

# A Unified Approach to Variable Renaming for Enhanced Vectorization

Prasanth Chatarasi<sup>1</sup>, Jun Shirako<sup>1</sup>, Albert Cohen<sup>2</sup>, and Vivek Sarkar<sup>1</sup>

<sup>1</sup> Georgia Institute of Technology, Atlanta GA, USA

{cprasanth, shirako, vsarkar}@gatech.edu

<sup>2</sup> INRIA & DI ENS, Paris, France

albert.cohen@inria.fr

**Abstract.** Despite the fact that compiler technologies for automatic vectorization have been under development for over four decades, there are still considerable gaps in the capabilities of modern compilers to perform automatic vectorization for SIMD units. One such gap can be found in the handling of loops with dependence cycles that involve memory-based anti (write-after-read) and output (write-after-write) dependences. Past approaches, such as variable renaming and variable expansion, break such dependence cycles by either eliminating or repositioning the problematic memory-based dependences. However, the past work suffers from three key limitations: 1) Lack of a unified framework that synergistically integrates multiple storage transformations, 2) Lack of support for bounding the additional space required to break memory-based dependences, and 3) Lack of support for integrating these storage transformations with other code transformations (e.g., statement reordering) to enable vectorization.

In this paper, we address the three limitations above by integrating both Source Variable Renaming (SoVR) and Sink Variable Renaming (SiVR) transformations into a unified formulation, and by formalizing the “cycle-breaking” problem as a minimum weighted set cover optimization problem. To the best of our knowledge, our work is the first to formalize an optimal solution for cycle breaking that simultaneously considers both SoVR and SiVR transformations, thereby enhancing vectorization and reducing storage expansion relative to performing the transformations independently. We implemented our approach in PPCG, a state-of-the-art optimization framework for loop transformations, and evaluated it on eleven kernels from the TSVC benchmark suite. Our experimental results show a geometric mean performance improvement of  $4.61\times$  on an Intel Xeon Phi (KNL) machine relative to the optimized performance obtained by Intel’s ICC v17.0 product compiler. Further, our results demonstrate a geometric mean performance improvement of  $1.08\times$  and  $1.14\times$  on the Intel Xeon Phi (KNL) and Nvidia Tesla V100 (Volta) platforms relative to past work that only performs the SiVR transformation [5], and of  $1.57\times$  and  $1.22\times$  on both platforms relative to past work on using both SiVR and SoVR transformations [8].

**Keywords:** Vectorization · Renaming · Storage transformations · Polyhedral compilers · Intel KNL · Nvidia Volta · TSVC Suite · SIMD.

## 1 Introduction

There is a strong resurgence of interest in vector processing due to the significant energy efficiency benefits of using SIMD parallelism within individual CPU cores as well as in streaming multiprocessors in GPUs. These benefits increase with widening SIMD vectors, reaching vector register lengths of 512 bits in the Intel Xeon Phi Knights Landing (KNL) processor, Intel Xeon Skylake processor and 2048 bits in the scalable vector extension of the Armv8 architecture [18]. Further, there is a widespread expectation that compilers will continue to play a central role in handling the complexities of dependence analysis, code transformation and code generation necessary for vectorization for CPUs. Even in cases where the programmer identifies a loop as being vectorizable, the compiler still plays a major role in transforming the code to use SIMD instructions. This is in contrast with multicore and distributed-memory parallelism (and even with GPU parallelism in many cases), where it is generally accepted that programmers manually perform the code transformations necessary to expose parallelism, with some assistance from the runtime system but little or no help from compilers. It is therefore important to continue advancing the state of the art of vectorizing compiler technologies, so as to address the growing needs for enabling modern applications to use the full capability of SIMD units.

This paper focuses on advancing the state of the art with respect to handling *memory-based anti* (write-after-read) or *output* (write-after-write) dependences in vectorizing compilers. These dependences can theoretically be eliminated by allocating new storage to accommodate the value of the first write operation thereby ensuring that the following write operation need not wait for the first write to complete. However, current state-of-the-art vectorizing compilers only perform such storage transformations in limited cases, and often fail to vectorize loops containing cycles of dependences that include memory-based dependences. This is despite a vast body of past research on storage transformations, such as variable renaming [15, 13, 7, 14] and variable expansion [10], which have shown how removing storage-related dependences can make it possible to “break” dependence cycles.

We believe that the limited use of such techniques in modern compilers is due to three key limitations that currently inhibit their practical usage:

1. Lack of a unified framework that synergistically integrates multiple storage transformations,
2. Lack of support for bounding the additional space required to break memory-based dependences, and
3. Lack of support for integrating these storage transformations with other code transformations (e.g., statement reordering) to enable vectorization.

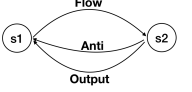
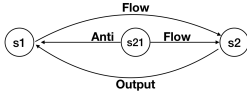
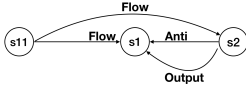
The goal of this paper is to enhance the current state-of-the-art in vectorizing compilers to enable more loops to be vectorized via systematic storage transformations (variable renamings) that remove selected memory-based dependences to break their containing cycles, while optionally using a bounded amount of additional space. We view our tool, called *PolySIMD*, as an extension to vectorization technologies that can be invoked when a state-of-the-art

vectorizer fails to vectorize a loop. Thus, we do not focus on replicating all state-of-the-art vectorization capabilities in *PolySIMD*. For example, we focus on enabling vectorization of innermost loops in *PolySIMD*, though many state-of-the-art compilers support outer loop vectorization as well (and we believe that our contributions can also be applied to outer loop vectorization). By default, our tool takes sequential code as input, and focuses on identifying the best use of variable renamings to maximize opportunities for vectorization. An input loop can optionally be annotated with a pragma that specifies a bound (*spacelimit*) on the maximum amount of extra storage that can be allocated to break dependences. As discussed later, the two main variable renaming transformations that we employ in our approach are *Source Variable Renaming* (SoVR) and *Sink Variable Renaming* (SiVR).

The main technical contributions of this paper are as follows:

- We formalize the problem of identifying an optimized set of SoVR and SiVR variable renaming transformations to break cycles of dependences as a minimum weighted set cover optimization problem, and demonstrate that it is practical to use ILP formulations to find optimal solutions to this problem. If the user provides an optional *spacelimit* parameter, our formalization ensures that the additional storage introduced by our transformations remains within the user-provided bounds.
- We created a new tool, *PolySIMD*, to implement our approach by selecting and performing an optimal set of SoVR and SiVR transformations, along with supporting statement reordering transformations. Given an input sequential loop, *PolySIMD* either generates transformed sequential CPU code that can be input into a vectorizing compiler like ICC or generates GPU code (CUDA kernels) that can be processed by a GPU compiler like NVCC. *PolySIMD* is implemented as an extension to the PPCG framework [20, 1], so as to leverage PPCG’s dependence analysis and code generation capabilities.
- We evaluated our approach on eleven kernels from the TSVC benchmark suite [16], and obtained a geometric-mean performance improvement of  $4.61\times$  on an Intel Xeon Phi (KNL) machine relative to the optimized performance obtained by Intel’s ICC v17.0 product compiler.
- We also compared our approach with the two most closely related algorithms from past work, one by Calland et al. [5] that only performed SiVR transformations, and the other by Chu et al. [8] that proposed a (not necessarily optimal) heuristic to combine SiVR and SoVR transformations.

Relative to Calland et al’s approach, our approach delivered an overall geometric-mean performance improvement of  $1.08\times$  and  $1.14\times$  on the Intel KNL and Nvidia Volta platforms respectively, though our approach selected exactly the same (SiVR-only) transformations for six of the eleven benchmarks. Relative to Chu et al’s approach, our approach delivered an overall geometric-mean performance improvement of  $1.57\times$  and  $1.22\times$  on the Intel KNL and Nvidia Volta platforms respectively.

Original program having cycles	Applying SoVR( $s_2$ , $a[i+1]$ ) on the original program	Applying SiVR( $s_1$ , $a[i]$ ) on the original program
<pre> for i = 1 to N {   a[i] = b[i]+c[i]; //s1   a[i+1] = a[i-1]+2*a[i+1]; //s2 } </pre>	<pre> for i = 1 to N {   a[i] = b[i]+c[i]; //s1   float k = a[i+1]; //s21   a[i+1] = a[i-1]+2*k; //s2 } </pre>	<pre> float a_temp[N]; for i = 1 to N {   a_temp[i] = b[i]+c[i]; //s11   a[i] = a_temp[i]; //s1   a[i+1] = (i &gt; 1) ? \ //s2     (a_temp[i-1] : a[i-1])+2*a[i+1] } </pre>
Dependence graph of the original program 	Dependence graph after applying SoVR( $s_2$ , $a[i+1]$ ) on the original program 	Dependence graph after applying SiVR( $s_1$ , $a[i]$ ) on the original program 

**Table 1.** An example to illustrate SoVR and SiVR transformations.

## 2 Discussion on Variable Renaming Transformations

In this section, we discuss on two variable renaming transformations that are considered in this paper, and they are *Source variable renaming* (SoVR)<sup>3</sup> introduced by Kuck et al. in [15] and *Sink variable renaming* (SiVR) introduced by Chu et al. in [7]. Furthermore, these two renaming transformations were formalized by Calland et al. in [5], and referred SoVR and SiVR as T1 and T2 transformations respectively.

### 2.1 Source Variable Renaming (SoVR)

Source variable renaming transformation is introduced to handle anti-dependences in cycles of memory-based dependences, and the transformation is applied on a read access of a statement to reposition an outgoing anti-dependence edge from the read access [15]. Applying SoVR on a read access (say  $r$ ) of a statement introduces a new assignment statement that copies the value of  $r$  into a temporary variable (say  $k$ ), and then the original statement’s read access is replaced with  $k$ . Since the transformation is renaming source (read access) of an anti-dependence, we call this transformation as a source variable renaming transformation.

*Example.* Applying SoVR on the read access  $a[i+1]$  of the statement  $s_2$  in the original program (shown in Table 1) introduces a new assignment statement  $s_{21}$  copying the value of  $a[i+1]$  into a temporary variable  $k$ , and then the statement  $s_2$  refers to  $k$  in-place of  $a[i+1]$ . As a result, the source of the anti-dependence from the read access  $a[i+1]$  is repositioned to  $s_{21}$ . This reposition helps in breaking one of the cycles through  $s_2$ , i.e., the cycle involving a flow-dependence from  $a[i]$  of  $s_1$  to  $a[i-1]$  of  $s_2$ , and an anti-dependence from  $a[i+1]$  of  $s_2$  to  $a[i]$  of  $s_1$ .

*Usefulness.* Since SoVR transformation is applied on a read access of a statement, the transformation can modify only incoming flow- and outgoing anti-dependences related to that read access. Hence, applying a SoVR transformation on a statement is useful in breaking cycles if the statement has an incoming

<sup>3</sup>SoVR was also referred as node splitting by Kuck et al. in [15].

anti- or output-dependences and an outgoing anti-dependence [5]. Also, SoVR transformation can be useful if the statement’s incoming flow-dependence and outgoing anti-dependence are on different accesses.

*Space requirements & Additional memory traffic.* The temporary variable introduced as part of a SoVR transformation is private to a loop carrying an anti-dependence that we are interested in repositioning. Hence, SoVR requires an additional space equivalent to the length of vector registers (i.e., VLEN) of target hardware. Furthermore, the transformation additionally introduces only one scalar load and one scalar store per every iteration of the target loop.

## 2.2 Sink Variable Renaming (SiVR)

Sink variable renaming transformation is introduced to handle both anti- and output-dependences in cycles of memory-based dependences [7]. The transformation is applied on a write access of a statement to reposition an outgoing flow-dependence from the write access and also an outgoing anti-dependence from the statement. Applying SiVR on a write access (say  $w$ ) of a statement  $s$  introduces a new assignment statement that evaluates the right hand side of the statement into a temporary array (say  $\mathit{temp}$ ), and then any references to the value of  $w$  are replaced by accessing the  $\mathit{temp}$ . Since SiVR transformation is applied on a write access of a statement, the transformation can modify only incoming anti- or output-dependences related to that write access. As a result, applying SiVR transformation is useful in breaking cycles if the statement has either an incoming anti- or output-dependences and either an outgoing flow- or anti-dependences [5]. Since the transformation is renaming the sink (the write access) of an incoming anti- or output-dependence, this transformation is called as sink variable renaming transformation [7].

*Example.* Applying SiVR on the write access  $a[i]$  of the statement  $s1$  in the original program (shown in Table 1) introduces a new assignment statement  $s11$  that evaluates the rhs of  $s1$  into a temporary array  $a\_temp$ , and then the transformation replaces the references to  $a[i]$  (such as  $a[i-1]$ ) with the  $a\_temp$ . As a result, the source of the flow-dependence from the write access  $a[i]$  is repositioned to  $s11$ . This repositioning helps in breaking all of the cycles present in the original program including the one that is not broken by the previous SoVR transformation, i.e., the cycle involving a flow-dependence from  $a[i]$  of  $s1$  to  $a[i-1]$  of  $s2$ , and an output-dependence from  $a[i+1]$  of  $s2$  to  $a[i]$  of  $s1$ .

*Space requirements* The temporary array introduced as part of a SiVR transformation is not private to a loop unlike SoVR transformation, because references to the newly allocated storage can be across iterations. Hence, SiVR requires an additional space equivalent to the number of iterations of a loop. However, the additional storage can be reduced by strip mining the loop, and vectorizing only the strip [21]; whose space requirement is now proportional to the strip size, and the strip can be as minimal as vector length.

*Additional memory traffic.* SiVR transformation introduces pointer-based loads and stores, unlike the SoVR transformation which introduces only scalar loads and stores. The new assignment statement as part of a SiVR transformation introduces one additional pointer-based store and one pointer-based load

per one iteration of the loop. Along with a new assignment statement, each reference to the newly allocated storage introduces one additional pointer-based load, leading to overall  $(1+\#\text{references})$  of pointer-based loads per one iteration of the loop. In this work, we focus on applying renaming transformations for vectorizing only inner-most loops, and this focus helps in conservatively counting the references to the newly allocated storage by traversing the loop body and ignoring conditionals.

### 2.3 Synergy between SoVR and SiVR

In general, SoVR transformation is neater in code generation and performs more efficiently than SiVR since the SoVR transformation introduces scalar loads and stores. But, SoVR transformation has limited applicability (i.e., handling only anti-dependences) in breaking cycles compared to SiVR, which has broader applicability through breaking output-dependences. Table 2 shows a comparison between SoVR and SiVR transformations related to space requirements and additional stores and loads.

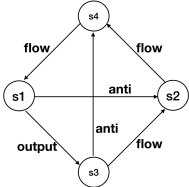
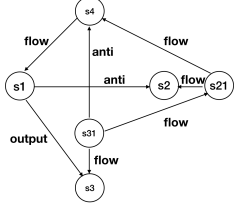
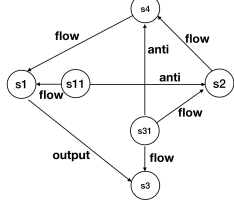
		SoVR	SiVR
Storage	#Additional space	Vector length	Loop length
Additional loads & stores	#scalar loads	1	0*
	#scalar stores	1	0*
	#pointer-based loads	0	$1+\#\text{references}$
	#pointer-based stores	0	1

**Table 2.** A comparison between SoVR and SiVR transformations related to the space requirements and additional stores, loads introduced by these transformations in one iteration of the target loop. \* – Additional scalar loads/stores for SiVR transformation may go negative in case of renaming scalars.

## 3 Motivating Example

To motivate the need of a unified framework that synergistically integrates multiple variable renaming transformations, we consider a running example (shown in Figure 1) from [5] whose dependence graph consists of three cycles (i.e.,  $s1-s3-s2-s4-s1$ ,  $s1-s3-s4-s1$ , and  $s1-s2-s4-s1$ ) which prohibit vectorization. Past work by Calland et al. [5] uses only SiVR transformations to eliminate all of the above three cycles by applying SiVR transformations on the statements **s2** and **s3**. But, these transformations require an additional space close to 2 times the number of iterations of the `loop-i`, i.e., a total of  $(2 \times T)$ , and also introduce additional 2 pointer-based stores and 4 pointer-based loads per one iteration of the loop.

However, instead of applying SiVR transformation on the statement **s2** to break the cycle (c3), SoVR transformation can be applied on the **s1** to break the same cycle (c3). This results in lesser additional space  $(T + VLEN)$ , and also introduces lesser additional 1 pointer-based store and 2 pointer-based loads per one iteration of the loop. Our approach identifies such optimal transformations from a set of valid SoVR and SiVR transformations by formalizing the “cycle-breaking” problem as a minimum weighted set cover optimization problem with a goal of reducing overhead arising from additional loads and stores introduced

Original program from [5] having cycles	Past approach by Calland et al. [5] on the original program	Our approach on the original program
<pre>float a[N], b[N], c[N]; for i = 4 to T {   a[i+5]=c[i-3]+b[2i+2]; //s1   b[2i] = a[i-1] + 1; //s2   a[i] = c[i+5] - 1; //s3   c[i] = b[2i-4]; //s4 }</pre>	<pre>float a[N], b[N], c[N]; float a_temp[T], b_temp[T]; for i = 4 to T {   a[i+5] = c[i-3]+b[2i+2]; //s1   b_temp[i] = (i &gt;= 5) ? \     a_temp[i-1]:a[i-1]+1; //s21   b[2i] = b_temp[i]; //s2   a_temp[i]=c[i+5]-1; //s31   a[i] = a_temp[i]; //s3   c[i] = (i &gt;= 6) ? \     b_temp[i-2]:b[2i-4]; //s4 }</pre>	<pre>float a[N], b[N], c[N]; float a_temp[T]; for i = 4 to T {   float k = b[2i+2]; //s11   a[i+5] = c[i-3] + k; //s1   b[2i] = (i &gt;= 5) ? \     a_temp[i-1]:a[i-1]+1; //s2   a_temp[i]=c[i+5]-1; //s31   a[i] = a_temp[i]; //s3   c[i] = b[2i-4]; //s4 }</pre>
<p>Dependence graph of the original program</p> 	<p>Dependence graph after applying the past approach by Calland et al. [5] on the original program SiVR(s2, b[2i]), SiVR(s3, a[i])</p> 	<p>Dependence graph after applying our approach on the original program SoVR(s1, b[2i+2]), SiVR(s3, a[i])</p> 

**Fig. 1.** A running example from [5] whose dependence graph consists of three cycles  $c1/c2/c3$ :  $s1-s3-s2-s4-s1/s1-s3-s4-s1/s1-s2-s4-s1$  which prohibit vectorization. The table also lists dependence graphs and transformed codes after applying past approach [5] and our integrated approach on the original program.

by these transformations. The speedup’s after applying our approach over the original program is  $5.06\times$  and  $4.02\times$  compared to the original program and the transformed program after applying the Calland et al. approach [5] respectively on the Intel Knights Landing processor (More details about the architectures and compiler options can be found in Table 4).

## 4 Our Unified Approach to Variable Renaming

In this section, we introduce our approach that synergistically integrates SoVR and SiVR transformations into a unified formulation to break cycles of dependences involving memory-based dependences, and the approach is implemented in a tool called *PolySIMD*.

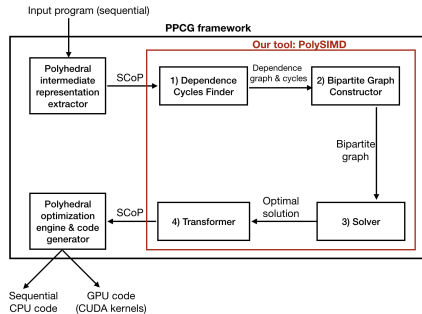
The overall approach is summarized in Figure 2, which is implemented as an extension to the PPCG framework [20] (a state-of-the-art optimization framework for loop transformations), and consists of the following components: 1) Dependence cycles finder (Extracting flow-, anti-, and output-dependences on a target loop, then constructing a dependence graph, and then finding cycles in the graph using the *Johnson’s algorithm* [12]), 2) Bipartite graph constructor (Building a bipartite mapping from a union over useful SoVR and SiVR transformations to the breakable cycles, in such a way that there is an edge between them if the transformation can break the cycle), 3) Solver (Reducing the problem

of breaking cycles as a weighted set covering optimization problem and finding an optimal solution using the ILP solver of ISL framework [19]), 4) Transformer (Applying SoVR and SiVR transformation from the optimal solution to break cycles).

#### 4.1 Dependence Cycles Finder

This component takes the polyhedral intermediate representation (also referred to as SCoP) extracted from a target loop as an input. Then, the loop-carried and loop-independent flow-, anti-, and output-dependences (including both data and control dependences) of the target loop are computed using the PPCG dependence analyzer. Afterwards, these dependences are represented as a directed graph, where a node denotes a statement, and an edge denotes a dependence between two statements. Also, each edge of a directed graph is annotated with a dependence type: flow-, anti-, or output-. Now, *PolySIMD* computes all strongly connected components (SCC's) of the directed graph using the *Tarjan's algorithm* [11]. Then, all elementary cycles<sup>4</sup> for every SCC of the dependence graph are identified using the *Johnson's algorithm* [12], an efficient algorithm to enumerate all elementary cycles of a directed graph. The worst case time complexity of the algorithm is  $O((n+e)(c+1))$  where  $n$  is the number of vertices,  $e$  is the number of edges and  $c$  is the number of distinct elementary cycles in a directed graph. For example, applying *Johnson's algorithm* on a dependence graph of the running example (shown in Figure 1 and has only one SCC) results in three elementary cycles  $c1/c2/c3$ :  $s1-s3-s2-s4-s1/s1-s3-s4-s1/s1-s2-s4-s1$  on the `loop-i`.

Note that SoVR and SiVR transformation cannot break a cycle if the cycle is either a pure flow- or pure output-dependence cycle [5]. Since our approach considers only SoVR and SiVR into the formulation, if *PolySIMD* encounters any dependence cycle involving pure flow- or pure output-dependences in a SCC, then the tool ignores the SCC and continues with the rest of SCC's. If each SCC have either a pure flow- or pure output-dependence cycle, then *PolySIMD* will skip rest of steps in our approach, otherwise the tool continues with next steps. Since the three cycles  $c1$ ,  $c2$ , and  $c3$  of the running example are neither pure-flow nor pure-output dependence cycles, our approach proceeds to the next step.



**Fig. 2.** Workflow of *PolySIMD* implemented as an extension to the PPCG [20].

<sup>4</sup>An elementary cycle of a directed graph is a path in which no vertex appears twice except the first and last vertices. Since elementary cycles form a basis for enumerating all cycles in a directed graph, breaking all of them results in an acyclic graph.



## 4.2 Bipartite Graph Constructor

This component constructs a bipartite graph between a union of useful SoVR and SiVR transformations (see Section 2 for usefulness criteria) and breakable cycles of the dependence graph such that there is an edge between them if applying the transformation can break the cycle. As from the usefulness criteria, Table 3 shows a tabular version of the bipartite graph constructed for the running example.

Transformations (T)	Cycles (C)
t1 = SoVR(s1, b[2i+2])	c3
t2 = SiVR(s2, b[2i])	c3
t3 = SiVR(s3, a[i])	c1, c2
t4 = SoVR(s3, c[i+5])	c2
t5 = SiVR(s4, c[i])	c2

**Table 3.** Bipartite graph constructed on the dependence graph of the original program in Figure 1.

## 4.3 Solver

After constructing the bipartite graph, the problem of finding an optimal set of transformations for cycle breaking is reduced to a minimum weighted set cover optimization problem  $(C, T, W)$  where  $C$  refers to a collection of cycles,  $T$  refers to a set of useful SoVR and SiVR transformations, and  $W$  refers to a set of weights for each transformation. The goal of the optimization problem is to identify the minimum weighted sub-collection of  $T$  whose union covers all cycles in  $C$ , and the optimization problem is known to be NP-hard. Hence, we formulate the minimum weighted set covering problem as the following integer linear programming (ILP) in our tool-chain.

### Variables:

- A variable  $t_i$  for each transformation of  $T$

$$t_i \in \{0, 1\}, \forall t_i \in T$$

where  $t_i = 1$  indicates that the transformation  $t_i$  should be applied on the original program, otherwise it should be ignored.

- A weight parameter  $w_i$  for each transformation  $t_i$  to indicate an additional execution overhead (ignoring cache effects), and is measured using the additional loads and stores introduced by the transformation per one iteration of the target loop (See Table 2 for more details).
- A *latencyratio* parameter to indicate the ratio of access times of main memory to registers, and this parameter is used in converting weight parameters of SiVR transformations (introduced pointer-based loads/stores) into same units as of weight parameters of SoVR transformations (introduces scalar-based loads/stores).

**Acyclicity constraint:** The acyclic constraint on the dependence graph is modeled into a condition that each cycle of  $C$  should be covered by at-least one transformation of  $T$ .

$$\forall c_j \text{ in } C, \left( \sum_{\substack{\forall t_i \text{ in } T \\ \text{such that } t_i \text{ can break } c_j}} t_i \right) \geq 1$$

**Objective function:** Our approach targets at minimizing additional overhead introduced by the optimal set of transformations.

$$\text{Minimize } \left( \sum_{\forall t_i \text{ in } T} w_i \times t_i \right)$$

The ILP formulation for the example is as follows (Assuming *latencyratio* as 50).

$$T = \{t1, t2, t3, t4, t5\}, \quad C = \{c1, c2, c3\}, \quad t_i \in \{0, 1\}, \quad \forall t_i \in T,$$

$$w1 = w4 = 2, \quad w2 = w3 = w5 = 50 \times 3 = 150,$$

$$t3 \geq 1, \quad t3 + t4 + t5 \geq 1, \quad t1 + t2 \geq 1,$$

$$\text{Minimize } \left( 2 \times (t1 + t4) + 150 \times (t2 + t3 + t5) \right)$$

The optimal solution obtained for the above formulation is ( $t1=1, t2=0, t3=1, t4=0, \text{ and } t5=0$ ), i.e., applying SoVR on `s2` and SiVR on `s3` can break all cycles present in the running example with minimal additional overhead introduced. Note that the above solution is different to the solution ( $t2 = 1, t3 = 1$ ) from the Calland et al’s approach in [5] since our approach considers both SoVR and SiVR transformations into the formulation, unlike the Calland et al’s approach which includes only SiVR transformations.

*Heuristics.* There can be simple heuristics such as applying SoVR transformation in the beginning to break as many cycles it can and followed by applying SiVR transformation to break rest of cycles, which can lead to the similar performance improvements compared to our approach. The solution from such heuristics may include redundant SoVR transformations, which can be observed on the running example. Applying transformation `t4` (SoVR) on the running example (ahead of SiVR transformations) to break the cycle `c2` is redundant because the transformation `t3` (SiVR) will eventually break the cycle `c2` and also can break cycle `c1` that cannot be broken by any SoVR transformation.

There can exist other heuristics or greedy algorithms to the minimum weighted set cover optimization problem. But, we believe that an ILP formulation formalizes the optimization problem without being tied to specific heuristics, which in turn reduces performance anomalies that can occur in optimization heuristics; Also, the compile-times for the results in this experimental evaluation are less than half a second (see Table 5 for more details). We also believe that our framework can be easily extended to include other heuristics or greedy algorithms to the optimization problem.

#### 4.4 Transformer

This component applies the optimal set of transformations obtained from the solver onto the intermediate polyhedral representation of the target loop. It is

also mentioned in [5] that the order of applying SoVR and SiVR transformations doesn’t have any effect on the final program. Hence, *PolySIMD* first applies SoVR transformations from the optimal solution, and then followed by SiVR transformations from rest of the optimal solution. The generation of new assignment statements, modifying schedules of statements, and updating the references as part of the code transformations are implemented using the dependence analyzer and schedule trees of the PPCG framework.

After applying all transformations from the optimal solution, *PolySIMD* feeds the transformed intermediate polyhedral representation to the PPCG optimization engine to perform statement reordering based on the topological sorting of the transformed dependence graph. Note that all of the benchmarks in the experimental evaluation required statement reordering transformation to be performed without which the Intel’s ICC v17.0 product compiler couldn’t vectorize. This demonstrates the necessity of coupling storage optimizations with the loop optimization framework. Finally, *PolySIMD* leverages code generation capabilities of the PPCG framework to generate transformed sequential CPU code that can be input into a vectorizing compiler like ICC or generates GPU code (CUDA kernels) that can be processed by a GPU compiler like NVCC.

#### 4.5 Bounding Additional Space

We believe that one of the major key limitations in the unavailability of variable renaming techniques (especially on arrays) in modern compilers is due to the lack of support for bounding the additional space required to break memory-based dependences. Hence, we provide a clause (i.e., *spacelimit*) to the directive “`#pragma vectorize`” that can help programmers to limit the additional space to enable enhanced vectorization of inner-most loops, and the *spacelimit* is expressed in multiples of vector registers length. The clause *spacelimit* essentially helps our approach to compute strip size that can be vectorized, and the formula to compute the strip size (in multiples of vector length) is as follows.

$$\text{strip size} = \left\lfloor \frac{\text{spacelimit} \times VLEN - |T_{SoVR}| \times VLEN}{|T_{SiVR}| \times VLEN} \right\rfloor = \left\lfloor \frac{\text{spacelimit} - |T_{SoVR}|}{|T_{SiVR}|} \right\rfloor$$

where  $|T_{SoVR}|$  and  $|T_{SiVR}|$  refer to number of SoVR and SiVR transformations in the optimal solution respectively. If the strip size value is non-positive for a given *spacelimit*, then our approach ignores applying renaming transformations. Otherwise, our approach does strip mining of the target loop before applying any of the renaming transformations from the optimal solution.

## 5 Performance Evaluation

In this section, we present an evaluation of our *PolySIMD* tool relative to Intel’s ICC v17.0 product compiler and to the two algorithms presented in past work [5, 8] for performing SiVR and SoVR transformations to break cycles of a dependence graph. We begin with an overview of the experimental setup and the benchmark suite used in our evaluation, and then present experimental results for the three different comparisons.

## 5.1 Experimental Platforms

Our evaluation uses the following two SIMD architectures. 1) A many-core Intel Xeon Phi Knights Landing (KNL) processor with two 512-bit vector processing units (VPU) per core. Thus, each 512-bit VPU can perform SIMD operations on 16 single-precision floating point values, i.e., the VPU has an effective vector length of 16 (for 32-bit operands). Since we are evaluating vectorization for single-threaded benchmarks, we only use one core of the KNL processor in our evaluation, though our approach can be applied to multithreaded applications as well. 2) An Nvidia Volta accelerator (Tesla V100) with 80 symmetric multiprocessors (SMs), each of which can multiplex one or more thread blocks. A thread block can contain a maximum of 1024 threads, which are decomposed into 32-thread warps for execution on the SM. Thus, each SM can be viewed as being analogous to a VPU with an effective vector length of 32 (for 32-bit operands). For consistency with our KNL results, we only generate one block of 1024 threads per benchmark, thereby only using one SM in the GPU. However, our approach can be applied to multi-SM executions as well. Table 4 lists the system specifications and the compiler options used in our evaluations. The comparison with ICC could only be performed on KNL, since ICC does not generate code for Nvidia GPUs. The comparison with the two algorithms from past work [5, 8] were performed on both platforms.

	Intel Xeon Phi	Nvidia Volta
Microarch	Knights Landing	Tesla V100
SIMD lanes	16 SP per VPU (2 VPU's per core)	32 SP per SM
Compiler	Intel ICC v17.0	Nvidia NVCC v9.1
Compiler flags	-O3 -xmic-avx512	-O3 -arch=sm_70 -ccbin=icc

**Table 4.** Summary of SIMD architectures and compiler flags used in our experiments. SP refers to Single Precision floating point operands, VPU refers to a KNL Vector Processing Unit, and SM refers to a GPU Streaming Multiprocessor.

## 5.2 Benchmarks

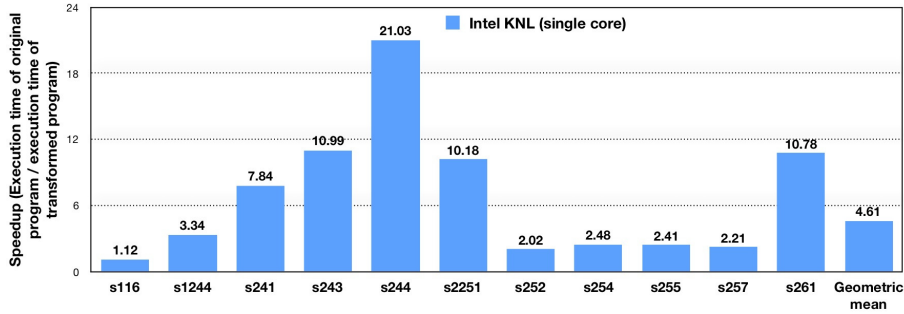
We use the Test Suite for Vectorizing Compilers (TSVC) benchmark suite in our evaluation, originally developed in FORTRAN to assess the vectorization capabilities of compilers [4]. Later, the benchmark suite was translated into C with additional benchmarks to address limitations in the original suite [16], so we used this C version for our evaluations. A detailed study of these benchmarks, along with the vectorization capabilities of multiple compilers can be found in [16, 9]. Since our goal is to evaluate the effectiveness of renaming variables on breaking dependence cycles that inhibit vectorization, we restrict our attention to TSVC benchmarks that contain multi-statement dependence cycles containing at least one anti/output dependence and that cannot be broken by scalar privatization. Further, since *PolySIMD* is based on a polyhedral optimization framework, we further restricted our attention to the subset of these benchmarks that do not contain non-affine expressions that prevent polyhedral analysis<sup>5</sup>. This selection

<sup>5</sup>This constraint arises from the implementation of our algorithm in *PolySIMD*; our algorithm can be applied in a non-polyhedral compiler setting as well.

resulted in 11 benchmarks from the TSVC suite that will be the focus of our evaluation, and are summarized in Table 5.

Benchmark	#Stmts	#Deps	#Elementary cycles	Our ILP Solution		Compilation time (sec)	
				#SoVR's	#SiVR's	PolySIMD	Total
s116	5	5	1	1	0	0.08	0.10
s1244	2	2	1	1	0	0.01	0.02
s241	2	3	1	1	0	0.01	0.03
s243	3	6	2	1	0	0.02	0.04
s244	3	4	1	1	0	0.02	0.03
s2251	3	4	1	0	1	0.02	0.03
s252	3	5	2	0	2	0.02	0.04
s254	2	2	1	0	1	0.01	0.02
s255	3	6	3	0	2	0.02	0.04
s257	2	3	1	0	1	0.02	0.04
s261	4	9	3	0	2	0.02	0.04

**Table 5.** Summary of the 11 benchmarks from the TSVC suite used in our evaluation, including the number of statements, number of dependences, and number of elementary cycles per benchmark (excluding self-loop cycles). The benchmarks were executed using  $N = 2^{25}$  and  $T = 200$  as input parameters. Number of SiVR and SoVR transformations performed by *PolySIMD* for the 11 benchmarks, and also the overall compilation times required. Coincidentally, none of these benchmarks triggered a case in which both SiVR and SoVR transformations had to be performed.



**Fig. 3.** Speedups using *PolySIMD* on the eleven benchmarks from the TSVC suite, compiled using the Intel’s ICC v17.0 product compiler and running on a single core of Intel Knights Landing processor.

### 5.3 Comparison with ICC

As discussed in Figure 2, *PolySIMD* takes a sequential program as input, and generates sequential code as output with selected variable renamings and statement reorderings that enable enhanced vectorization. Figure 3 shows the speedups obtained by using *PolySIMD* as a preprocessor to Intel’s ICC v17.0 product compiler on the KNL platform. The speedup represents the ratio of the execution time of the original program compiled with ICC to the execution time of the transformed program compiled with ICC, using the compiler options in

Table 4 in both cases. As can be seen in Figure 3, the use of *PolySIMD* as a preprocessor results in significant performance improvements for the 11 kernels. The transformations performed by *PolySIMD* are summarized in Table 5; the fact that no benchmark required both SiVR and SoVR transformations is a pure coincidence. We now discuss the two groups of benchmarks for which *PolySIMD* applied the SoVR and SiVR transformations respectively.

**Source Variable Renaming (SoVR):** The benchmarks `s116`, `s1244`, `s241`, `s243`, `s244` in the first five entries of Table 5 contain multi-statement recurrences involving outgoing anti-dependences. Hence, *PolySIMD* applied the SoVR transformation on these benchmarks to reposition these outgoing anti-dependence edges to break the cycles, as dictated by the column titled *SoVR* under *ILP solution* of Table 5. There are a few interesting observations that can be made from the results in Table 5 for these five benchmarks: 1) The SoVR transformation enabled vectorization for all five benchmarks (as confirmed by the compiler log output), and resulted in speedups varying from  $1.12\times$  to  $21.02\times$  on Intel KNL relative to the original program using the Intel’s ICC v17.0 product compiler. 2) The `s1244` benchmark involves dead-write statements (i.e., there are no reads of a write before another statement writing to the same location) whose removal eliminate dependence cycles. Currently, *PolySIMD* doesn’t check for dead-write statements unlike the Intel compiler (with O3 optimization flag enabled) which remove the dead writes to enable the vectorization. As a result, there is a lower speedup with our approach compared to the Intel compiler. 3) The reason for less speedup in case of the `s116` benchmark is the generation of non-unit (unaligned) strided loads and stores leading to inefficient vectorization (as confirmed by the compiler log output describing the estimated potential speedup as  $1.36\times$ ). 4) All these five benchmarks required statement reordering to be performed after the SoVR transformations, without which the Intel’s compiler wasn’t able to vectorize. This indicates the necessity of loop transformations framework to output the final code that can be vectorizable by the existing compilers.

**Sink Variable Renaming (SiVR):** The column titled *SiVR* under *ILP solution* indicates that the SiVR transformation should be performed on the remaining benchmarks (`s2251`, `s252`, `s254`, `s255`, `s257`, `s261`) in Table 5. These benchmarks have dependence cycles involving anti- and output-dependences, and hence our approach chose only the SiVR transformations to break these dependence cycles. As with the earlier five benchmarks, there are a few interesting observations that can be made from the results in Table 5 for these later three benchmarks: 1) The SiVR transformation enabled vectorization for all the remaining six benchmarks, and resulted in speedups varying from  $2.02\times$  to  $10.77\times$  on the Intel KNL platform relative to the original program. The compiler log output shows that vectorization was indeed performed in all cases. 2) The benchmarks `s252`, `s254`, `s255`, `s257`<sup>6</sup> have loop-carried flow-dependence and loop-independent anti-dependences on scalars, and resolving these dependences on scalars using our approach introduced higher overhead from temporary arrays

---

<sup>6</sup>Also, most of accesses in these benchmarks are dominated with scalars.

pointer-based loads and stores. As a result, the performance improvements in these benchmarks are relatively low. 3) As seen with earlier five benchmarks benefited by the SoVR transformation along with the statement reordering, these six benchmarks also required statement reordering to be performed after the SiVR transformations, without which the Intel’s compiler wasn’t able to vectorize.

#### 5.4 Comparison with Calland et al’s approach

The heuristics proposed by Calland et al.[5] aim to find the minimum number of SiVR transformations to break all dependence cycles involving memory-based dependences. As a result, the heuristics choose only SiVR transformations for vectorizing all the eleven benchmarks. However, our approach chooses to perform SoVR transformations on five of the eleven benchmarks (**s116**, **s1244**, **s241**, **s243**, **s244**), since SoVR incurs less overhead than SiVR. Hence, we observe speedups (shown in Table 6) with our approach relative to Calland et al, varying from  $1.07\times$  to  $1.24\times$  on the Intel KNL platform and  $1.12\times$  to  $1.57\times$  on the NVIDIA Volta. For the remaining six benchmarks, our approach chose exactly the same set of SiVR transformations as did their approach, and hence there is no performance improvement in these cases. The overall geometric-mean speedups on all of the eleven benchmarks are  $1.08\times$  and  $1.14\times$  relative to their approach on the KNL and Volta platforms.

Bench -mark	Intel KNL		NVIDIA Volta	
	Calland et al. approach	Chu et al. approach	Calland et al. approach	Chu et al. approach
<b>s116</b>	1.20x	1.03x	1.29x	1.27x
<b>s1244</b>	1.10x	4.03x	1.57x	1.51x
<b>s241</b>	1.07x	1.49x	1.31x	1.70x
<b>s243</b>	1.27x	1.59x	1.47x	1.61x
<b>s244</b>	1.24x	1.22x	1.12x	1.32x
<b>s257</b>	1.00x	9.74x	1.00x	1.08x
<b>s261</b>	1.00x	1.20x	1.00x	1.19x

**Table 6.** Speedups on the Intel KNL processor and NVIDIA Volta accelerator using *PolySIMD* on seven benchmarks from the eleven benchmarks relative to past approaches, i.e., Calland et al.[5] and Chu et al.[8]. We excluded the remaining four benchmarks from the table since our results were similar to both of the past works.

#### 5.5 Comparison with Chu et al’s approach

Chu et al. proposed an algorithm for resolving general multistatement recurrences which considers both SoVR and SiVR transformation[8]. The solution obtained by their algorithm depends on a traversal of the dependence graph, and may not be optimal in general. Further, their algorithm may include redundant SiVR transformations, which were observed when applying their algorithm to benchmarks **s241**, **s243**, **s257** and **261**, leading to lower performance compared to our approach. We observed performance improvements on these benchmarks with our approach (relative to Chu et al), varying from  $1.20\times$  to  $9.74\times$  on KNL and  $1.08\times$  to  $1.70\times$  on Volta. For the remaining three benchmarks **s116**, **s1244** and **s244** in Table 6, our approach chose the same solution as their approach, but we still obtained better performance because *PolySIMD* generates

private scalars for SoVR transformations, unlike their algorithm which generates temporary arrays for the SoVR transformations. The generation of private scalars enabled our approach to achieve performance improvements speedups ranging from  $1.03\times$  to  $4.03\times$  on KNL and  $1.27\times$  to  $1.51\times$  on Volta. The overall geometric-mean speedups on all of the eleven benchmarks were  $1.57\times$  and  $1.22\times$  on the KNL and Volta platforms.

## 6 Related Work

Since there exists an extensive body of research literature in handling memory-based dependences, we focus on past contributions that are closely related to variable expansion [10], variable renaming including SoVR [15, 5], SiVR [8, 7, 6, 5] and Array SSA [14, 17].

*Comparison with past approaches involving SoVR and/or SiVR transformations.* Calland et al. [5] formally defined both SoVR and SiVR transformations, and also explained the impact of these transformations on a dependence graph. Also, Calland et al. proved that the problem of finding the minimum number of statements to be transformed—to break artificial dependence paths involving anti- or output-dependences—is NP-complete, and proposed some heuristics. However, the implementation and impact of these techniques on the performance of representative benchmarks were not mentioned. But, *PolySIMD* utilizes both SoVR and SiVR in a complementary manner to coordinate each other, and is built on a polyhedral framework (PPCG), and leveraged it for statement reordering to enable vectorization. Also, we did not find a framework publicly available from the past approaches. Chu et al. work in [8, 7] discussed dependence-breaking strategies in the context of recurrence relations, and developed an algorithm for the resolution of general multi-statement recurrences using the proposed strategies. But, the proposed algorithm for the resolution of cycles is not optimal and may generate solutions having redundant SoVR transformations.

*Other works on storage transformations.* Array SSA has been developed to convert a given program into a static single assignment form to enable automatic parallelization of loops involving memory-based dependences [14], and also to extend classical scalar optimizations to arrays [17]. However, applying renaming on writes of every statement of a loop body is significantly expensive in terms of additional space requirements, and may not be required for enabling vectorization. Other approaches such as variable expansion [10] can be used to break specific memory-based dependences. The variable expansion may be beneficial for applying onto scalars but expanding multi-dimensional arrays inside the inner-most loop for vectorization is expensive in terms of additional space. But, variable expansion can be useful in eliminating pure-output dependence cycles unlike with SoVR and SiVR, which is a part of our future work.

*Bounding additional space.* There has been lack of support for bounding the extra space required to break memory-based dependences in the past approaches [5, 8]. But, our approach provides a *spacelimit* clause that can help programmers to specify the maximum amount of extra storage that can be allocated. An alternative approach to enable parallelization or vectorization has always



been to convert the program to (dynamic) single assignment form, through array expansion, followed by affine scheduling [3] for vectorization, and then applying storage mapping optimization [2] (a generalized form of array contraction). Yet no such scheme can provide the guarantees that the affine transformations obtained on the fully expanded arrays will enable storage mapping optimization to restore a low-footprint implementation. Enforcing an a priori limit on memory usage would be even harder to achieve. Furthermore, no integrated system enabling vectorization through such a complex path of expansion and contraction has been available until now.

## 7 Conclusions & Future work

Despite the fact that compiler technologies for automatic vectorization have been under development for over four decades, there are still considerable gaps in the capabilities of modern compilers to perform automatic vectorization for SIMD units. This paper focuses on advancing the state of the art with respect to handling *memory-based anti* (write-after-read) or *output* (write-after-write) dependences in vectorizing compilers. In this work, we integrate both Source Variable Renaming (SoVR) and Sink Variable Renaming (SiVR) transformations into a unified formulation, and formalize the “cycle-breaking” problem as a minimum weighted set cover optimization problem. Our approach also can ensure that the additional storage introduced by our transformations remains within the user-provided bounds.

We implemented our approach in PPCG, a state-of-the-art optimization framework for loop transformations, and evaluated it on eleven kernels from the TSVC benchmark suite. Our experimental results show a geometric mean performance improvement of  $4.61\times$  on an Intel Xeon Phi (KNL) machine relative to the optimized performance obtained by Intel’s ICC v17.0 product compiler. Further, our results demonstrate a geometric mean performance improvement of  $1.08\times$  and  $1.14\times$  on the Intel Xeon Phi (KNL) and Nvidia Tesla V100 (Volta) platforms relative to past work that only performs the SiVR transformation [5], and of  $1.57\times$  and  $1.22\times$  on both platforms relative to past work on using both SiVR and SoVR transformations [8]. We believe that our techniques will be increasingly important in the current era of pervasive SIMD parallelism, since non-vectorized code will incur an increasing penalty in execution time on future hardware platforms.

As part of the future work, we plan to work on extending the unified formulation by including variable expansion [10] and forward propagation techniques [15] to break pure-output and to handle pure-flow dependence cycles respectively. Also, we plan to extend our approach and implementation to handle non-affine regions of codes, and also to support vectorization of outer loops as well. Furthermore, we plan to investigate into enabling loop transformations (such as tiling in case of cycles on tiles) using variable renaming transformations.

## References

1. Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A.F., Ketema, J., Absar, J., Haastregt, S.v., Kravets, A., Lokhmotov, A., David, R., Hajiyev, E.: PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In: Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT). pp. 138–149. PACT '15, IEEE Computer Society, Washington, DC, USA (2015). <https://doi.org/10.1109/PACT.2015.17>, <https://doi.org/10.1109/PACT.2015.17>
2. Bhaskaracharya, S.G., Bondhugula, U., Cohen, A.: SMO: An Integrated Approach to Intra-array and Inter-array Storage Optimization. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 526–538. POPL '16, ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2837614.2837636>, <http://doi.acm.org/10.1145/2837614.2837636>
3. Bondhugula, U., Acharya, A., Cohen, A.: The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. *ACM Trans. Program. Lang. Syst.* **38**(3), 12:1–12:32 (Apr 2016). <https://doi.org/10.1145/2896389>, <http://doi.acm.org/10.1145/2896389>
4. Callahan, D., Dongarra, J., Levine, D.: Vectorizing Compilers: A Test Suite and Results. In: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing. pp. 98–105. Supercomputing '88, IEEE Computer Society Press, Los Alamitos, CA, USA (1988), <http://dl.acm.org/citation.cfm?id=62972.62987>
5. Calland, P., Darté, A., Robert, Y., Vivien, F.: On the Removal of Anti- and Output-Dependences. *International Journal of Parallel Programming* **26**(2), 285–312 (1998). <https://doi.org/10.1023/A:1018790129478>, <https://doi.org/10.1023/A:1018790129478>
6. Chang, W.L., Chu, C.P., Ho, M.S.H.: Exploitation of parallelism to nested loops with dependence cycles. *Journal of Systems Architecture* **50**(12), 729 – 742 (2004). <https://doi.org/https://doi.org/10.1016/j.sysarc.2004.06.001>, <http://www.sciencedirect.com/science/article/pii/S1383762104000670>
7. Chu, C.P.: A Theoretical Approach Involving Recurrence Resolution, Dependence Cycle Statement Ordering and Subroutine Transformation for the Exploitation of Parallelism in Sequential Code. Ph.D. thesis, Louisiana State University, Baton Rouge, LA, USA (1992), uMI Order No. GAX92-07498
8. Chu, C.P., Carver, D.L.: An analysis of recurrence relations in Fortran Do-loops for vector processing. In: [1991] Proceedings. The Fifth International Parallel Processing Symposium. pp. 619–625 (Apr 1991). <https://doi.org/10.1109/IPPS.1991.153845>
9. Evans, G.C., Abraham, S., Kuhn, B., Padua, D.A.: Vector Seeker: A Tool for Finding Vector Potential. In: Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing. pp. 41–48. WPMVP '14, ACM, New York, NY, USA (2014). <https://doi.org/10.1145/2568058.2568069>, <http://doi.acm.org/10.1145/2568058.2568069>
10. Feautrier, P.: Array Expansion. In: Proceedings of the 2Nd International Conference on Supercomputing. pp. 429–441. ICS '88, ACM, New York, NY, USA (1988). <https://doi.org/10.1145/55364.55406>, <http://doi.acm.org/10.1145/55364.55406>
11. Hopcroft, J., Tarjan, R.: Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM* **16**(6),

- 372–378 (Jun 1973). <https://doi.org/10.1145/362248.362272>, <http://doi.acm.org/10.1145/362248.362272>
12. Johnson, D.B.: Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing* **4**(1), 77–84 (1975). <https://doi.org/10.1137/0204007>, <https://doi.org/10.1137/0204007>
  13. Kennedy, K., Allen, J.R.: *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
  14. Knobe, K., Sarkar, V.: Array SSA Form and Its Use in Parallelization. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 107–120. POPL '98, ACM, New York, NY, USA (1998). <https://doi.org/10.1145/268946.268956>, <http://doi.acm.org/10.1145/268946.268956>
  15. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M.: Dependence Graphs and Compiler Optimizations. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 207–218. POPL '81, ACM, New York, NY, USA (1981). <https://doi.org/10.1145/567532.567555>, <http://doi.acm.org/10.1145/567532.567555>
  16. Maleki, S., Gao, Y., Garzarán, M.J., Wong, T., Padua, D.A.: An Evaluation of Vectorizing Compilers. In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. pp. 372–382. PACT '11, IEEE Computer Society, Washington, DC, USA (2011). <https://doi.org/10.1109/PACT.2011.68>, <http://dx.doi.org/10.1109/PACT.2011.68>
  17. Rus, S., He, G., Alias, C., Rauchwerger, L.: Region Array SSA. In: *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. pp. 43–52. PACT '06, ACM, New York, NY, USA (2006). <https://doi.org/10.1145/1152154.1152165>, <http://doi.acm.org/10.1145/1152154.1152165>
  18. Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., Reid, A., Rico, A., Walker, P.: The ARM Scalable Vector Extension. *IEEE Micro* **37**(2), 26–39 (Mar 2017). <https://doi.org/10.1109/MM.2017.35>, <https://doi.org/10.1109/MM.2017.35>
  19. Verdoolaege, S.: Isl: An Integer Set Library for the Polyhedral Model. In: *Proceedings of the Third International Congress Conference on Mathematical Software*. pp. 299–302. ICMS'10, Springer-Verlag, Berlin, Heidelberg (2010), <http://dl.acm.org/citation.cfm?id=1888390.1888455>
  20. Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., Catthoor, F.: Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* **9**(4), 54:1–54:23 (Jan 2013). <https://doi.org/10.1145/2400682.2400713>, <http://doi.acm.org/10.1145/2400682.2400713>
  21. Weiss, M.: Strip Mining on SIMD Architectures. In: *Proceedings of the 5th International Conference on Supercomputing*. pp. 234–243. ICS '91, ACM, New York, NY, USA (1991). <https://doi.org/10.1145/109025.109083>, <http://doi.acm.org/10.1145/109025.109083>