Temporal Vectorization for Stencils

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

Hang Cao
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
caohang@ict.ac.cn

Yunquan Zhang
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China

Kun Li
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Pengqi Lu
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Yue Yue
SKL of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

ABSTRACT

Stencil computations represent a very common class of nested loops in scientific and engineering applications. Exploiting vector units in modern CPUs is crucial to achieving peak performance. Previous vectorization approaches often consider the data space, in particular the innermost unit-strided loop. It leads to the well-known data alignment conflict problem that vector loads are overlapped due to the data sharing between continuous stencil computations. This paper proposes a novel temporal vectorization scheme for stencils. It vectorizes the stencil computation in the iteration space and assembles points with different time coordinates in one vector. The temporal vectorization leads to a small fixed number of vector reorganizations that is irrelevant to the vector length, stencil order, and dimension. Furthermore, it is also applicable to Gauss-Seidel stencils, whose vectorization is not well-studied. The effectiveness of the temporal vectorization is demonstrated by various Jacobi and Gauss-Seidel stencils.

KEYWORDS

Stencil Computation, Vectorization, Data Alignment Conflicts

ACM Reference Format:
Liang Yuan, Hang Cao, Yunquan Zhang, Kun Li, Pengqi Lu, and Yue Yue. 2021. Temporal Vectorization for Stencils. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476149

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/10.1145/3458817.3476149

1 INTRODUCTION

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

The naive implementation of a d-dimensional stencil is comprised of $(d + 1)$ loops where the outermost loop traverses the time dimension and the inner loops update all grid points in the d-dimensional spatial space. It exhibits poor data reuse and is a typical bandwidth-bound kernel. To improve performance, blocking and vectorization are the two most powerful and commonly used transformation techniques.

There are two kinds of blocking methods for stencil computations: *spatial blocking* and *temporal blocking*. The spatial blocking algorithms promote data reuse in a single time step for 2D and higher dimension stencils by adjusting the data traversal pattern to an optimized order. An in-cache grid point may be reused to update all its neighbors before evicted from the cache. However, the locality exploited by space blocking is limited by the neighbor pattern size of a stencil. The temporal tiling [11, 29, 30, 34, 37, 52] takes the time dimension and spatial dimensions into consideration simultaneously. It has been exhaustively studied for stencils to further improve the data locality and alleviate the memory bandwidth demands.

The vectorization groups a set of data in a vector register and processes them in parallel to utilize vector units in modern CPU architectures. It exploits the data parallelism and serves to boost the in-core performance. There has been a long history of efforts to design efficient vectorization methods [1, 15, 21, 25, 28, 32, 40, 54]. Though the stencil computation is characterized by its apparently low arithmetic intensity, the vectorization is still profitable, especially for blocked stencil algorithms. Prior vectorization techniques for stencils focus on the data space, i.e. group points either in the unit-strided space dimension [16] or multiple space dimensions [50]. We refer to this scheme as *spatial vectorization*.

One well-known problem induced by the spatial vectorization of stencils is the data alignment conflict. It arises from the fact that continuous vectors require that the same value appears at different positions of vectors. Thus it incurs either redundant loads or additional data organization operations. Existing solutions [8, 16] reduce these overheads but still limit the performance or hurt the data locality. We will provide a detailed analysis in Section 2.

Furthermore, vectorization and blocking are often regarded as two orthogonal methods. However, the vectorization and tiling actually interact with each other for stencil computations. The vectorization often requires higher bandwidth and prefers loading data from the first-level cache. On the contrary, the tiling tries to minimize the data transfer at the memory-cache level and prefers the last-level cache. This performance gap motivates our work and will be discussed in Section 3.1.

In this paper, we propose a novel *temporal vectorization* scheme considering the entire iteration space. It expands the target scope of vectorization from the spatial space to the iteration space. A vector in the temporal vectorization scheme groups points with different time coordinates. It seeks to alleviate the data alignment conflict and bridges the above-mentioned performance gap. We also design a set of optimizations to alleviate weaknesses induced by the new scheme and adjust the data layout to explore its potential.

The temporal vectorization still requires data reorganization operations, but the overhead is fixed and smaller than previous methods. Furthermore, a vectorization scheme usually only affects the single-core performance. However, the proposed temporal vectorization method leads to better utilization of the memory bandwidth. In particular, it loads the data at a slower frequency. Thus it can expect less memory contention, especially for multi-core executions. We implemented the temporal vectorization with temporal blocking schemes and show that the speedup increases with the number of cores, especially for high-dimensional stencils. Finally, the temporal vectorization scheme is also applicable to the Gauss-Seidel stencils. Gauss-Seidel stencils update a point using the newest values of neighbor points. It is illegal to vectorize any single loop of the naive implementation code. To the best of our knowledge, we are not aware of vectorization methods for Gauss-Seidel stencils.

The remainder of this paper is organized as follows. Section 2 provides the background. The temporal vectorization is described in Section 3. We present the performance results in Section 4. Section 5 overviews related work and Section 6 concludes the paper.

2 BACKGROUND

2.1 Data Alignment Conflict of Vectorization

We take the 1D3P stencil as an example to illustrate the fundamental problem of the stencil computations caused by vectorization. The pseudo-code is listed in Algorithm 1. a_x^t represents the value at the

Algorithm 1: 1D3P Jacobi Stencil, scalar code

```
for ( $t = 0;\ t < T;\ t = t + 1$ ) {
    for ( $x = 1;\ x <= NX;\ x = x + 1$ ) {
        $a_x^{t+1} \leftarrow$Stencil($a_{x-1}^t,\ a_x^t,\ a_{x+1}^t$);
    }
}
```

point (t, x) in the iteration space where x and t are the coordinates in the time and space dimension, respectively. In each iteration of the inner space loop, it loads a_{x+1}^t, reuses in-register data a_{x-1}^t and a_x^t referenced by the previous calculation and writes the result a_x^{t+1} to memory. Observing the CPU-memory data transfer, one iteration of the inner loop is exactly similar to a common array copy algorithm.

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 a_x^{t+1}. We assume the vector register holds 4 elements (i.e. vector length $vl = 4$) in the rest of this paper. Thus the vectorized code performs the calculation with vector operations and outputs $(a_1^{t+1}, a_2^{t+1}, a_3^{t+1}, a_4^{t+1})$ using one vector register.

A well-known problem incurred by the vectorization of stencil codes is the input data alignment conflicts. For example, to compute $(a_1^{t+1}, a_2^{t+1}, a_3^{t+1}, a_4^{t+1})$, it requires three vectors: $(a_0^t, a_1^t, a_2^t, a_3^t)$, $(a_1^t, a_2^t, a_3^t, a_4^t)$ and $(a_2^t, a_3^t, a_4^t, a_5^t)$. The element a_2^t appears in all these vector registers but at different positions.

The fundamental reason for the data alignment conflict is the data sharing between continuous calculations, e.g., a_x^1 and a_{x+1}^1 depend on same points a_x^0 and a_{x+1}^0. We refer to this as the *read-read dependence*. Conventionally the read-read dependence is not a data hazard as other data dependencies including read-after-write, write-after-read, and write-after-write dependencies. Furthermore, common read-read dependence is usually exploited to promote data locality. However, for vectorized stencil codes, the data alignment conflict arises from the fact that components in one output vector or at the different positions of multiple output vectors have intra-vector or inter-vector read-read dependencies. Then the required data must redundantly appear in many vectors.

2.2 Existing methods

We present three existing solutions to the data alignment problem and discuss their drawbacks.

Multiple load vectorization. The common vectorization employed by production compilers loads all the needed vectors from memory straightforwardly as shown in Algorithm 2. Due to the low operational intensity, the stencil computation is often regarded as a memory-starved 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.

Algorithm 2: 1D3P Jacobi Stencil, multi-load code

```
for ( $t = 0$; $t < T$; $t = t + 1$ ) {
    for ( $x = 1$; $x <= NX - 3$; $x = x + 4$ ) {
        $v_0 \leftarrow (a^t_{x-1}, a^t_x, a^t_{x+1}, a^t_{x+2})$;
        $v_1 \leftarrow (a^t_x, a^t_{x+1}, a^t_{x+2}, a^t_{x+3})$;
        $v_2 \leftarrow (a^t_{x+1}, a^t_{x+2}, a^t_{x+3}, a^t_{x+4})$;
        $(a^{t+1}_x, a^{t+1}_{x+1}, a^{t+1}_{x+2}, a^{t+1}_{x+3}) \leftarrow$ Stencil($v_0$, $v_1$, $v_2$);
    }
}
```

Data reorganization vectorization. Another solution [8, 55] 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 data permutations instructions. Compared with the multiple load method, this data permutations method reduces the memory bandwidth usage and takes the 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.

A common disadvantage of these two approaches is that the number of redundant data loads or reorganization operations increases with the order of a stencil, the length of the CPU vector register and the dimensionality of the problem. For example, to compute the vector $(a^1_1, a^1_2, a^1_3, a^1_4)$ of the 1D5P stencil, it needs to put a^0_3 in four vectors at all different positions to update a^1_1, a^1_2, a^1_3 and a^1_4. Thus the redundancy is proportional to the order of a stencil and at most $vl - 1$. For the 2D9P stencil, the innermost loop incurs two redundant loads and the outer space loop incurs another four.

Dimension-Lifting Transpose (DLT). One milestone approach to address the data alignment conflict is the DLT method [16]. It turns to put the points with read-read dependencies in the same position of different vectors. Specifically, the original one-dimensional array of length N is viewed as a matrix of size $vl * (N/vl)$. It then performs a matrix transpose. Consider the DLT method for a one-dimensional array of 28 elements. The second $vl = 4$ elements in the transformed layout are contiguous stored and loaded into one output vector $(a^1_1, a^1_8, a^1_{15}, a^1_{22})$. All the three required input vectors: $(a^0_0, a^0_7, a^0_{14}, a^0_{21})$, $(a^0_1, a^0_8, a^0_{15}, a^0_{22})$ and $(a^0_2, a^0_9, a^0_{16}, a^0_{23})$ are free of data sharing and also stored contiguously in memory. DLT only needs to assemble input vectors for calculating output vectors at boundaries.

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 stencils and low-dimensional stencils in other applications like image processing, the time size is often too small to amortize the overhead. Third, 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. Finally, DLT fails to apply to Gauss-Seidel stencils.

3 TEMPORAL VECTORIZATION

3.1 Motivation

Our work is motivated by two observations on the data transfers between the CPU and cache, and between the cache and memory.

First, the vectorized codes often achieve higher performance than scalar codes. Furthermore, the data alignment conflict induced by vectorization requires either more data transfers or data reorganization operations. Consequently, the vectorization increases the bandwidth demands. As will be demonstrated in the Evaluation section, all sequential non-blocking stencil implementations with existing vectorization techniques achieve the highest performance when the problem sizes fit in the L1 cache and the performance decreases fast as the problem size increases. Since the L2 cache provides competitive bandwidth compared with the L1 cache, it implies that existing vectorization schemes are relatively cache-bandwidth-sensitive.

Second, L1 cache sizes in modern CPUs are often small. The typical size is around 32 KB. It holds up to 4000 elements for a double-precision floating-point kernel. Thus for higher dimension stencils, the space blocking sizes and the corresponding temporal blocking size are limited, which will lead to a high memory transfer volume. As will be demonstrated in the Evaluation section, the parallel blocking stencil implementations with existing vectorization techniques often get the best performance when the block fits in L1 cache or L2 cache for one-dimensional or high-dimensional stencils. This demonstrates the memory-bandwidth-bound restriction should be first satisfied even it incurs a slower in-core performance for L2 cache. This indicates that the blocking scheme is cache-size-sensitive especially for high dimension stencils.

These two observations illustrate the trade-off between the data locality exploitation at cache-memory level and the in-core performance of the vectorized codes at the CPU-cache level. The conventional innermost loop vectorization leads to the best data reuse at the memory-cache level while incurs redundant CPU-cache data transfers.

The DLT generates an optimal in-core data access pattern while hurting the data locality in the cache. The sequential DLT results [16] exhibit performance improvements for all stencils when the data fit in caches. However, the DLT with a blocking scheme [17] derives considerable speedups for 1D stencils but only competitive or even worse performance for high-dimensional stencils. This implies that the degradation of data locality outweights the benefits of the vectorization for the DLT. We seek to design a vectorization scheme that maintains the data reuse ability and incurs lightweight in-core data reorganizations simultaneously.

3.2 One-dimensional Stencil

The key idea is to extend the vectorization scope from the data space to the iteration space. Specifically, data points with different time coordinates are assembled in one vector. We take the 1D3P stencil as an example. The general form of a vector in our scheme for a one-dimensional stencil is $(a^{t_3}_{x_3}, a^{t_2}_{x_2}, a^{t_1}_{x_1}, a^{t_0}_{x_0})$. For the stencil kernels used in the temporal vectorization, we always set $t_{i+1} - t_i = 1$. The *space stride* $s = x_i - x_{i+1}$ is determined by the stencil. Note that it becomes the common vectorization in the multi-load and data reorganization approaches when $t_{i+1} = t_i$ and $s = 1$, and DLT when $t_{i+1} = t_i$ and $s = NX/4$.

To perform the stencil computation with a vector, it only requires that the elements in the vector are free of dependence. This is equivalent to respecting the dependence defined by a stencil. Let the dependence set of a stencil be D. Each dependence $(dt, dx) \in D$ implies that there exists two points (t', x') and (t'', x'') with $t'' - t' = dt$ and $x'' - x' = dx$ in the iteration space that $a_{x''}^{t''}$ depends on $a_{x'}^{t'}$. It is easy to show that the temporal vectorization is legal when $s > \max\{dx/dt | (dt, dx) \in D, dx > 0\}$. We only consider dependencies with $dx > 0$ since the innermost loop traverses the space dimension in increasing order. Take the 1D3P Jacobi stencil as an example, the dependencies are $(1, 0)$, $(1, 1)$ and $(1, -1)$. Thus it is sufficient to make the temporal vectorization legal by setting $s > 1$.

Algorithm 3 shows the pseudo-code of the temporal vectorization of the 1D3P Jacobi stencil with the space stride $s = 2$. Figure 1 illustrates the Algorithm. The outer-loop (Line 1) iteratively sweeps a time tile of the height equivalent to the vector length ($vl = 4$), i.e. each iteration forwards all grid points from time coordinate t to $t + 4$. Thus we can ignore t in the rest of the paper and only focus on the first iteration ($t = 0$) of the outer-loop.

In each iteration of the outer loop, Lines 2-4 update a small set of grid points at time $t + 1$, $t + 2$ and $t + 3$ (the iteration points surrounded by the blue staircase-like dashed line in Figure 1). Lines 5-7 collect some of these pre-computed values into vectors v_0, v_1, v_2 for the first vectorized stencil computation. Figure 1 shows these vectors with different colors (blue, green and orange). These vectors are called *input vectors* since they are fed to the vectorized stencil computing. Line 12 performs the calculations and generates an *output vector* v_3 (colored red in Figure 1).

As illustrated in Figure 1, the temporal vectorization requires the data reorganization of the *output vector* $v_3 = (a_x^4, a_{x+s}^3, a_{x+2s}^2, a_{x+3s}^1)$ to assemble an input vector $(a_{x+s}^3, a_{x+2s}^2, a_{x+3s}^1, a_{x+4s}^0)$ used in subsequent computations. It needs to rotate the output vector, copy out a_x^4 and copy in a_{x+4s}^0. In modern CPU architectures, the vector register is split into some 128-bit lanes (two components per lane in our example). Data movements inside lanes incur a lower latency, typical 1 cycle versus 3 for lane-crossing instructions. Since the rotation of the output vector requires a lane-crossing one, we try to implement all other data organizations with in-lane ones.

The component a_*^4 at the highest position of the output vector is the actual output value of the innermost loop and must be stored in memory since the next iteration of the outer time loop only requires the values with the time coordinate 4. To reduce the memory write instructions, the elements a_*^4 of the output vectors in every four continuous iterations of the innermost loop are assembled in one *top vector* $v_{top} = (a_{x+3}^4, a_{x+2}^4, a_{x+1}^4, a_x^4)$ (Line 15) and written to memory with a single vector-storing instruction (Line 18).

The following list shows the *MergeTop* operations in four iterations $x = 1, 2, 3, 4$. The first vector in each row is the output vector of each corresponding iteration and in each vector we only show the value with the time coordinate 4. The first two output vectors are rotated firstly since their values of time coordinate 4 will be moved

Algorithm 3: 1D3P Jacobi Stencil, temporal vectorization code with $s = 2$, $vl = 4$ and $T\%4 = 0$

```
1  for ( $t = 0;\ t < T;\ t = t + 4$ ) {
2      Compute{$a_1^{t+1}, \ldots, a_{2+2s}^{t+1}$};
3      Compute{$a_1^{t+2}, \ldots, a_{2+s}^{t+2}$};
4      Compute{$a_1^{t+3}, \ldots, a_2^{t+3}$};
5      $v_0 \leftarrow (a_0^{t+3}, a_s^{t+2}, a_{2s}^{t+1}, a_{3s}^t)$;
6      $v_1 \leftarrow (a_1^{t+3}, a_{1+s}^{t+2}, a_{1+2s}^{t+1}, a_{1+3s}^t)$;
7      $v_2 \leftarrow (a_2^{t+3}, a_{2+s}^{t+2}, a_{2+2s}^{t+1}, a_{2+3s}^t)$;
8      for ( $x = 1;\ x <= NX + 1 - 4s;\ x = x + 1$ ) {
9          if (1 == $x$%4) then
10             $v_{down} \leftarrow (a_{x+4s+3}^t, a_{x+4s+2}^t, a_{x+4s+1}^t, a_{x+4s}^t)$;
11         end
12         $v_3$ ←Stencil($v_0$, $v_1$, $v_2$);        // $v_3 = (a_x^{t+4}, a_{x+s}^{t+3}, a_{x+2s}^{t+2}, a_{x+3s}^{t+1})$
13         $v_0$ ←$v_1$;
14         $v_1$ ←$v_2$;
15         $v_{top}$ ←MergeTop($v_3$, $v_{top}$, $(x - 1)$%4);
16         $v_2$ ←MergeDown($v_3$,$v_{down}$, $(x - 1)$%4);
17         if (0 == $x$%4) then
18             $(a_{x+3}^{t+4}, a_{x+2}^{t+4}, a_{x+1}^{t+4}, a_x^{t+4}) \leftarrow v_{top}$;
19         end
20     }
21     $(a_{NX-3s-1}^{t+3}, a_{NX-2s-1}^{t+2}, a_{NX-s-1}^{t+1}, a_{NX-1}^t) \leftarrow v_0$;
22     $(a_{NX-3s+0}^{t+3}, a_{NX-2s+0}^{t+2}, a_{NX-s+0}^{t+1}, a_{NX+0}^t) \leftarrow v_1$;
23     $(a_{NX-3s+1}^{t+3}, a_{NX-2s+1}^{t+2}, a_{NX-s+1}^{t+1}, a_{NX+1}^t) \leftarrow v_2$;
24     Compute{$a_{NX}^{t+1}$};
25     Compute{$a_{NX-s}^{t+2}, \ldots, a_{NX}^{t+2}$};
26     Compute{$a_{NX-2s}^{t+3}, \ldots, a_{NX}^{t+3}$};
27     Compute{$a_{NX-3s}^{t+4}, \ldots, a_{NX}^{t+4}$};
28 }
```

to the two lower positions in v_{top}. The first value a_1^4 is directly copied to the empty v_{top} while the second vaule a_2^4 is shuffled with v_{top}. The other two output values a_3^4 and a_4^4 are blended to the two higher positions of v_{top} before rotating their corresponding output vectors. Note that there is a shuffle operation to the top vector in the iteration $x = 3$ to empty the highest component which will be filled in the last iteration with a_4^4. In sum, it needs 2, 1 and 4 data reorganization instructions for the rotation of output vector (lane-crossing), shuffle of top vector (in-lane), and combination of output and top vectors (in-lane), respectively.

$$x = 1 : (a_1^4, *, *, *) \xrightarrow{rotate} (*, *, *, a_1^4) \xrightarrow{copy} (*, *, *, a_1^4) = v_{top}$$

$$x = 2 : (a_2^4, *, *, *) \xrightarrow{rotate} (*, *, *, a_2^4) \xrightarrow{shuffle} (*, *, a_1^4, a_2^4) = v_{top}$$

$$x = 3 : (a_3^4, *, *, *) \xrightarrow{blend} (a_3^4, *, a_1^4, a_2^4) \xrightarrow{shuffle} (*, a_3^4, a_2^4, a_1^4) = v_{top}$$

$$x = 4 : (a_4^4, *, *, *) \xrightarrow{blend} (a_4^4, a_3^4, a_2^4, a_1^4) = v_{top}$$

Similarly, the *down vector* $v_{down} = (a_{x+4s+3}^0, a_{x+4s+2}^0, a_{x+4s+1}^0, a_{x+4s}^0)$ containing four continuous values with time coordinate 0 is loaded from memory by a vector-loading instruction (Line 10). It is blended with four output vectors calculated in four continuous

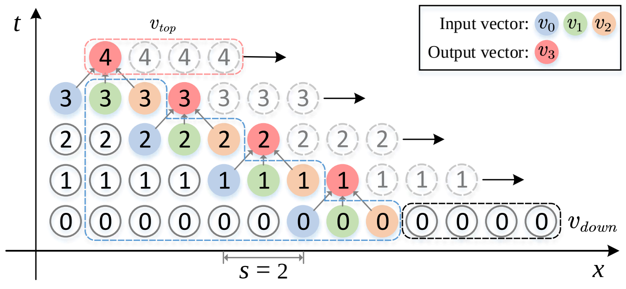

Figure 1: Temporal Vectorization of 1D3P Jacobi Stencil

iterations to generate four corresponding input vectors (Line 16). This task incurs 2, 2 and 4 data reorganization instructions for the output vector rotation (lane-crossing), down vector shuffle (in-lane) and output and down vectors combination (in-lane), respectively. We omit the details of *MergeDown* as it is similar to *MergeTop*.

The numbers of data reorganization operations in every four output vector computations are listed in the first row of Table 1. The data reorganization instructions involving the down or top vectors are in-lane operations that incur small latency. Other instructions manipulating the output vectors solely are lane-crossing instructions. Finally, Line 21-27 process some iteration points that are not covered by the inner loop.

Efficient data reorganization. To further reduce the data organization overhead, especially the number of lane-crossing instructions, our key observation is that only a_x^4 and a_{x+2s}^2 in the output vector are moved across lanes for the corresponding input vector assembling. Therefore, we turn to copy out the value with the time coordinate 2 and copy in a new one which is already in the high lane.

To achieve this, we add another space stride (denoted as sl) between lanes of the vector. Specifically, the form of output vectors becomes $(a_x^4, a_{x+s}^3, a_{x+2s+sl}^2, a_{x+3s+sl}^1)$. To form the corresponding input vector, both a_x^4 and $a_{x+2s+sl}^2$ are copied out to a *top-middle vector.* Then it is blended with a *middle-down* vector containing a_{x+2s}^2 and $a_{x+4s+sl}^0$ at different lanes to form the corresponding input vector $(a_{x+s}^3, a_{x+2s}^2, a_{x+3s+sl}^1, a_{x+4s+sl}^0)$. Thus the number of data reorganization instructions on the critical path of the input vector assembling is decreased to 2 and both of them can be implemented with in-lane instructions. In this paper, we always set $sl = 2$ and call this scheme the double-stride optimization and the previous one the single-stride method.

The following list shows the corresponding *MergeTop* in four iterations $x = 1, 2, 3, 4$ with the double-stride optimization. The four output vectors in the left of each row generate two top-middle vectors (in the gray region) of the form (a^4, a^4, a^2, a^2). They can be combined with a down vector (a^0, a^0, a^0, a^0) to produce two middle-down vectors (v_{md1} and v_{md2}) of the form (a^2, a^2, a^0, a^0) for subsequent input vector assembling. Finally, the two top-middle vectors are merged into a top vector. It takes 3 lane-crossing instructions to deal with the top and down vectors, and the lane-crossing instruction is no longer required for the input and output vector reorganization as in the single-stride assembling. We also omit the details of *MergeDown* as it is similar to *MergeTop*. As summarized in Table 1, the total number of data reorganization instructions is reduced to 3 lane-crossing and 8 in-lane instructions for every four vectorized stencil computations.

$$\begin{tikzcd}
x = 1 : (a_1^4, *, a_7^2, *) \arrow[r, "copy"'] & (a_1^4, *, a_7^2, *) \arrow[d] & & (a_{16}^0, a_{15}^0, a_{14}^0, a_{13}^0) = v_{down} \arrow[dl] \arrow[dddl] \\
x = 2 : (a_2^4, *, a_8^2, *) \arrow[r, "shuffle"'] & \boxed{(a_2^4, a_1^4, a_8^2, a_7^2)} \arrow[r, "permute"'] \arrow[dr] & (a_8^2, a_7^2, a_{14}^0, a_{13}^0) = v_{md1} & \\
x = 3 : (a_3^4, *, a_9^2, *) \arrow[r, "copy"'] & (a_3^4, *, a_9^2, *) \arrow[d] & (a_4^4, a_3^4, a_2^4, a_1^4) = v_{top} & \\
x = 4 : (a_4^4, *, a_{10}^2, *) \arrow[r, "shuffle"'] & \boxed{(a_4^4, a_3^4, a_{10}^2, a_9^2)} \arrow[r, "permute"'] \arrow[ur, "permute"'] & (a_{10}^2, a_9^2, a_{16}^0, a_{15}^0) = v_{md2} &
\end{tikzcd}$$

Data parallelism improvement. The temporal vectorization also introduces dependence between input and output vectors in different iterations of the innermost loop. The reason is that there is data dependence along the time dimension and our vectorization scheme incorporates that dependence in iterations of the innermost space loop. For example, after calculating the output vector $(a_1^4, a_3^3, a_5^2, a_7^1)$ of the current iteration $x = 1$, the calculation of a_2^4 in the output vector $(a_2^4, a_4^3, a_6^2, a_8^1)$ of the next iteration $x = 2$ depends on a_3^3. This dependence resembles the common true (read-after-write) dependence. It limits the number of concurrent instructions and may extremely impact performance.

One straightforward approach to increasing the number of concurrent computations for the temporal vectorization of one-dimensional stencils is to widen the space stride s. As Figure 1 shows, the number of input vectors for one-dimensional Jacobi stencils is $s + r$ where r is half of the stencil order. To determine the space stride s, one consideration is about the data reorganization process. Since a top vector groups the value of time coordinate 4 in every four output vectors, the number of available input vectors should be dividable by 4 to simplify the algorithm implementation. Another limitation to the upper bound of s is the number of available vector registers in the CPU. We conducted the experiments on CPUs with the AVX extension where the size of the vector register file is 16. Take the top, down, top-middle, middle-down and coefficients vectors into consideration, we always set the number of available input vectors to 8. Thus for 1D stencils, we set $s = 7$.

3.3 High-dimensional Stencil

The temporal vectorization for one-dimensional stencils in effect performs a strip-mining to the outermost time loop and turns it into two nested time loops. The outer time loop index is incremented by the vector length while the inner time loop traverses a time tile. The temporal vectorization reorders the calculations in the inner time loop and the space loop. For high-dimensional stencils, it is illegal to interchange the inner time loop with space loops, thus it is only allowable to perform the temporal vectorization on the outermost space loop. Consequently, the pre-computation in Line 2-4 of Algorithm 3 forwards grid points in several lines for 2D stencils or planes for 3D stencils 1, 2, or 3 time steps. Figure 2 shows a pictorial view of the temporal vectorization of the 2D5P stencil.

Table 1: Number of Data Organization Operations in Every Four Vector Computations (lane-crossing or in-lane type in brackets)

stride	top	down	output	output + down (middle-down)	output + top (top-middle)	total
single-	1 (in)	2 (in)	4 (cross)	4 (in)	4 (in)	4 (cross) + 11 (in)
double-	1 (cross)	2 (cross)	0	4 (in)	4 (in)	3 (cross) + 8 (in)

The data transfer and vectorized computation in Line 9-19 of Algorithm 3 can be easily extended to higher dimensions. Note that there are additional space loops inside the x loop in Line 8. Therefore the generated input vectors, e.g. $(a^3_{x+s,y}, a^2_{x+2s,y}, a^1_{x+3s,y}, a^0_{x+4s,y})$ in a 2D stencil must be stored to memory for the next iteration of outer space loops, e.g. the computations in $x+s-1$, $x+s$ and $x+s+1$ iterations.

Data layout transpose for high-dimensional stencils. The temporal vectorization of one-dimensional stencils is able to keep the input vectors in CPU vector registers. Consequentially there is no memory transfer for the values with time coordinates 1, 2 and 3 computed in the inner loop. However, as explained above, since the temporal vectorization targets the outermost space loop of a high-dimensional stencil, the input vectors must be stored in memory for subsequent stencil computations. It is desirable to store the input vector contiguously.

Although the values in one input vector are not adjacent in the data space, the values with the same time coordinate (at the same position of these input vectors) are stored contiguously in memory. For example, in four continuous input vectors $(a^3_{x+s,y+i}, a^2_{x+2s,y+i}, a^1_{x+3s,y+i}, a^0_{x+4s,y+i})$ $(i = 0, 1, 2, 3)$, the four values with time coordinate 3, $a^3_{x+s,y+i}$ logically occupy continuous positions. These four input vectors can be contiguously stored in these spaces and therefore can be implemented as a transposed layout. Another approach is to allocate an extra array of relative small size to store the input vectors that will be used in subsequent computations.

Initial input vectors loading (final output vectors storing). The previous optimization improves the storage of the computed input vectors for high-dimensional stencils. The initial input vectors loading (Lines 2-4 in Algorithm 3) and final input vectors storing (Lines 16-18) can be implemented in a similar approach. These values are loaded to vectors by basic vector read instructions (e.g. `_mm256_load_pd`). Each vector contains values with the same time coordinate. Every four input vectors can be obtained by transposing corresponding vectors. Similarly, the final input vector is stored in memory after a transpose.

3.4 Gauss-Seidel Stencil

Gauss-Seidel stencils use the newest neighbor values to update one point. Conventionally it only requires one array to store the latest values of all grid points as opposed to two arrays in Jacobi stencils. Another difference between Jacobi and Gauss-Seidel stencils is that the latter contains data dependencies in all the time and space loops. Thus it is illegal to perform the conventional vectorization.

Nevertheless, the temporal vectorization of Gauss-Seidel stencils is similar to that of Jacobi stencils. One can verify that $s > 1$ is still the correctness condition for Gauss-Seidel stencils. For the neighbors whose newest values are used in the calculation, the temporal vectorization uses their corresponding output vectors. For example, the output vector v_3 in Algorithm 3 is kept and used in the next iteration. The stencil computation is replaced with $Stencil(v_3, v_1, v_2)$ (Line 12). It also leads to the same data reorganization cost to Jacobi stencils. We omit the detailed explanation.

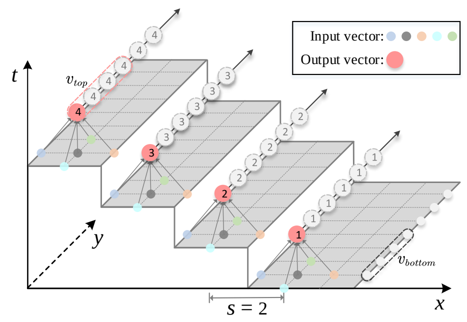

Figure 2: Temporal Vectorization of 2D5P Jacobi Stencil

3.5 Discussion

Compared with the multi-load vectorization method, our temporal vectorization method incurs no redundant data transfer. For one-dimensional stencils each value a^t_x only appears in one input vector. Furthermore, it is easy to align all the top and down vector transfers. For high-dimensional stencils $a^t_{x,y}$ may be loaded in serveral continuous iterations ($y-1$, y and $y+1$ for the 2D5P stencil) of the next-to-innermost space loop, but it still requires fewer data transfers than the multi-load vectorization.

Compared with the data reorganization approach, the number of data reorganization operations in our method is fixed and irrelevant to the vector length, the stencil order, and dimensionality. With the double-stride optimization, the temporal vectorization incurs 0.75 lane-crossing and 2 in-lane instructions. The data reorganization vectorization needs 1 lane-crossing and 2 in-lane instructions for the 1D3P stencil and more for higher-order and dimension stencils.

Compared with the DLT method, the temporal vectorization gives rise to a better reuse of the data in the cache. Though the temporal vectorization also incorporates data transpose operations, it only processes a small set of points and the overhead can be amortized by stencil calculations.

The temporal vectorization, data reorganization and DLT methods achieve the same reuse pattern to the scalar code. For example, it only needs to load one input vector to compute an output vector for the 1D3P stencil. Thus the memory transfer volumes of their blocking implementations are also similar. However, the temporal vectorization leads to slower data reference frequency. Specifically, the straightforward vectorization method touches all the input points in the data space in one time step while the temporal vectorization loads these data in four time steps. It can then expect less memory bandwidth contention, especially in multi-core executions.

Furthermore, for the two arrays in Jacobi stencils, the output data a^4_x and input data a^0_x actually share the same array. The input vectors can be stored in a fixed range of the other array. Thus it actually uses one array in non-blocking Jacobi implementations and only stores the data in two arrays at boundaries for blocking implementations. Therefore the memory transfer volume is reduced by a factor of 2 for Jacobi stencils.

The temporal vectorization would be represented with a set of loop transformations, i.e. the strip-mining of the time loop, the time skewing of the inner time loop and outermost space loop, the loop-peel and finally the outer-loop vectorization. However, there are some difficulties to implement it with general compiler techniques. First, the temporal vectorization needs auxiliary variables, e.g. extra arrays for high-dimensional stencil. These auxiliary data often need manual efforts [38]. Second, it often requires a transposed data layout for efficient vector loads for high-dimensional stencils that complicates the compiler transformations. Third, the data with time coordinates 1, 2 and 3 are reused in CPU registers and not stored in memory for one-dimensional stencils. Conventionally compilers are conservative to store data to memory as performed by the original code. Finally, for high-dimensional stencils, it is illegal to interchange the time loop and spaces loops, this complicates the outer-loop vectorization since it must be applied to a loop that contains more than one inner loop.

Nevertheless, given the temporal vectorization algorithms, it is straightforward to implement a code generator. As a comparison, the DLT method can be viewed as a combination of the strip-mining of the innermost loop and the outer-loop vectorization of the two innermost loops. A domain-specific framework for temporal vectorization of stencil computations can be designed similarly.

The temporal vectorization also resembles the wavefront method [46] (loop skewing). The wavefront method utilizes the parallelogram block shape in temporal blocking. The temporal blocking aims at exploiting the data reuse in caches and reducing the memory transfer volume, while the temporal vectorization primarily serves to lessen the bandwidth pressure between the CPU and cache. Furthermore, the classic correctness condition of blocking [18] in our context for the 1D3P stencil is $s \geq 1$. For example, it is legal to group a_x^{t+1} and a_{x+1}^{t} in one atomic block but calculating them in one vector is not allowable with the temporal vectorization. Thus, the temporal vectorization requires a more strict condition than the temporal blocking.

4 EVALUATION

4.1 Setup

Our experiments were conducted on three machines made of two Intel Xeon E5-2670 processors (totally 24 cores) with 2.3 GHz clock speed, two Intel Xeon Gold 6140 Processors (totally 36 cores) with 2.3 GHz clock speed, two AMD EPYC 7452 Processors (totally 64 cores) with 2.4 GHz clock speed, respectively. We compiled the program with the ICC compiler version 19.1.1 using the optimization flag '-O3 -xavx2' on the two machines with Intel CPUs, and GCC compiler version 9.1.0 with optimization flag '-O3 -mavx2' on the AMD platform.

We employ 6 Jacobi and 4 Gauss-Seidel stencils. Heat-1D, 2D and 3D are most commonly used kernels [7, 17, 43] in the study of optimization of stencils. They are 1D3P, 2D5P and 3D7P star stencils that only contain dependencies on the points along each axis. Laplacian-2D and 3D are 2D9P and 3D27P box stencils [17] that have diagonal data dependencies. We also evaluate a 2D9P box stencil, a particular variant called B2S23 of Conway's Game of Life used in Pluto [7]. Three Gauss-Seidel stencils, GS-1D, 2D and 3D are imported from Pluto [7]. The Longest common subsequence (LCS) is a classic problem solved by the dynamic programming method. It can be implemented as a 1D Gauss-Seidel stencil. The performance is reported using Gstencils/s, i.e. the number of points updated per second.

In the next two subsections the relatively comprehensive results of the temporal vectorization on the Intel Xeon E5-2670 platform are presented and analyzed. In the last subsection we show the results on the other two platform to demonstrate the performance portability.

4.2 Sequential Results

We implemented a sequential code of the temporal vectorization without any blocking scheme for each stencil kernel, The results of sequential codes exhibit the sensitivity to cache bandwidth.

Jacobi Stencils. Figure 3 to 8 show the sequential performance results of Jacobi stencils. The *auto* curves are the results of compiler auto-vectorization where ICC uses the multi-load method. The temporal vectorization of the Heat-1D stencil achieves significant performance improvement over the auto-vectorization and scalar code for problem size larger than 512. However, the auto-vectorization performs better than the temporal vectorization for smaller problem sizes. The reason is the transpose overhead of initial and final input vectors assembling in the temporal vectorization and this overhead can be amortized for larger sizes. The auto-vectorization curve likes a staircase that contains sharp falls at sizes of cache levels. This kind of performance curve is ubiquitous. For stencil computations, this demonstrates that the auto-vectorization is more sensitive to the cache bandwidth than the scalar code. On the contrary, the temporal vectorization produces a flatter curve due to fewer CPU-cache data accesses. It indicates that the temporal vectorization is less sensitive to the cache bandwidth.

The Heat-2D and 3D stencils are star stencils. For continuous output vector calculations, there is no data alignment conflict over outer space loops. Thus the data alignment conflict only exists in the uni-stride space dimension and the auto-vectorization makes an optimal vector utilization for other space dimensions. The benefits of the temporal vectorization may be outweighted by its downsides. For sequential results, the temporal vectorization still incur flatter curves than the auto-vectorization. It obtains competitive and worse performance for sizes smaller than last-level cache compared with the auto-vectorization for Heat-2D and 3D stencil, respectively. This again implies the better bandwidth utilization of the temporal vectorization. As demonstrated by existing work, the 3D7P Jacobi stencil exhibits limited improvements on new parallelization [2] and vectorization schemes [17]. Our results of Heat-3D reveal a similar phenomenon.

The Laplacian-2D, 3D and Life stencils are box stencils. The auto-vectorization leads to data alignment conflicts on all space dimensions. Therefore the temporal vectorization achieves visible and similar improvements for them. The Life stencil performs more operations than the Laplacian-2D stencil. Thus the auto-vectorization is less sensitive to cache bandwidth as shown in the figure 8.

Table 2 lists the number of arithmetic operations, data reorganization instructions and memory accesses. The left columns show the analytical results per vector computation. The second column exhibits the number of arithmetic instructions and Flops (in brackets). In our temporal vectorization, Heat-1D utilizes the double-stride optimization and incurs 0.75 lane-crossing and 2 in-lane instructions according to Table 1. All other stencils use the single-stride method and lead to 1 lane-crossing and 2.75 in-lane instructions per vector computation.

Table 2: Analytical and Measured Numbers of Vector Instructions ($\times 10^9$) in Sequential Executions of Floating Point Arithmetic Jacobi Stencils.

	analytical number per vector computation						Problem	#vector	measured number (corresponding analytical number in bracket) ($\times 10^9$)					
	arith.	our			auto		Input	computations	arith.	our			auto	
	(Flops)	organ.	load	store	load	store	Size	($\times 10^9$)		organ.	load	store	load	store
Heat-1D	3 (4)	0.75+2	0.25	0.25	3	1	$16M \times 6K$	24	72.2 (72)	78.3 (66)	6.3 (6)	6.2 (6)	72.4 (72)	24.2 (24)
Heat-2D	8 (10)	1+2.75	3.25	1.25	5	1	$8K^2 \times 2K$	32	257 (256)	131 (120)	121 (104)	41.4 (40)	161 (160)	33.2 (32)
Heat-3D	11 (15)	1+2.75	5.25	1.25	7	1	$800^3 \times 200$	25.6	285 (281)	107 (96)	160 (134)	34.1 (32)	183 (179)	26.5 (25.6)
Lapl-2D	9 (10)	1+2.75	3.25	1.25	9	1	$8K^2 \times 2K$	32	295 (288)	136 (120)	119 (104)	40.8 (40)	287 (288)	32.7 (32)
Lapl-3D	27 (30)	1+2.75	9.25	1.25	27	1	$800^3 \times 200$	25.6	695 (691)	116 (96)	312 (236)	38.5 (32)	712 (691)	31.2 (25.6)

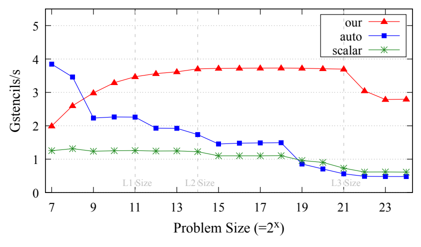

Figure 3: Heat-1D

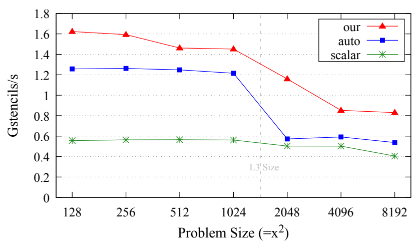

Figure 4: Heat-2D

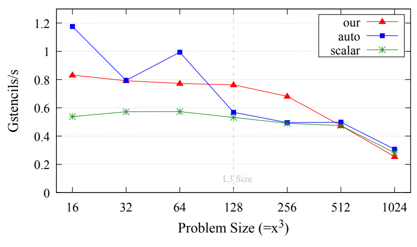

Figure 5: Heat-3D

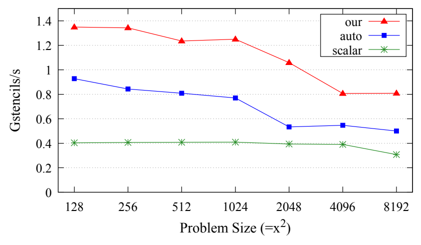

Figure 6: Laplacian-2D

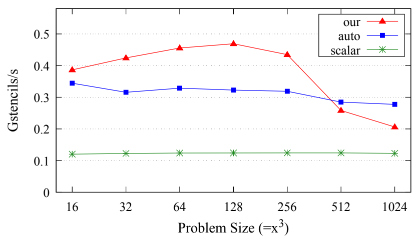

Figure 7: Laplacian-3D

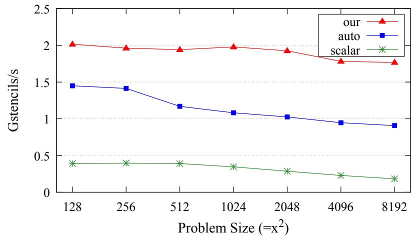

Figure 8: Life

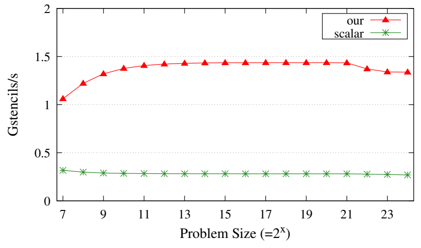

Figure 9: GS-1D

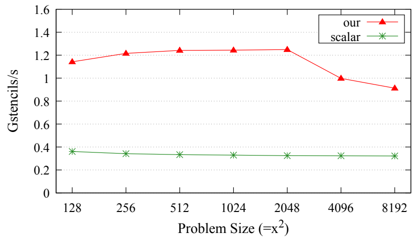

Figure 10: GS-2D

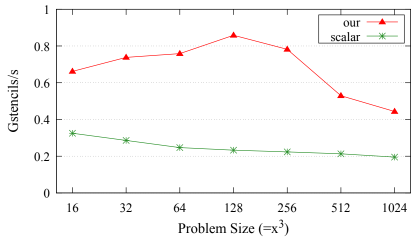

Figure 11: GS-3D

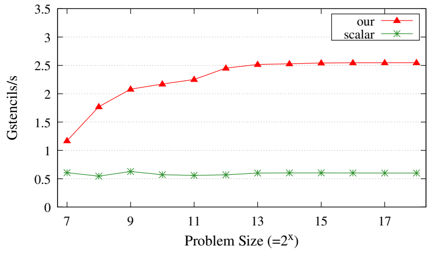

Figure 12: LCS

The middle columns show the problem sizes. The right columns present the measured and analytical numbers (in brackets). The analytical number equals the corresponding value multiplied by the number of vector computations listed in the middle column. There are close matches between the measured and analytical numbers. The temporal vectorization incurs a larger total number of instructions except arithmetic ones than the auto-vectorization. However, the data reorganization instructions are executed in an independent execution unit in Intel microarchitectures. Furthermore, although the number of memory stores is larger with the temporal vectorization, the total number of memory accesses is smaller.

Gauss-Seidel Stencils. Figure 9 to 12 exhibits the sequential performance results of Gauss-Seidel stencils. The temporal vectorization achieves significant performance improvements for all Gauss-Seidel stencils over the scalar codes. All sequential executions of the scalar codes without blocking achieve a similar absolute performance of around 0.4 Gstencils/s. This demonstrates that Gauss-Seidel stencils are limited by data dependencies.

For the 1D sequential execution, it obtains a super-linear speedup of up to 4.4. The reason is that the temporal vectorization leads to better utilization of the memory bandwidth. The curve is similar to that of the Head-1D Jacobi stencil. For smaller problem size the *scalar ratio*, i.e. the percentage of points processed with scalar arithmetic operations is larger. For example, with the space stride

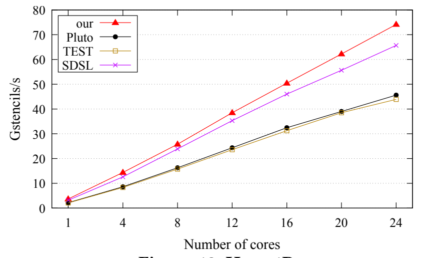

Figure 13: Heat-1D

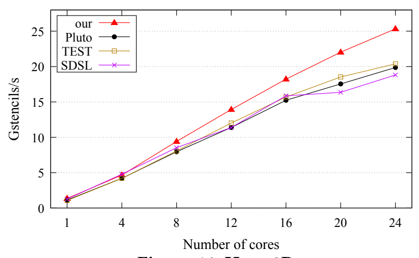

Figure 14: Heat-2D

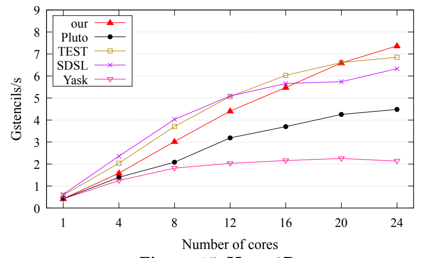

Figure 15: Heat-3D

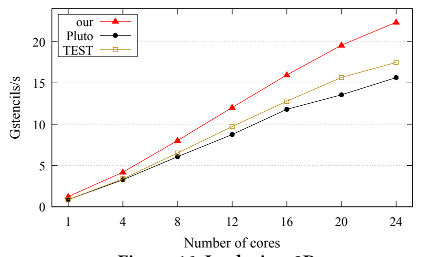

Figure 16: Laplacian-2D

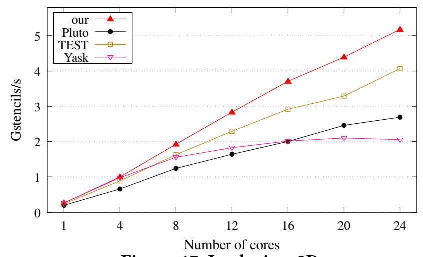

Figure 17: Laplacian-3D

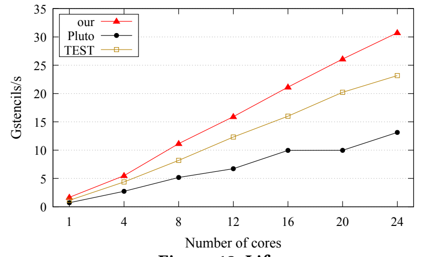

Figure 18: Life

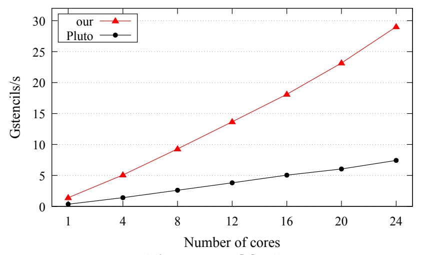

Figure 19: GS-1D

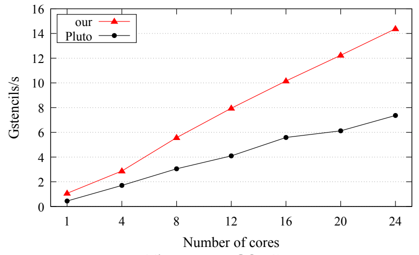

Figure 20: GS-2D

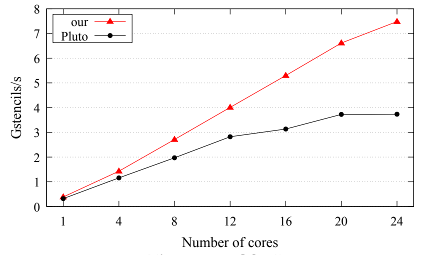

Figure 21: GS-3D

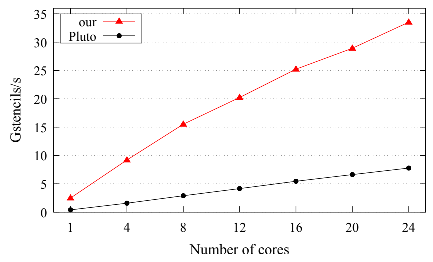

Figure 22: LCS

Table 3: Blocking Sizes and Performance of Gauss-Seidel Stencils on 24-core Executions

	Input Size	Our Blocking	Pluto Blocking	Our Gstencils	Pluto Gstencils	Speedup over Pluto
GS-1D	$16M \times 6K$	2048×64	256×128	29.0	7.43	3.90
GS-2D	$8K^2 \times 2K$	$128^2 \times 32$	$32^2 \times 32$	14.3	7.36	1.95
GS-3D	$800^3 \times 200$	$32^3 \times 32$	$8^3 \times 8$	7.48	3.73	2.00
LCS	$200K \times 200K$	4096×4096	128×128	33.5	7.76	4.31

$s = 7$, there are 84 points in a time tile need to be calculated by the scalar codes (Lines 2-4 and 24-27 in Algorithm 3). This leads to a scalar ratio of 16% for problem size $NX = 128$ and 1% for $NX = 2048$.

For higher-dimensional Gauss-Seidel stencils, we see similar trends in sequential results but more sharp performance decreases when the problem size is larger than the L3 cache size. However, the in-cache performance is relatively steady and it demonstrates the less sensitivity to the cache bandwidth of the temporal vectorization. The maximal speedup is 3.8 for the 2D stencil with a problem size 2048^2 and 3.6 for the 3D stencil with a problem size 128^3, respectively.

The LCS stencil result is similar to that of the 1D Gauss-Seidel stencil. It processes integer values with integer SIMD instructions and has a theoretical maximal speedup of 8. The scalar code performs either $max(lcs[x-1][y], lcs[x][y-1])$ or $lcs[x-1][y-1]+1$ for calculating $lcs[x][y]$ depending on the equality of two characters $A[x]$ and $B[y]$. However, the temporal vectorization needs to execute all these computations and obtains the correct vector by a blend instruction with a mask vector of equalities. Thus we would expect smaller speedups compared with the 1D Gauss-Seidel stencil. Nevertheless, the temporal vectorization still achieves good improvements and yields a maximal speedup of 4.3 over the scalar code.

4.3 Parallel Results

We also implemented parallel codes with a diamond and parallelogram [47] hybrid tiling for Jacobi stencils. It is illegal to employ diamond tiling for Gauss-Seidel stencils. Thus we utilize parallelogram tiling for all space dimensions of Gauss-Seidel stencils. The parallel codes are implemented with OpenMP and scaled from unicore to all 24 cores. We exhaustively tested all blocking sizes and

Table 4: Performane and Measured Hardware event counts ($\times 10^9$) of Jacobi Stencils on 24-core Executions

		Blocking size	Gstencils	our speedup	memory accesses	L2 misses	L3 misses
Heat-1D	our	16384 × 128	74.1	1	15.3	0.027	0.015
	Pluto	2048 × 2048	45.6	1.63	100	0.008	0.002
	TEST	2048 × 128	43.9	1.69	100	0.085	0.035
	SDSL	512 × 256	65.7	1.13	102	0.104	0.045
Heat-2D	our	256 × 256 × 64	25.3	1	186	2.11	0.49
	Pluto	64 × 64 × 64	19.8	1.28	278	1.08	0.10
	TEST	128 × 256 × 64	20.3	1.25	211	18.0	0.41
	SDSL	128 × 128 × 64	18.8	1.35	240	18.6	0.53
Heat-3D	our	64 × 16 × 64 × 32	7.37	1	440	24.3	4.13
	Pluto	16 × 16 × 16 × 16	4.47	1.65	615	11.7	5.35
	TEST	32 × 32 × 64 × 16	6.85	1.08	312	33.9	3.83
	SDSL	16 × 16 × 128 × 8	6.33	1.16	320	24.4	3.08
Lapl-2D	our	256 × 256 × 64	22.3	1	205	2.48	0.65
	Pluto	64 × 64 × 64	15.6	1.43	425	1.05	0.01
	TEST	128 × 256 × 64	17.4	1.28	361	12.9	0.42
Lapl-3D	our	64 × 16 × 64 × 32	5.17	1	933	31.4	4.23
	Pluto	12 × 12 × 12 × 12	2.65	1.95	1497	9.97	6.94
	TEST	32 × 32 × 64 × 16	4.07	1.27	943	36.6	3.74
Life	our	256 × 256 × 32	30.7	1	138	1.07	0.38
	Pluto	128 × 128 × 128	13.1	2.34	381	0.74	0.074
	TEST	128 × 128 × 32	23.1	1.33	234	1.21	0.74

show the one producing the best performance. However, we observed that the performance is very sensitive to the tile sizes, but this requires significant effort in auto-tuning and should be done separately.

Jacobi Stencils. Figure 13 to 18 show the parallel results of Jacobi stencils. The problem sizes are identical to the ones listed in Table 2. For Heat stencils, we compared the temporal vectorization with Pluto [2], SDSL [17] and TEST [52]. Pluto and TEST employ auto-vectorization while SDSL utilizes the DLT vectorization scheme. Table 4 lists the blocking sizes and the performance, the number of memory accesses and cache misses on 24-core executions. We also provide the results of Yask [51] that implements vector-folding [50] for 3D Jacobi stencils. Note that Yask has an auto-tuning scheme so we omit the blocking sizes and profiling results for it.

For Heat-1D, all methods produce similar scalabilities with speedup around 20x for 24-core over uni-core. On 24-core execution, the speedups of the temporal vectorization are 1.62x, 1.68x and 1.12x over Pluto, TEST and SDSL, respectively. The temporal vectorization favors much larger blocking sizes as it extremely decreases the memory accesses. We also see a similar data locality, total memory accesses around 100×10^9 for other methods.

The temporal vectorization obtains visible performance improvement for Heat-2D but only comparable results for Heat-3D. This trend is consistent with the sequential performance results. The reason is similar, i.e. the auto-vectorization only incurs data alignment conflicts over the unit-stride dimension. The event counts in Table 4 also demonstrate that for higher dimension star stencils the temporal vectorization leads to similar or higher memory accesses numbers for 2D or 3D Heat stencil. It is difficult to connect the event counts with the final performance with a simple model. However, the numbers still exhibit similar results to those in sequential executions and are consistent with the Gstencils.

The temporal vectorization obtain obvious speedups for box stencils as shown in Figure 16 to 18. SDSL is unable to deal with the life stencil since it contains conditional statements. We also failed to generate correct SDSL codes for the Laplacian-2D and 3D kernels. The scalabilities are similar to that of Heat-2D and 3D stencils. They both have a decline in the 24-core execution for Pluto and TEST.

Table 5: Performane Results on Intel Xeon Gold 6140 (36 cores)

	our	Pluto	TEST	SDSL	Speedup over Pluto	Speedup over TEST	Speedup over SDSL
Heat-1D	110	79.3	70.0	89.2	1.39	1.57	1.23
Heat-2D	41.1	33.6	38.1	33.5	1.22	1.08	1.22
Heat-3D	9.9	6.0	7.4	8.4	1.65	1.34	1.17
Lapl-2D	30.3	24.4	28.1	-	1.25	1.07	-
Lapl-3D	7.3	4.4	4.9	-	1.66	1.49	-
Life	49.3	20.5	40.2	-	2.45	1.23	-
GS-1D	26.2	8.2	-	-	3.19	-	-
GS-2D	20.9	11.4	-	-	1.83	-	-
GS-3D	10.2	5.4	-	-	1.89	-	-
LCS	58.8	14.3	-	-	4.11	-	-

The temporal vectorization produces better scalabilities for both stencils due to fewer data accesses as demonstrated in Table 4. Note that Pluto incurs the most memory accesses due to its complicated loop control variables, while obtain the fewest cache misses for all stencils. In both 3D stencils, Yask leads to the lowest performance. It is suitable for higher-order stencils where the folding of vectors incurs less memory footprint.

Gauss-Seidel Stencils. Figure 19 to 22 shows the parallel results of Gauss-Seidel stencils. We only compared with Pluto since SDSL and TEST are not applicable to Gauss-Seidel stencils. Pluto can only generate scalar codes. Table 3 lists the blocking sizes used in the temporal vectorization and Pluto, and the performance results in the 24-core executions.

For GS-1D stencil both the temporal vectorization and Pluto achieve good scalabilities. Compared with the single-core execution, the 24-core results reach speedups of 20.7 and 19.6 for the vectorization and Pluto code, respectively. These speedup results are comparable to those of Jacobi stencils and demonstrate that the parallelogram tiling provides competitive scalabilities. The temporal vectorization delivers an average speedup of 3.5 and a maximal speedup of 3.9 over Pluto. For LCS, the speedup on 24-core execution is 4.31 over Pluto.

For the 2D and 3D stencil, the temporal vectorization achieves better scalabilities than Pluto. For example, the speedup of 24-core over a single core of the GS-3D is 11.7 for Pluto and 19.3 for the temporal vectorization. For the method comparison, the temporal vectorization obtains an average speedup of 1.94 and 1.53 for 2D and 3D stencils, respectively.

4.4 Performance Portability

Table 5 and 6 show the results on the machines with two Intel Xeon Gold 6140 CPUs (totally 36 cores) and with two AMD EPYC 7452 CPUs (totally 64 cores), respectively. The blocking sizes are similar to those used above.

Table 6: Performane Results on AMD EPYC 7452 (64 cores)

	our	Pluto	TEST	SDSL	Speedup over Pluto	Speedup over TEST	Speedup over SDSL
Heat-1D	222	186	166	176	1.19	1.34	1.26
Heat-2D	71.2	63.7	57.4	60.1	1.13	1.25	1.18
Heat-3D	11.5	12.6	7.1	11.7	0.91	1.61	0.98
Lapl-2D	66.6	61.4	60.5	-	1.08	1.10	-
Lapl-3D	7.5	4.7	6.2	-	1.60	1.25	-
Life	80.4	42.3	43.3	-	1.90	1.86	-
GS-1D	38.4	8.2	-	-	4.80	-	-
GS-2D	35.8	6.5	-	-	5.51	-	-
GS-3D	12.1	3.4	-	-	3.55	-	-
LCS	81.3	22.8	-	-	3.57	-	-

The speedups on the Intel Xeon Gold 6140 CPU are similar to those on the Intel Xeon E5-2670 processor. For the Heat-1D stencil, SDSL still achieves the best performance among the three compared counterparts. The temporal vectorization obtains a speedup of 1.23 over SDSL, which is higher than 1.13 on the Intel Xeon E5-2670 CPUs. For high-dimensional stencils, the temporal vectorization achieves similar speedups over Pluto and SDSL but smaller speedups over TEST on Heat-2D and Heat-3D stencils. For box stencils, Lapl-2D, Lapl-3D and Life, the speedups on both Intel machines are comparable. The temporal vectorization obtains lower speedups on the newer Intel CPUs for all Gauss-Seidel stencils. The reason may be the improved pipelined execution efficiency in the latter CPU and consequently the temporal vectorization derives fewer improvements over scalar codes. However, it still achieves visible performance increases and it demonstrates the effectiveness and performance portability on both Intel CPUs.

The performance on the AMD machine seems a little different from that on Intel platforms. On the AMD machine, the temporal vectorization obtains lower speedups for all Jacobi stencils. The reason is that the parallelism of Pluto, TEST and SDSl makes better utilization of the 64 cores than ours. In particular, they all utilize a diamond-like blocking scheme while our blocking scheme will incur a pipelined start-up and drain phase. Thus For Heat-1D, Lapl-3D and Life stencils, the improvements are still visible and comparable to those on the Intel platforms. However, for Heat-2D, Heat-3D and Lapl-2D stencils, Pluto, TEST or SDSL is competitive or even faster than our scheme.

On the contrary, the temporal vectorization obtains higher speedups for all Gauss-Seidel stencils on the AMD machine. The reason is two-fold. First, all methods must employ the parallelogram tiling scheme for Gauss-Seidel stencils and they exhibit similar parallelism. Our blocking scheme will not suffer from concurrency loss compared with other schemes as in the Jacobi stencils. Second, the microarchitecture pipeline implementation may be inferior to Intel CPUs, which benefits the temporal vectorization to achieve super-linear speedups for GS-1D and GS-2D.

5 RELATED WORK

Tiling [24, 45] is regarded as a technique to improve the data locality. However, better data locality generally gives rise to preferable parallelism between blocks. This parallelism is coarse-grained since tiling targets the cache level and groups a large number of data elements. Vectorization, on the contrary, is to exploit fine-grained parallelism and group small data elements in vector registers. In sum, tiling and vectorization concentrates the cache level and CPU microarchitecture level, and explores inter-block and intra-register parallelism, respectively. On the other hand, tiling and vectorization share similar goals and fundamental analytical approaches [49], especially for the temporal tiling and temporal vectorization. Our work shows a stricter condition in data dependence analysis for the temporal vectorization. However, a thorough comparison between them is beyond the scope of this paper.

The compiler community has been studying sophisticated universal vectorization techniques [1, 15, 21, 25, 32, 40]. Previous work [13, 26, 48] has proposed many solutions to address unaligned vector transfers. There are also many studies focusing on reducing the data preparation overhead [16, 36, 55]. The DLT [16] assembles points in the unit-stride space dimension that are free of intra-vector read-read dependencies in one vector. Other more general data organization optimizations include data alignment optimization [6] and data interleaving [31, 55], However, some of these works are too general to capture the specific properties of stencils. For example, the data alignment conflicts for stencil refer to overlapped vector loads, it is unable to attack this problem by stream shifts [13].

The state-of-art compilers usually vectorize the innermost loop. Outer-loop vectorization techniques [32] often focus on the case where the innermost loop is illegal to be vectorized. One relaxed legality condition of the outer-loop vectorization is that it is interchangeable with the inner loop. However, the time loop is not interchangeable with space loops according to the data dependencies. Though more strict conditions exist, direct outer-loop vectorization at the time loop is still illegal. For slightly complicated codes, outer-loop vectorization requires auxiliary arrays and programmers need to explicitly declares them [38]. Some outer-loop vectorization techniques require the iteration number of the inner loop is invariant to the outer-loop. However, many blocked stencil algorithms [14, 17, 43, 52] shrink or expand the data range in all space dimensions as the time proceeds. For example, the classic diamond tiling [23] introduces a diamond shape in the iteration space for a one-dimensional stencil. It enlarges the data space in the first half time and then decreases it in the rest time.

There exists a lot of work on improving the register reuse by utilizing the associative property of stencil calculations. The fundamental idea is to find a better order of statements across iterations. Deitz et al [10] proposes a compiler formulation and transformation called array common subexpression elimination. A similar idea called partial sum is also studied in [3]. Cruz and Araya-polo [9] and Stock et al [41] combine the gather (update an output element with all its neighbors) and scatter (update all its neighbors with one input element) patterns in one and many space dimensions, respectively. Several studies [19, 27, 35, 53] describe register reuse frameworks for GPUs. Yount [50] proposed a vector folding method to group points in the entire data space rather than a single dimension. However, prior work either targets scalar registers for high-order stencils on GPUs or specific stencils with constant and symmetrical coefficients. They only consider the reordering in one time step and data alignment conflicts are inevitable with vectorization. Furthermore, these methods are not applicable to and examined for Gauss-Seidel stencils. The temporal vectorization preserves the same calculation order to the scalar code. Thus we believe this work can be incorporated into our scheme.

Another branch of work focuses on the multigrid method [22, 39]. Douglas et al [12] studied the locality exploration. Kazhdan and Hoppe [20] proposed the streaming concept, a schedule of blocks to improve both inter-block locality and parallelism. Park et al [33] utilizes a reordering scheme to explore parallelism. Bell et al [5] studied the fine-grained parallelism in multigrid method using a set of parallel primitives. For vectorization, most work either depends on the compiler auto-vectorization directly [4, 42] or employs simple vectorization schemes. For example, in [44], the straightforward SIMDization scheme for Gauss-Seidel stencils "wastes half the compute capability" on both CPU and KNC. For the multigrid method, the temporal vectorization is only applicable to the update inside each grid level since the data sizes (the spatial space) of grids with various granularities are different.

In sum, to the best of our knowledge, we are unaware of any vectorization scheme along the time dimension in existing stencil and multi-grid literature.

6 CONCLUSION

We have presented a new temporal vectorization scheme. It vectorizes the stencil computation in the whole iteration space and assembles points with different time coordinate in one vector. The temporal vectorization leads to a small fixed number of vector reorganizations that is irrelevant to the vector length, stencil order, and dimension. Furthermore, it is also applicable to Gauss-Seidel stencils. The effectiveness of the temporal vectorization is demonstrated by various Jacobi and Gauss-Seidel stencils. Future work will design a framework to automatically generate the stencil codes. We will also design an auto-tuning method to efficiently search the best block size.

Our ongoing work includes: (1) the temporal vectorization extension on more complicated stencils like Finite-difference time-domain (FDTD), Lattice Boltzmann methods (LBM), red-black Gauss-Seidel and variable coefficient stencils, (2) a domain-specific compiler that automatically generates the temporal blocking and vectorization codes.

ACKNOWLEDGMENTS

The authors would like to thank all the reviewers for their valuable comments. 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] Randy Allen and Ken Kennedy. 1987. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 4 (1987), 491–542.

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

[3] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-Directed Transformation for Higher-Order Stencils *(IPDPS '15)*. 313–323.

[4] Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, Phillip Colella, and Mary Hall. 2017. Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers. *Parallel Comput.* 64 (2017), 50–64. https://doi.org/10.1016/j.parco.2017.04.002 High-End Computing for Next-Generation Scientific Discovery.

[5] Nathan Bell, Steven Dalton, and Luke N. Olson. 2012. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152. https://doi.org/10.1137/110838844

[6] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. 2002. Automatic Intra-Register Vectorization for the Intel Architecture. *Int. J. Parallel Program.* 30, 2 (April 2002), 65–98.

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

[8] Diego Caballero, Sara Royuela, Roger Ferrer, Alejandro Duran, and Xavier Martorell. 2015. Optimizing overlapped memory accesses in user-directed vectorization. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 393–404.

[9] 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.

[10] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. 2001. Eliminating Redundancies in Sum-of-product Array Computations *(ICS '01)*. 65–77.

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

[12] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rude, and Christian Weiss. 2000. Cache optimization for structured and unstructured grid multigrid. *ETNA. Electronic Transactions on Numerical Analysis* 10 (2000), 21–40. http://eudml.org/doc/120653

[13] Alexandre E Eichenberger, Peng Wu, and Kevin O'brien. 2004. Vectorization for SIMD architectures with alignment constraints. *Acm Sigplan Notices* 39, 6 (2004), 82–93.

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

[15] Mark Hampton and Krste Asanovic. 2008. Compiling for vector-thread architectures. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 205–215.

[16] 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 *(CC'11/ETAPS'11)*. 225–245.

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

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

[19] Mengyao Jin, Haohuan Fu, Zihong Lv, and Guangwen Yang. 2016. Libra: An Automated Code Generation and Tuning Framework for Register-limited Stencils on GPUs *(CF '16)*. 92–99.

[20] Michael Kazhdan and Hugues Hoppe. 2008. Streaming Multigrid for Gradient-Domain Operations on Large Images. *ACM Trans. Graph.* 27, 3 (Aug. 2008), 1–10. https://doi.org/10.1145/1360612.1360620

[21] Ken Kennedy and John R. Allen. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[22] Markus Kowarschik, Christian Weib, Wolfgang Karl, and Ulrich Rude. 2000. Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions. *Computing* 64, 4 (July 2000), 381–399. https://doi.org/10.1007/s006070070032

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

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

[25] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 145–156.

[26] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. 2002. Increasing and detecting memory address congruence. In *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 18–29.

[27] Wen-Jing Ma, Kan Gao, and Guo-Ping Long. 2016. Highly Optimized Code Generation for Stencil Codes with Computation Reuse for GPUs. *Journal of Computer Science and Technology* 6, 31 (2016), 1262–1274.

[28] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. 2011. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 372–382.

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

[30] 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.

[31] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 132–143.

[32] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2–11.

[33] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D. Kalamkar, Xing Liu, Md. Mosotofa Ali Patwary, Yutong Lu, and Pradeep Dubey. 2014. Efficient Shared-Memory Implementation of High-Performance Conjugate Gradient Benchmark and Its Application to Unstructured Matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New Orleans, Louisana) *(SC '14)*. IEEE Press, 945–955. https://doi.org/10.1109/SC.2014.82

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

[35] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register Optimizations for

Stencils on GPUs *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 168–182.

[36] Gang Ren, Peng Wu, and David Padua. 2006. Optimizing data permutations for SIMD devices. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 118–131.

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

[38] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. 2012. Can traditional programming bridge the Ninja performance gap for parallel computing applications?. In *ISCA*. 440–451.

[39] Sriram Sellappa and Siddhartha Chatterjee. 2001. Cache-Efficient Multigrid Algorithms. In *Proceedings of the International Conference on Computational Sciences-Part I (ICCS '01)*. Springer-Verlag, Berlin, Heidelberg, 107–116.

[40] Narasimhan Sreraman and Ramaswamy Govindarajan. 2000. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming* 28, 4 (2000), 363–400.

[41] 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 *PLDI*. 65–76.

[42] Hari Sundar and Omar Ghattas. 2015. A Nested Partitioning Algorithm for Adaptive Meshes on Heterogeneous Clusters. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 319–328. https://doi.org/10.1145/2751205.2751246

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

[44] Samuel Williams, Dhiraj D. Kalamkar, Amik Singh, Anand M. Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. 2012. Optimization of geometric multigrid for emerging multi- and manycore processors. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.85

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

[46] Michael Wolfe. 1986. Loop skewing: the wavefront method revisited. *International Journal of Parallel Programming* 15, 4 (1986), 279–293.

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

[48] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. 2005. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, USA, 153–164.

[49] Jingling Xue. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, USA.

[50] Charles Yount. 2015. Vector Folding: Improving Stencil Performance via Multi-Dimensional SIMD-Vector Representation *(HPCC-CSS-ICESS '15)*. IEEE Computer Society, USA, 865–870.

[51] 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 *Proceedings of the Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC* (Salt Lake City, Utah) *(WOLFHPC '16)*. IEEE Press, 30–39.

[52] 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*. 1–13.

[53] 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 *SC*. 1–44.

[54] Hao Zhou and Jingling Xue. 2016. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–26.

[55] Hao Zhou and Jingling Xue. 2016. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *CGO*. IEEE, 59–69.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Our experiments were conducted on a machine made of two Intel Xeon E5-2670 processors with 2.30 GHz clock speed. We compiled the program with the ICC compiler version 19.1.1, using the optimization flag '-O3 -xHost'.

We employ 6 Jacobi and 4 Gauss-Seidel stencils. Heat-1D, 2D and 3D are most commonly used kernels in the study of optimization of stencils. They are 1D3P, 2D5P and 3D7P star stencils that only contain dependencies on the points along each axis. Laplacian-2D and 3D are 2D9P and 3D27P box stencils that have diagonal data dependencies. We also evaluate a 2D9P box stencil, a particular variant called B2S23 of Conway's Game of Life used in Pluto. Three Gauss-Seidel stencils, GS-1D, 2D and 3D are imported from Pluto. The Longest common subsequence (LCS) is a classic problem solved by the dynamic programming method. It can be implemented as a 1D Gauss-Seidel stencil. The performance is reported using Gstencils/s, i.e. the number of points updated per second.

The compared software:
- SDSl: http://hpcrl.cse.ohio-state.edu/sdslc/sdslc-0.3.2.tar.gz
- Pluto: https://github.com/bondhugula/pluto/
- TEST: https://github.com/yuan-liang/testencil
- Yask: https://github.com/intel/yask

Author-Created or Modified Artifacts:

```
Persistent ID: 10.5281/zenodo.5151491
Artifact name: VecTime
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz

Operating systems and versions: CentOS release 8.2.2004, Linux kernel 4.18.0

Compilers and versions: icc version 19.1.1.217