1$ to $\mathbf{VS}_k$ have been updated $k-1$ to 0 times, respectively. Due to the overwriting property of the COMPUTE function, it needs

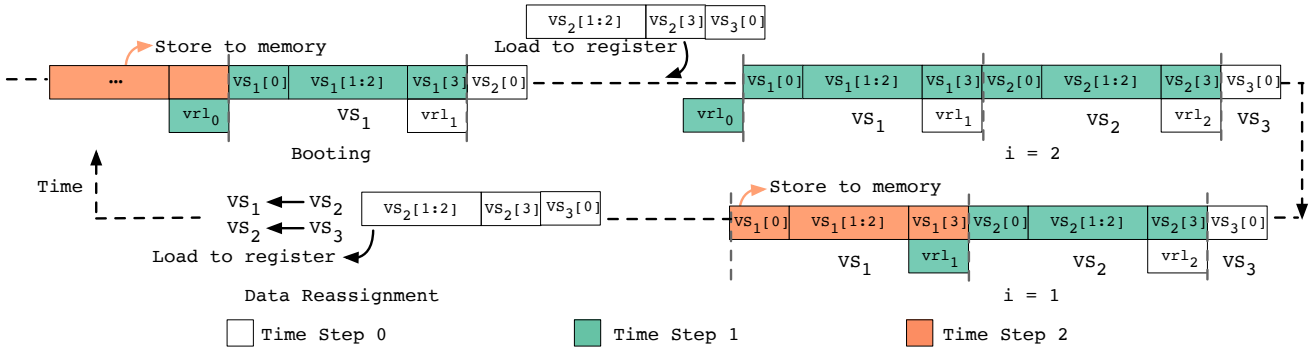

Fig. 4. Illustration of stencil computation for two time steps.

Algorithm 1 Unroll-and-jam the Time Loop

function ASSEMBLE($\mathbf{v}_a$,$\mathbf{v}_b$)
  $\mathbf{v}_c$=_mm256_blend_pd($\mathbf{v}_a$,$\mathbf{v}_b$)
  $\mathbf{v}_c$=_mm256_permute4×64_pd($\mathbf{v}_c$)
  return $\mathbf{v}_c$
end function
function COMPUTE($\mathbf{v}_{left}$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, $\mathbf{v}_4$, $\mathbf{v}_{right}$)
  $\mathbf{v}_0 \leftarrow$ ASSEMBLE($\mathbf{v}_{left}$,$\mathbf{v}_4$)
  $\mathbf{v}_5 \leftarrow$ ASSEMBLE($\mathbf{v}_1$,$\mathbf{v}_{right}$)
  for $i = 1 \rightarrow 4$ **do**
    $\mathbf{v}_{i-1} \leftarrow$ STENCIL($\mathbf{v}_{i-1}$, $\mathbf{v}_i$, $\mathbf{v}_{i+1}$)
  end for
  $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4 \leftarrow \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$
end function
function MULTIPLETIMESTEPS($\mathbf{VS}_{1:k}$,$\mathbf{vrl}_{0:k-1}$,k)
  for $j = k+1 \rightarrow N$ **do**
    $\mathbf{VS}_{k+1} \leftarrow$ Load the j-th Vector Set
    for $i = k \rightarrow 1$ **do**
      $\mathbf{vrl}_i \leftarrow \mathbf{VS}_i[3]$
      COMPUTE($\mathbf{vrl}_{i-1}, \mathbf{VS}_i[0:3], \mathbf{VS}_{i+1}[0]$)
    end for
    Store $\mathbf{VS}_1$
    for $i = 1 \rightarrow k$ **do**
      $\mathbf{VS}_i \leftarrow \mathbf{VS}_{i+1}$
      $\mathbf{vrl}_{i-1} \leftarrow \mathbf{vrl}_i$
    end for
  end for
end function
function MAIN( )
  while $t < T$ **do**
    Booting computation.
    MULTIPLETIMESTEPS($\mathbf{VS}_{1:k}$,$\mathbf{vrl}_{0:k-1}$,k)
    Epilogue computation.
    $t+ = k$
  end while
end function

to preserve the value of the last time of the vector to each vector set's left, denoted as $\mathbf{vrl}_i$. As the figure shows, $\mathbf{vrl}_i$ and $\mathbf{VS}_i[3]$ store the value of the same vector at time $t-1$ and t, respectively.

The MULTIPLETIMESTEPS function forwards all the vector sets from right to left by one time step. Meanwhile, it preserves the old value of their rightmost vector in $\mathbf{vrl}$. At the end of each iteration, $\mathbf{VS}_1$ has been updated k times and is stored in memory. Then after some data reassignments, the next loop is ready to execute. Each iteration loads and stores one vector set of $vl*vl$ elements and performs $k*vl*vl$ stencil computations. As mentioned above, it increases the in-CPU flops/byte ratio by k times.

From the algorithm, we see that it needs k vector sets and k additional vectors to unroll-and-jam the time loop, i.e., total $(vl+1)*k$ registers in addition to coefficient vector registers. In modern CPUs, the typical number of available vector registers is $vl*4$, where vl is the capacity of double precision variables in one register, therefore in this work we always set $k = 2$.

There is another advantage of the algorithm. Conventionally the stencil of Jacobi style is implemented with two arrays, storing the value at odd and even time respectively. If we set $k = 2$, then the input and output value are all at the even time. It's legal to reuse the input data space and make the whole computation in-place. The space usage is then reduced.

D. Integrated With Tiling

Vectorization and tiling are two orthogonal methods. They target at different levels. Vectorization boosts the computation using the data parallelism at the execution level, while tiling serves to exploit the data reuse at cache levels. The transpose layout described above identifies a vectorization technique as the solution to the data alignment conflict for stencils. The multiple time update further improves the data reuse ability at the CPU vector register level. In the following, we present the combination of the transpose layout and a tiling framework.

The tessellation tiling [35] can be viewed as a tessellation in iteration space by utilizing shaped tiles. Figure 5 (a) and Figure 5 (b) illustrate the tiling framework for a one-dimensional stencil. The iteration space is tessellated by triangles and inverted triangles in alternative stages. Thus, concurrent execution is processed by two stages which are started in each triangle with a given time range first, followed closely by the execution of inverted triangles over the same time range concurrently. Updates in different time steps are distinguished from each other by different colors, and the state of each element along

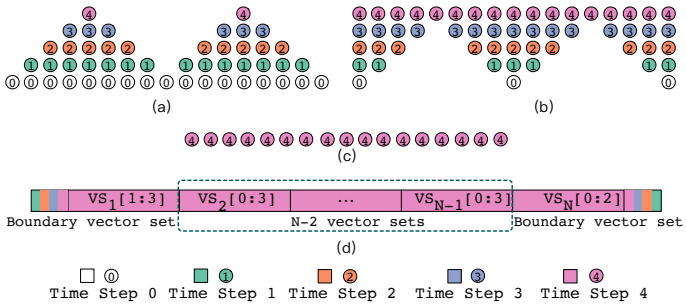

Fig. 5. Tessellate tiling iteration space for 1D updated with two time steps on register transpose layout.

the time dimension is represented with a number in Figure 5. For the example in Figure 5 (a), the new state of each triangle contains (0,1,2,3,4,3,2,1,0) where the center element is updated four steps and its neighbors are updated fewer steps proportional to the distance with the center element. To make all elements updated with the same steps, two half parts from adjacent triangles constitute new inverted triangles and the elements are updated with the state (4,3,2,1,0,1,2,3,4). As Figure 5 (c) shows, all elements are updated to four steps by adding the projection of the triangles with inverted triangles. With the tessellate tiling strategy, concurrent execution for different tiles is enabled over a given time range without redundant computation.

The only problem for applying the transpose layout is the calculations at the two boundaries of each block. The execution of triangles is a 'shrinking' process, the range of processed elements decreases as the time forwards. Similarly, an 'expanding' process occurs in the execution of inverted triangles. Since the physical neighbor elements are stored apart from each other in one vector set, the calculations of the vector set that covers a boundary are too complex to implement. As the basic computing unit in the transpose layout is a vector set, we convert the vector set at boundary back to the original format before the computation and employ a simple data reorganization method to process them. As illustrated in Figure 5 (d), the shrinking and expanding process could be simplified in this way. When the boundary slides away, the vector set is transposed again.

Further, the register transpose layout and time loop fusion make it feasible to achieve multiple time steps computation in registers over the tiles efficiently without reloading operations.

The tessellate tiling could also be applied for multidimensional stencil computations. For a d-dimensional stencil, tessellation in iteration space contains $d+1$ stages. Similar to the 1D stencil example in Figure 5, the spatial space in stage i is tessellated by $tiles_i$ $(0 \leq i \leq d)$. $tiles_1$ is a hypercube (typically a line segment in 1D, square in 2D, cube in 3D). $tiles_{i+1}$ is built by recombining the sub-tiles split from adjacent $tiles_i$ along some dimensions. Applying the transpose layout to higher-dimensional stencils is exactly similar to the one-dimensional case since the layout only affects the unit-stride dimension.

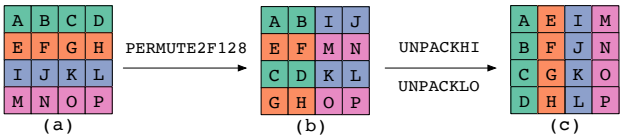

Fig. 6. Transpose for *double* type using AVX2 instructions.

E. Transpose

Unlike previous work [16] that performs a global dimension-lifted transformation, we only need a transpose on-the-fly for each register set twice throughout the whole process. The lower bound on the memory operations for completing a matrix transpose of size $vl * vl$ is $vl \log(vl)$, e.g., 8 data reorganization instructions for $vl = 4$. In modern CPU architectures, these 8 instructions can be launched continuously in 8 cycles. However, as detailed in Section V, the implementation of existing algorithm generally adopts lane-crossing instructions, which increases the overhead by 25%.

Figure 6 illustrates our improved version where the long-latency instructions are hidden by their following single-cycle instructions. In the first stage, pairs of two vectors with distance 2, e.g., (A, B, C, D) and (I, J, K, L), exchange data using the `permute2f128` instruction. In the second stage, the pairs of two adjacent vectors, e.g., (A, B, I, J) and (E, F, M, N), swap elements by the `unpackhi` or `unpacklo` instruction. The total cost of the new transpose scheme is then reduced to 8 cycles. Similarly, the transpose by using AVX-512 instructions contains three stages where the last stage consists of in-lane instructions.

IV. Evaluation

To achieve the proposed scheme efficiently, we employ the Advanced Vector Extensions (AVX) for vectorized implementation. First proposed by Intel in March 2008, AVX is an instruction set for microprocessors that rely on the x86 family of instruction set architectures [14]. AVX2 expands most integer operations to 256 bits and introduces fused multiply-accumulate (FMA) operations. Now, AVX-512 expands AVX to 512-bit operations using a new enhanced vector extension (EVEX) prefix encoding. In this section, we evaluate our proposed scheme for 1D, 2D and 3D stencils with AVX2 and AVX-512 instructions.

TABLE I
Parameter description for stencils used in experiments

Dim	Pts	Problem Size	Blocking Size
1D	3	10240000×1000	2000×1000
1D	5	10240000×1000	2000×500
2D	5	3000×3000×1000	200×200×50
2D	9	3000×3000×1000	120×128×60
3D	7	128×128×128×1000	23×23×10
3D	27	128×128×128×1000	23×23×10

A. Setup

a) Platforms: Our experiments were performed on a machine composed of two Intel Xeon Gold 6140 processors with 2.30 GHz clock speed, which owns 36 physical cores organized into two sockets. Each core contains a 32KB private L1 data cache, a 1 MB private L2 cache, and a unified 24.75MB L3 cache. It is also configured with 6 memory channels providing a max bandwidth of 127.96 GB/s per processor. 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 GFlop/s/core (2649.6 GFlop/s in aggregate).

b) Baselines: We first performed the sequential block-free experiments with three classic vectorization methods (Auto Vectorization [31], Data Reorganization [36], and DLT [16]) to investigate the absolute performance on a single process in Section IV-B. Since the recent tiling technique proposed by Yuan [36] and the nested/hybrid tiling technique (denoted as SDSL, which is the name of the software package.) presented by Henretty [17] outperform the other stencil research like Pluto [5], [7] and Pochoir [30], we then take them for further comparison on multicore architecture in Section IV-C. At last, the scalability was evaluated thoroughly compared with highly-optimized work and state-of-the-art compilers (SDSL [17], Pluto [5], and Tessellation [35]) in Section IV-D. The techniques that all benchmarks adopted for vectorization, register tiling, cache blocking, and parallelization are listed in Table II.

TABLE II
TECHNIQUES FOR VECTORIZATION, CACHE-BLOCKING, AND PARALLELIZATION IN BENCHMARKS

Benchmarks	Vectorized Methods	Register Tiling	Cache Blocking	Parallelization
SDSL [17]	DLT [16]	-	Split tiling [17]	OpenMP
Pluto [5]	AutoVec.	-	Diamond tiling [7]	OpenMP
Tesselation [35]	AutoVec.	-	Tessellate tiling [36]	OpenMP
Our*	Locally Trans.	U.&J.	Integrated tessellate	OpenMP

*For better clarity, Locally Transpose, Unroll-and-jam the Time Loop, and Integrated With Tiling proposed in our work are abbreviated with Locally Trans., U.&J., and Integrated tessellate respectively.

c) Kernels: The detailed parameters for stencils of various orders used in experiments are described in Table I, which consists of four star stencils (1D 3-Points, 1D 5-Points, 2D 5-Points, and 3D 7-Points) and two box stencils (2D 9-Points and 3D 27-Points) corresponding to the references [17], [36]. Time blocking sizes are the last numbers in the Problem Size column of Table I. The default value of total time steps is 1000 or 200 in the references. Thus, we fix it as a larger value of 1000 in our experiments. Other parameters of each stencil are also fine-tuned on the basis of references work to guarantee that the peak performance for all methods could be reached exactly. All programs were compiled using the ICC compiler version 19.0.3, with the '-O3 -xHost -qopenmp -ipo' optimization flags. Since the performance is sensitive to the stencil parameters, significant efforts are required in automatic tuning and this will be done separately as future work.

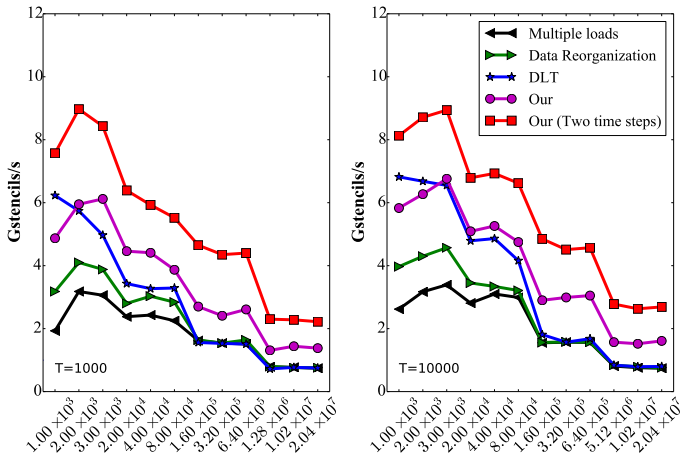

Fig. 7. Absolute performance comparison for tested methods in single-thread blocking-free experiments. The results are shown separately with different total time steps.

d) Metrics: Most stencil work (e.g., DLT [17], SDSL [26], Tessellation [35], [36], Pluto [5], etc.) exhibits results in terms of arithmetic performance (Stencils/s or Flop/s). In this work, we also adopt the metric of stencils per second (Stencils/s) defined in Equation 1 for measuring the performance. Here, N_x, N_y, N_z are the stencil size for each dimension; T is the iteration item; $time$ is the total execution time. Since stencil computation is memory-intensive, memory throughput is also a metric to measure the performance for stencil computation. The related discussion is expanded in Section IV-D on memory bandwidth for an additional analysis.

$$\textbf{stencils per second} = \frac{N_x \cdot N_y \cdot N_z}{time} \times T \qquad (1)$$

B. Sequential Block-free Results

In this subsection, we present performance results of varied methods across problem sizes ranging from L1 cache to main memory with a single thread. The spatial and temporal blocking method are not applied to them for investigating the pure improvements on various storage levels. The multiple loads and data reorganization methods represent a class of auto-vectorization in modern compilers [36]. DLT is the dimension-lifting transpose strategy designed by Henretty [16]. The used stencil is the classic Jacobi-style one-dimensional Heat equation kernel. All the methods are implemented by hand-written codes optimized with the appropriate strategies such as alignment and loop unrolling to ensure fairness.

Figure 7 shows the performance comparison of our methods with the other three methods. The results are illustrated separately in two subfigures on the basis of the total time steps T. It can be seen that our method updating two time steps outperforms others apparently in both experiments, which demonstrates the effectiveness of the improvement of the

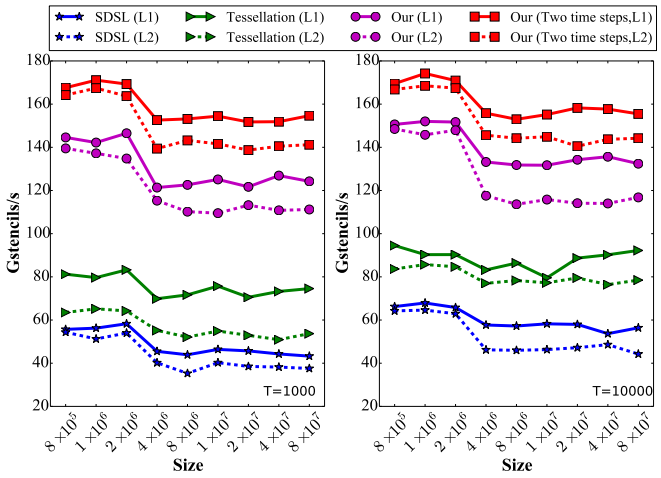

Fig. 8. Absolute performance comparison for varied methods in multicore cache-blocking experiments. The tiling size is fixed in L1 or L2 cache level. The results are shown separately with different total time steps.

flop/byte ratio. Our method without time loop unroll-and-jamming also achieves better performance results than the hand-written DLT in most cases. The performance has a decrease at the size of 1000 in L1 cache. This can be attributed to the cheaper dimension-lifting transpose operation in small size for DLT. The multiple loads method exhibits the worst performance among them due to the overhead caused by redundant loads.

TABLE III
PERFORMANCE IMPROVEMENTS ON DIFFERENT STORAGE LEVEL IN SINGLE-THREAD BLOCKING-FREE EXPERIMENTS

Storage Level	Data Reorganization	DLT	Our	Our (2 steps)
L1	1.28x	2.06x	2.16x	3.13x
L2	1.11x	1.37x	1.67x	2.07x
L3	1.01x	0.95x	2.02x	2.92x
Memory	1.00x	1.01x	1.97x	2.96x
Mean	1.11x	1.35x	1.98x	2.81x

To further investigate the effect of total time steps T, we perform a tenfold increase on the default value to $T = 10000$, which is illustrated in Figure 7 (b). It can be observed that the performance trends of $T = 10000$ are still largely consistent with the results in Figure 7 (a). However, the performance of our method falls slightly behind the DLT in L1 cache, and this performance anomaly is primarily due to the diluted dimension-lifting transpose cost by overly long time steps. Notably, only the performance of DLT in L1 cache drops gradually as problem size increases for both results in Figure 7, which is resulted from a costly data layout transformation and indicates a potential bottleneck for cache-blocking.

C. Multicore Cache-blocking Experiments

In this subsection, we present the multicore experimental results configured with the same stencil kernel in Section IV-B.

TABLE IV
PERFORMANCE IMPROVEMENTS ON DIFFERENT STORAGE LEVEL IN MULTICORE BLOCKING EXPERIMENTS

	Blocking Level	Tessellation	Our	Our (Two time steps)
L3 Cache	L1	1.43x	2.54x	2.99x
	L2	1.21x	2.58x	3.01x
Memory	L1	1.62x	2.76x	3.42x
	L2	1.39x	2.92x	3.58x
Mean	L1	1.56x	2.69x	3.29x
	L2	1.32x	2.79x	3.48x

The results are shown in Figure 8 (a) and Figure 8 (b) with time steps of $T = 1000$ and $T = 10000$ respectively. All methods adopt cache-blocking techniques, and we demonstrate the benefits of them progressively. The SDSL employs a split tiling technique (nested tiling in 1D, hybrid tiling for higher dimensions) to achieve temporal blocking. The Tesselation utilizes auto vectorization supported by the compiler [36]. The curve for "our" shows the performance of proposed locally transpose + Integrated tiling. The "our(two time steps)" curve presents the results of locally transpose + Integrated tiling + Unroll-and-jam the time loop strategy.

As can be seen from Figure 8 (a), the performance drops apparently as the problem size moves from L3 cache to the memory hierarchy, which is mainly caused by the cost of data transfers. We also further investigate the influence of the block size on performance. In the case of L1 blocking, the observed performance is higher than that with L2 blocking overall. Since the smaller stencils could be prefetched into cache directly, the performance gap between different blocks is further aggravated when the problem size lies in the memory hierarchy. Surprisingly, our method with two time steps could still take up a leading position, approximately 3.29x and 3.48x improvements are obtained compared to SDSL with L1 blocking and L2 blocking respectively. The performance of SDSL is inferior to tessellation, which is resulted from the blocking technique constrained to its data layout. Longer time steps of $T = 10000$ are evaluated in Figure 8 (b), and similar performance trends but higher values are observed compared with Figure 8 (a).

Table IV shows the detailed performance improvements on different storage levels as before. Our method could obtain better optimization results when the problem size lies in L3 cache and memory. The speedup ranges from 2.54 to 2.76x with L1 blocking, showing that our method integrated with tiling provides a significant benefit over others on varied problem size.

D. Scalability

We also evaluate the scalabilities of our schemes. The detailed parameters are given in Table I, where all problem sizes exceed the L3 cache. Since our tiling framework is the same as the tessellation scheme, the performance improvements of our method with respect to the tessellation method are fully

TABLE V
DASHBOARD FOR MEMORY BANDWIDTH (BW) IN SCALING EXPERIMENTS

Kernels	Pts	Stencil traffic (stencils/s)	Memory BW (GB/s)	BW utilization (%)
1D-Heat	3	2.58	41.28	16.13
1D5P	5	3.05	48.91	19.11
2D-Heat	5	3.67	58.74	22.95
2D9P	9	4.02	64.33	25.13
3D-Heat	7	4.77	76.38	29.84
3D27P	27	4.90	78.49	30.66
Theoretical Memory Bandwidth (GB/s)				127.96

derived from the vectorization. Here the results by state-of-the-art compiler Pluto [4], [7] are also shown for an auxiliary comparison on scaling performance.

Figure 9 illustrates the results of 1D, 2D and 3D stencils implemented with AVX2 and AVX-512 instructions respectively. The SDSL doesn't support the AVX-512 architecture. We also omitted Pluto since it has been proved to be inferior to the Tessellation technique. It can be observed that our method could obtain the highest performance while the SDSL performs the lowest performance in most cases. In one-dimensional stencils, all these methods achieve nearly linear scaling on both instruction sets and the proposed time loop fusion strategy 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. Compared to the results implemented with AVX2 instructions, the performance of the right half in Figure 9 shows a slight increase.

The speedups and scalabilities for high-order stencils including 1D5P, 2D9P, and 3D27P also decrease gradually from 1D to 3D. However, the overall performance falls behind the corresponding one-order results, which is resulted from complex data access patterns in high-order stencils. Our method could also obtain a substantial performance improvement in all experiments. Taking all stencils with AVX2 instructions into account, remarkable performance benefits are observed from our method updating two time steps, 3.52x and 2.92x respectively for 1D3P and 1D5P. The performance improvement ranges from 1.66x to 2.77x with a mean of 2.10x, demonstrating that our vectorization scheme provides a significant benefit in a large problem size compared to the referenced work. For scalability, our method obtains a 20.1x speedup while the value of DLT is only 9.4x for 3D7P, which indicates a sustainable performance for our method in multidimensional stencils.

The achieved throughput of global memory is related to the transferred data size, as well as the efficiency of utilizing cache. It is approximately proportional to the transferred data size before saturating the global memory bandwidth if there is no contention. The memory bandwidths under our vectorized methods with 36 cores are shown in Table V, which are obtained by Intel Processor Counter Monitor [1]. As can be seen from Table V, the memory throughput increases with the growing order and dimension of stencils. Since the stencils with smaller order or dimension are calculated more quickly, the cache requires to be updated frequently, which leads to a larger size of transferred data. The bandwidth of 3D27P is still unsaturated with a 30.66% utilization, which illustrates an effective cache-blocking optimization in our vectorized methods.

E. 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.

Sequential block-free experiments examine a variety of vectorization methods and demonstrate that our scheme with multiple time steps updating can achieve an considerable 2.81x improvement on average compared with the multiple loads method. Subsequently, the performance gains for a larger time steps are still significant and consistent with the results of the small time steps. Moreover, the DLT method is more appropriate only on the relatively small size and long time steps, and this is partly explained by the performance penalty associated with additional dimension-lifting transpose in memory. Since the problem size ranges from L1 cache to main memory, clear insights are provided that the overall performance trends drop consistently with various memory hierarchy.

Multicore cache-blocking experiments conduct stencil cases with 36 cores, and an average 2.69x speedup is obtained by our method on the basis of SDSL. Due to the reduced data transfers by our time loop unroll-and-jam, our method updated with two time steps achieve a further 3.29x speedup. We also study the influence of blocking size, and the results prove that appropriate L1 blocking or in-cache problem size could contribute to better performance for all methods. The overall trends are in accord with the sequential block-free experiments, and our method updated with two time steps outperforms others obviously.

The scalability experiments demonstrate that our vectorized scheme leveraging tessellate tiling successfully outperforms the referenced fastest multicore stencil work to date across a broad variety of configurations. Constrained to its specific data layout, DLT 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, and our method could still obtain a better performance.

V. RELATED WORK

Research on optimizing stencil computation has been intensively studied [16], [17], [20], [35], [36], and it can be broadly classified as optimization methods to improve the computation performance and enhance the data reuse.

Vectorization by using SIMD instructions is an effective way to improve computation performance for stencils. Prior

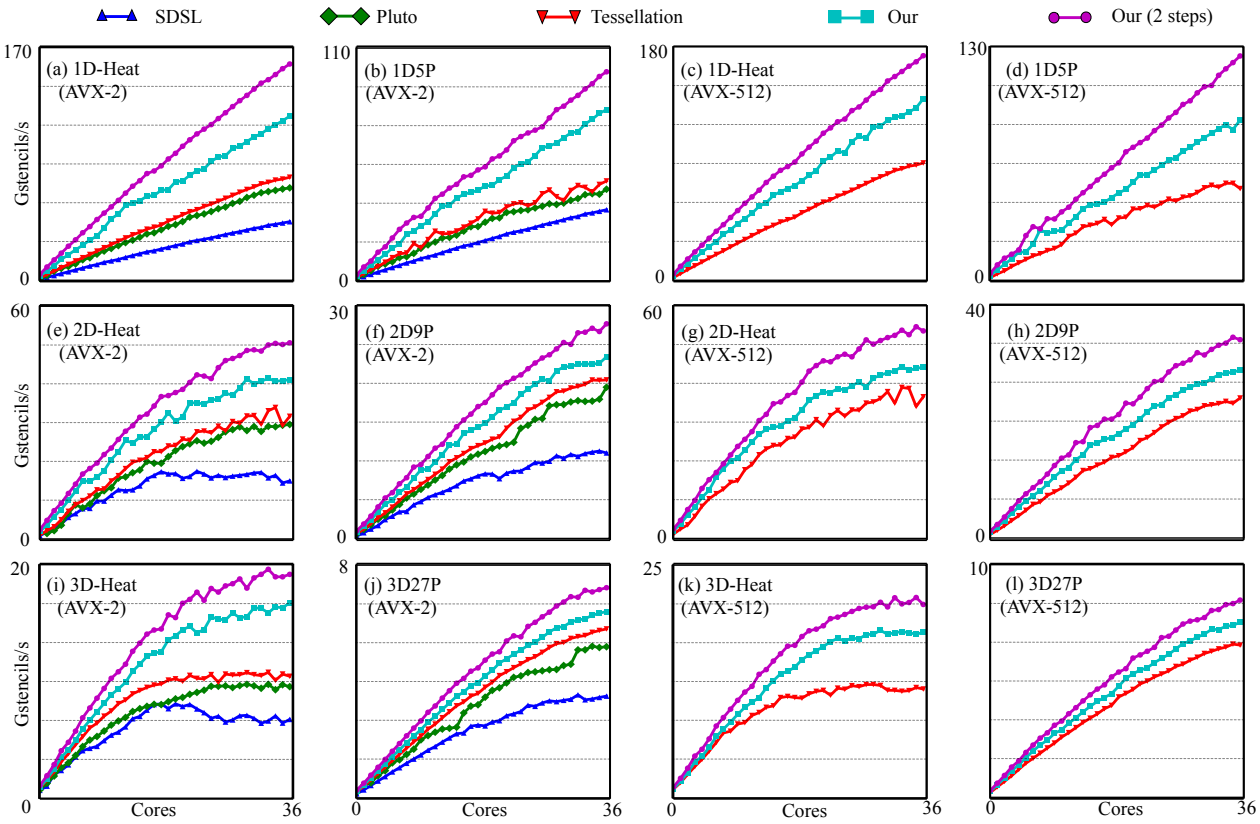

Fig. 9. Performance comparison for stencils of various orders with different dimensions in a multicore environment.

work 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 [39]. 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 [34] could improve data reuse by using common expression elimination and unrolling based on their vector-folding methods with fine-grained blocks [33], which is less feasible for high-order complex stencils [38]. Henretty proposes a new method DLT [16], [17] to overcome input data alignment 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 [17]. Specifically, the original innermost loop traverses the corresponding dimension from 1 to N. 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. In addition, the in-place matrix transpose involved in our work has also been widely studied and a kernel of 4×4 matrix transpose consists of two stages basically. Hormati splits the vector register to some 128-bit lanes [18], and the lane-crossing instructions for $double$ incur a longer latency, typically 3 to 4 cycles. Zekri [37] use in-lane instructions in four stages only for $float$ type. Springer [29] utilize SHUFFLE and PERMUTE2F128 instructions for $double$ type in two stages, while it requires 8 integers as parameters.

Tiling 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 [24], time skewed tiling [19], diamond tiling [4], cache oblivious Tiling [13], split-tiling [17] and tessellating [35]. Wonnacott and Strout present a comparison on the scalability of many existing tiling schemes [32]. 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], [15], [28] 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. The Pluto [7] is able to generate the diamond tiling for 1D stencil. Bandishti [4] extended it to higher dimension stencils.

VI. Conclusion

In this paper, we propose a novel transpose layout to overcome the input data alignment conflicts efficiently for vectorization. A time loop unroll-and-jam strategy with in-register multiple time steps processing is designed on the basis of the proposed transpose layout. Furthermore, we describe how the proposed vectorization scheme is integrated with a tessellate tiling framework for enhancing data reuse and concurrency. 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 [7], [17], [31], [35], [36].

References

[1] Intel processor counter monitor.

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

[3] Krste Asanovic, Ras Bodik, James Demmel, et al. The parallel computing laboratory at uc berkeley: A research agenda based on the berkeley view. *University of California, Berkeley, Tech. Rep*, 2008.

[4] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. SC '12, pages 1–11, Nov 2012.

[5] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *SC'12*, pages 1–11. IEEE, 2012.

[6] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella. Compiler-directed transformation for higher-order stencils. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 313–323, 2015.

[7] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI 08*, 2008.

[8] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS 2011*. IEEE.

[9] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review*, 51(1):129–159, 2009.

[10] Hikmet Dursun, Manaschai Kunaseth, Ken-ichi Nomura, Jacqueline Chame, RobertF. Lucas, Chun Chen, Mary Hall, RajivK. Kalia, Aiichiro Nakano, and Priya Vashishta. Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters. *The Journal of Supercomputing*, 62(2):946–966, 2012.

[11] Fabian Dütsch, Karim Djelassi, Michael Haidl, and Sergei Gorlatch. Hlsf: A high-level; c++-based framework for stencil computations on accelerators. WOSC '14, pages 41–4, 2014.

[12] Stephan Falke, Florian Merz, and Carsten Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 108–128. Springer, 2013.

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

[14] Intel Intrinsics Guide. Url: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. *IntrinsicsGuide (access date: 12.6.2021)*.

[15] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. Modesto: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *ICS 2015*, pages 177–186, 2015.

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

[17] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. ICS '13, pages 13–24, 2013.

[18] Amir H Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Macross: macro-simdization of streaming applications. *ACM SIGARCH computer architecture news*, 38(1):285–296, 2010.

[19] Guohua Jin, John Mellor-Crummey, and Robert Fowler. Increasing temporal locality with skewing and recursive blocking. SC '01, pages 43–43, 2001.

[20] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. MSP '05, pages 36–43, 2005.

[21] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. PLDI '07, pages 235–244, 2007.

[22] Kun Li, Honghui Shang, Yunquan Zhang, et al. Openkmc: a kmc design for hundred-billion-atom simulation using millions of cores on sunway taihulight. In *SC'19*, 2019.

[23] Yulong Luo, Guangming Tan, Zeyao Mo, and Ninghui Sun. Fast: A fast stencil autotuning framework based on an optimal-solution space model. ICS '15, pages 187–196, 2015.

[24] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. SC '10, pages 1–13, Nov 2010.

[25] P. S. Rawat, A. Sukumaran-Rajam, A. Rountev, F. Rastello, L. Pouchet, and P. Sadayappan. Associative instruction reordering to alleviate register pressure. In *SC'18*, pages 590–602, 2018.

[26] Prashant Rawat, Martin Kong, Tom Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J Ramanujam, Atanas Rountev, and P Sadayappan. Sdslc: A multi-target domain-specific compiler for stencil computations. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 1–10, 2015.

[27] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P Sadayappan. Register optimizations for stencils on gpus. In *PPOPP'2018*, pages 168–182.

[28] Rodrigo C. O. Rocha, Alyson D. Pereira, Luiz Ramos, and Luís F. W. Góes. Toast: Automatic tiling for iterative stencil computations on gpus. *Concurrency and Computation: Practice and Experience*, 2017.

[29] Paul Springer, Jeff R Hammond, and Paolo Bientinesi. Ttc: A high-performance compiler for tensor transpositions. *ACM Transactions on Mathematical Software (TOMS)*, 44(2):1–21, 2017.

[30] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *SPAA'11*, New York, NY, USA, 2011. ACM.

[31] Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, and Ernesto Su. Intel® openmp c++/fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6(1), 2002.

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

[33] Charles Yount. 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*, pages 865–870. IEEE, 2015.

[34] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In *WOLFHPC 2016*, pages 30–39. IEEE, 2016.

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

[36] Liang Yuan, Yunquan Zhang, Peng Guo, and Shan Huang. Tessellating stencils. In *SC'17*, New York, NY, USA, 2017. ACM.

[37] Ahmed Sherif Zekri. Enhancing the matrix transpose operation using intel avx instruction set extension. *International Journal of Computer Science & Information Technology*, 6(3):67, 2014.

[38] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. Exploiting reuse and vectorization in blocked stencil computations on cpus and gpus. In *SC'19*, pages 1–44, 2019.

[39] Gerhard Zumbusch. *Vectorized Higher Order Finite Difference Kernels*, pages 343–357. 2012.