A Unified Approach to Variable Renaming for Enhanced Vectorization

Prasanth Chatarasi[1],
Jun Shirako[1], Albert Cohen[2] and Vivek Sarkar[1]

[1] Georgia Institute of Technology, Atlanta, USA
[2] INRIA & DI ENS, Paris, France

Introduction

- Strong resurgence of interest in vector processing
  - Significant performance and energy efficiency benefits
  - Applicable to SIMT parallelization for GPU's
    - Wide vector SIMD units[1]

- Compilers will continue to play a central role in vectorization
  - Programmers still annotate loops for vectorization; leaving hard job of vectorization to compilers

- Very important to advance state of art vectorizing compiler technologies to full use SIMD units!

1: https://theincredibleholk.wordpress.com/2012/10/26/are-gpus-just-vector-processors/

Vectorization

- Vectorization of a loop is legal as long as no dependence cycles

- So, we focus on dependence cycles involving memory-based dependences (anti-, output-)
  - These dependences can be eliminated with additional storage
  - Past storage transformations
    - Variable expansion, Variable renaming, Array SSA, Privatization..

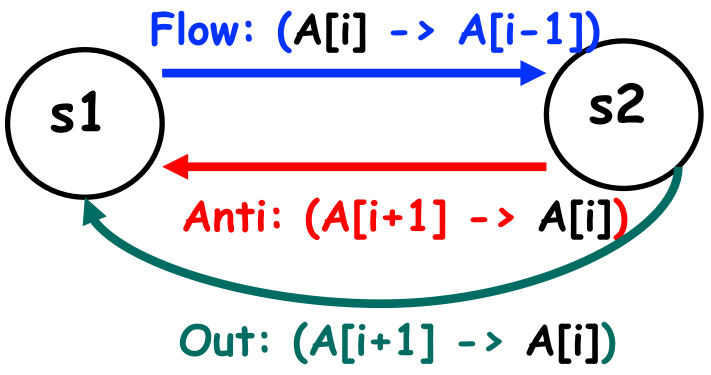
- Our contributions
  1. Unify multiple storage transformations, via a single formulation, to break cycles optimally
  2. Restricting additional space required by transformations

Agenda

- Introduction
- Background
  - Source variable renaming (SoVR)
  - Sink variable renaming (SiVR)
- Our approach
  - Unifying SoVR and SiVR transformations
- Evaluation
  - Intel KNL and Nvidia Volta
- Conclusions and future work

An example (Calland et al. IJPP'96)

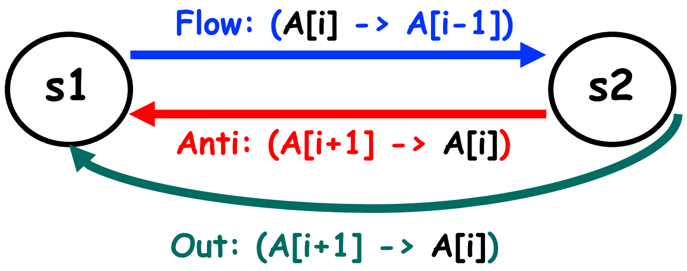
```
1: for(int i = 1; i < size; i++) {
2:       A[i] = B[i] + C[i]; //s1
3:       A[i+1] = A[i-1] + 2*A[i+1]; //s2
   }
```



Multi-statement dependence cycles (flow, anti) & (flow, output)

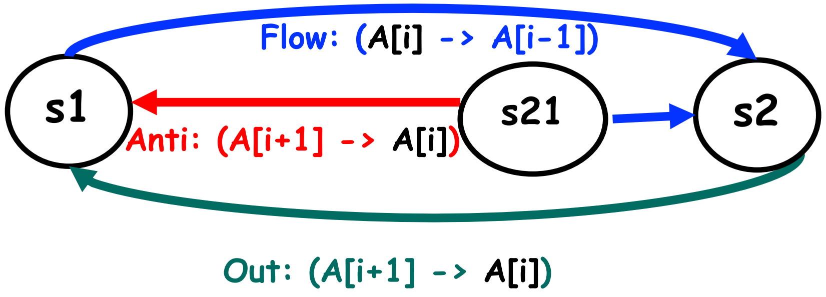
Handling (flow, anti) cycle

```
1: for(int i = 1; i < size; i++) {
2:        A[i] = B[i] + C[i]; //s1
3:        A[i+1] = A[i-1] + 2*A[i+1]; //s2
   }
```



Reading *source* of anti dependence, A[i+1], from a different location (k)

```
1: for(int i = 1; i < size; i++) {
2:       A[i] = B[i] + C[i]; //s1
3:       int k = A[i+1];  //s21
4:       A[i+1] = A[i-1] + 2*k;  //s2
    }
```



- The source of anti-dependence moved to s21

Source Variable Renaming (SoVR) (Kuck et al. POPL'81)

- Rename source of anti-dependence into a temporary private scalar -- Special case of "node splitting"

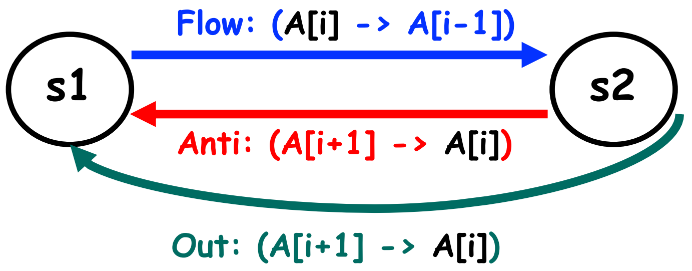
- Applied on a read access of a statement
  - Useful in breaking cycles if the statement has an outgoing anti-dependence and
    - An incoming flow-dependence incident on a different location
    - Or An incoming anti- or output-dependences

- The temporary variable introduced is private to the loop
  - It has only single *use*

- SoVR broke (flow, anti), but not (flow, output)?
  - Do we have a single storage transformation that can break both?

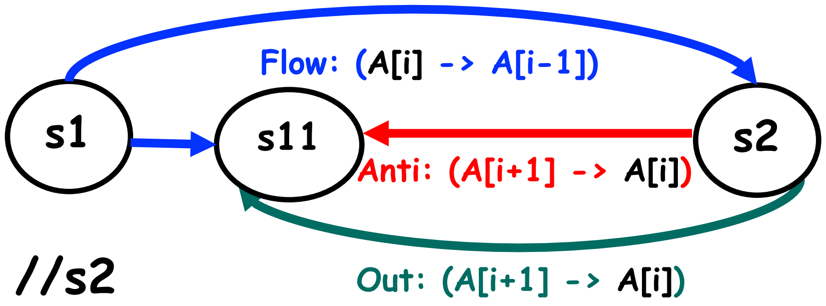
Handling (flow, anti), (flow, out) cycles

```
for(int i = 1; i < size; i++) {
        A[i] = B[i] + C[i]; //s1
        A[i+1] = A[i-1] + 2*A[i+1]; //s2
  }
```



Writing value of sink of anti-dependence, (B[i]+C[i]), into a different location temp[i], and updating A[i] *uses* with temp[i].

```
for(int i = 1; i < size; i++) {
        temp[i] = B[i] + C[i];//s1
        A[i] = temp[i]; //s11
        A[i+1] = temp[i-1] + 2*A[i+1]; //s2
  }
```



- Sinks of both anti-, output-dependence are moved to s11

Sink Variable Renaming (SiVR) (Chu PhD thesis’92)

- Rename the *sink* of anti-, output-dependences into a temporary array, and change the *sink* uses with new array

- Applied on a write access of a statement
  - Useful in breaking cycles if the statement has
    - Either an incoming anti- or output-dependences, and
    - Either an outgoing flow- or anti-dependence

- The SiVR transformation introduces a temporary array
  - *Uses* of a sink can be in other iterations

- SiVR broke both (flow, anti) and (flow, output)
  - But expensive due to pointer-based loads and stores

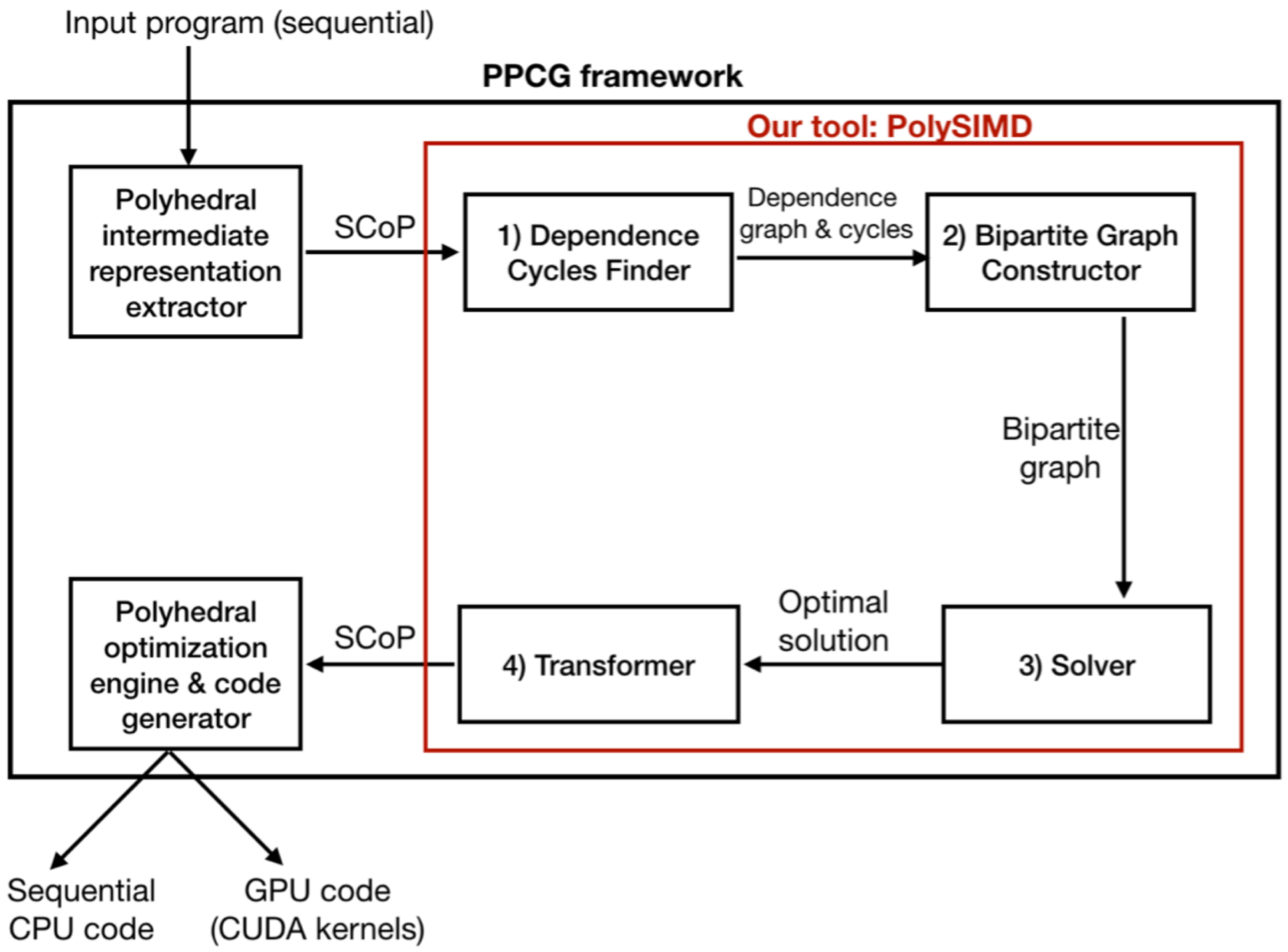
Problem statement & Related work

- SiVR can handle more cycles compared to SoVR; however SiVR is more expensive because it involves pointer-based loads and stores to array temporaries
  - How do you find out the best combination that can break all cycles optimally, i.e., with less execution overhead?

- Related work in breaking cycles
  1. Calland et al. IJPP'96
     - Considers only SiVR transformation to break cycles
  2. Chu et al. PhD thesis'92
     - Considers both SoVR and SiVR transformations, but not optimal

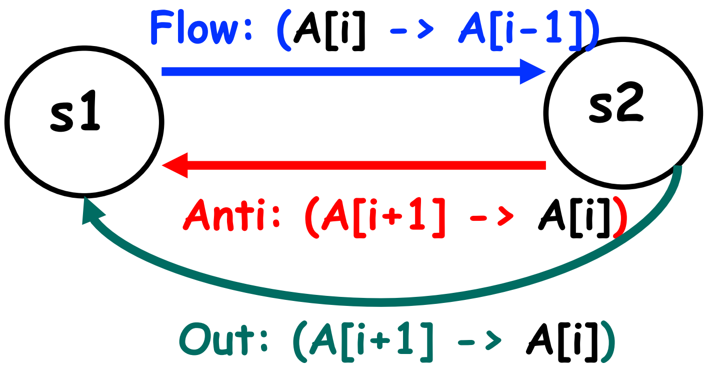
- Our approach
  - Optimal solution considering both SoVR and SiVR
  - Evaluation in the context of mature loop-optimization framework

Our workflow (PolySIMD)

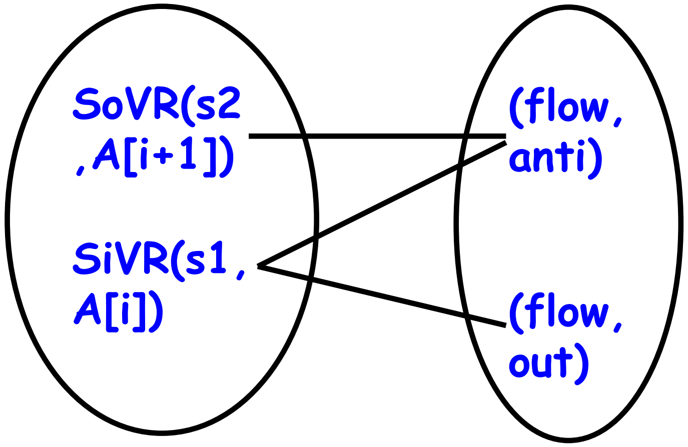

Input program (sequential)
PPCG framework
Our tool: PolySIMD
Polyhedral intermediate representation extractor
SCoP
1) Dependence Cycles Finder
Dependence graph & cycles
2) Bipartite Graph Constructor
Bipartite graph
3) Solver
Optimal solution
4) Transformer
SCoP
Polyhedral optimization engine & code generator
Sequential CPU code
GPU code (CUDA kernels)

Our approach (PolySIMD)

1. Dependence cycles finder
   - Johnson's algorithm[1] is used to enumerate all elementary cycles
   - Cycles: (S1, S2)
     - (flow, anti), (flow, out)



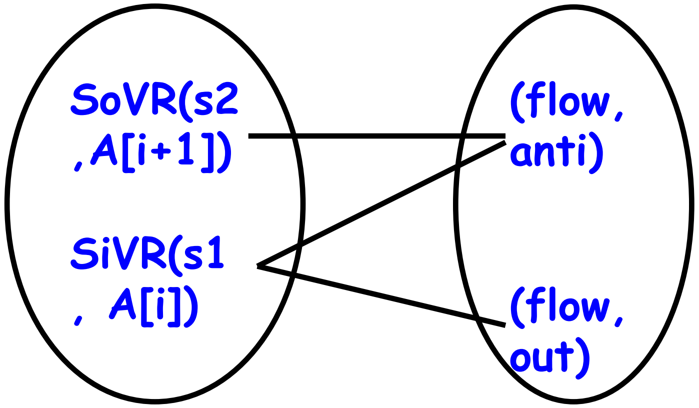
2. Bipartite graph constructor
   - Map between SoVR/SiVR transformations to the cycles
     - Global view for optimality
   - Weight for each transformation based on additional loads/store introduced



1. Complexity: O((n+e)(c+1)) where n, e, c are number of vertices, edges and distinct number of elementary cycles respectively.

Our approach (PolySIMD)

3. Solver
   - We reduce the problem to weighted set-cover optimization
   - We use standard ILP formulation to solve the above problem
     - Solution: SiVR(s1, A[i])



4. Transformer
   - Uses statement reordering transformation
   - Leverages PPCG code generation capabilities for CPU and GPU code

```
//CPU code (ignoring boundary conditions)
1: for(int i = 1; i < size; i++) {
2:      temp[i] = B[i] + C[i];//s1
3:      A[i+1] = temp[i-1] + 2*A[i+1]; //s2
4:      A[i] = temp[i]; //s11
}
```

```
//CUDA kernel (ignoring boundary conditions)
1: for(int i = threadId.x; i < size; i+=1024){
2:      temp[i] = B[i] + C[i];//s1
        __syncthreads();
3:      A[i+1] = temp[i-1] + 2*A[i+1]; //s2
        __syncthreads();
4:      A[i] = temp[i]; //s11
}
```

Overview of today’s talk

- Introduction
- Background
  - Source variable renaming (SoVR)
  - Sink variable renaming (SiVR)
- Our approach
  - Integration of SoVR and SiVR transformations
- <u>Evaluation</u>
  - <u>Intel KNL and NVIDIA Volta</u>
- Conclusions and future work

Evaluation (CPU)

- Experimental setup
  - On a single core of Intel KNL
    - 2 VPU's per core (512-bit vector wide)
      Potential for 32x speedup due to vectorization of 32-bit floats
    - Compiler & flags: Intel ICC v17.0 -O3 -xmic-avx512

- Evaluation on single threaded benchmarks
  - C version of the TSVC suite (Callahan et al., SC'88)
  - We restrict our attention to benchmarks with
    - Multi-statement dependence cycles containing at-least one anti-/output-dependence, which are breakable by SoVR/ SiVR
    - Analyzable affine programs
  - 11 TSVC benchmarks satisfy the above criteria

Summary of benchmarks

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

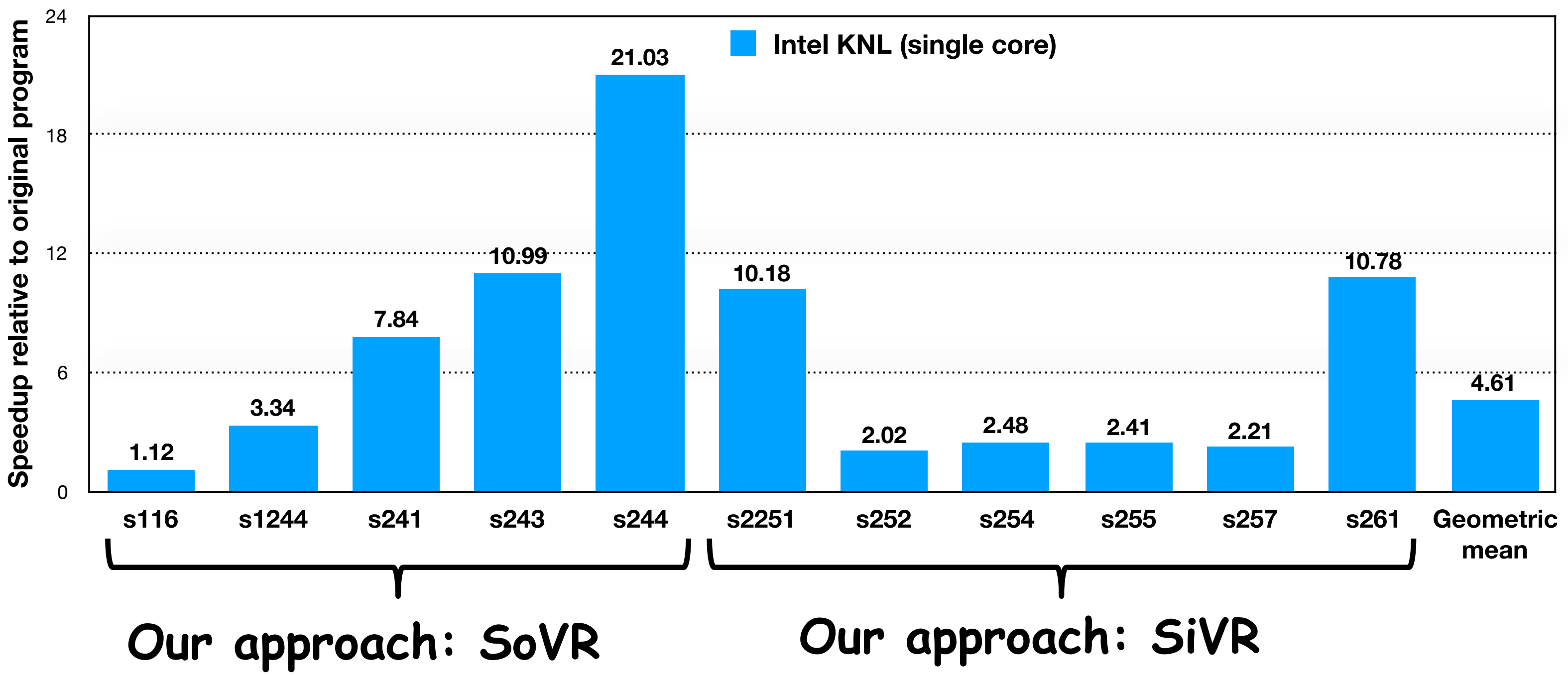
- Coincidentally, none of these benchmarks triggered a case in which both SiVR and SoVR transformations were performed.
  - The paper includes an example from a past approach (Calland et al.) that triggers both SiVR and SoVR transformations
- Compilation time is not an issue, even with ILP solver

Performance comparison: Variants

- Four variants
  - Original kernel (Relying on ICC auto-vectorizer)
    - Considers neither SoVR nor SiVR
  - Vectorized kernel after applying Calland's approaches
    - Considers only SoVR
  - Vectorized kernel after applying Chu's approach
    - Considers both SoVR and SiVR; but sub-optimal
  - Vectorized kernel after applying PolySIMD (our) approach
    - Considers both SoVR and SiVR, and optimal

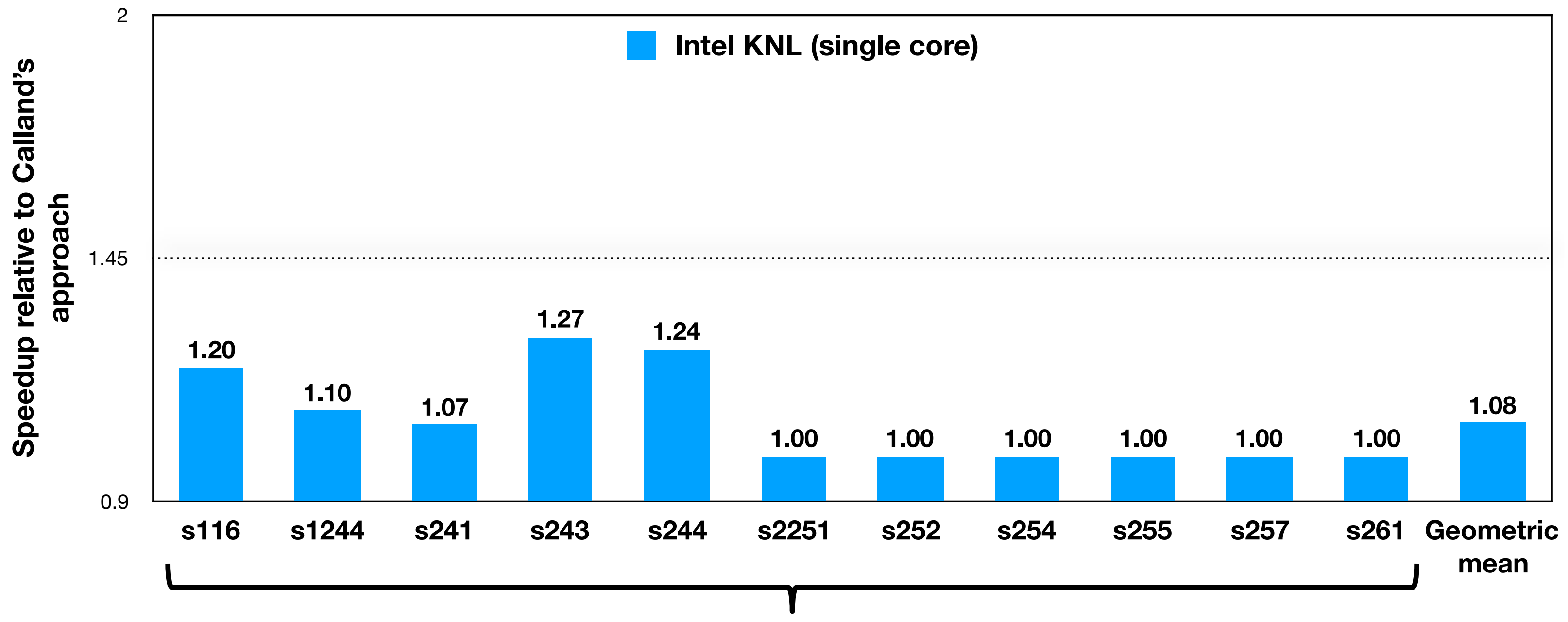
- Past approach variants (Calland's and Chu's) are generated by our tool by bypassing solver and feeding the solutions

1) Comparison with ICC + O3 (auto-vectorizer)



- S116: involves lot of unaligned memory stores and loads
- S1244: involves dead writes && O3 eliminates them
- Most of dependences in s252, s254, s255, s257 are on scalars, and replacing with arrays didn't result in much improvement
- ICC required all of them to be reordered for vectorization.

2) Comparison with Calland's approach

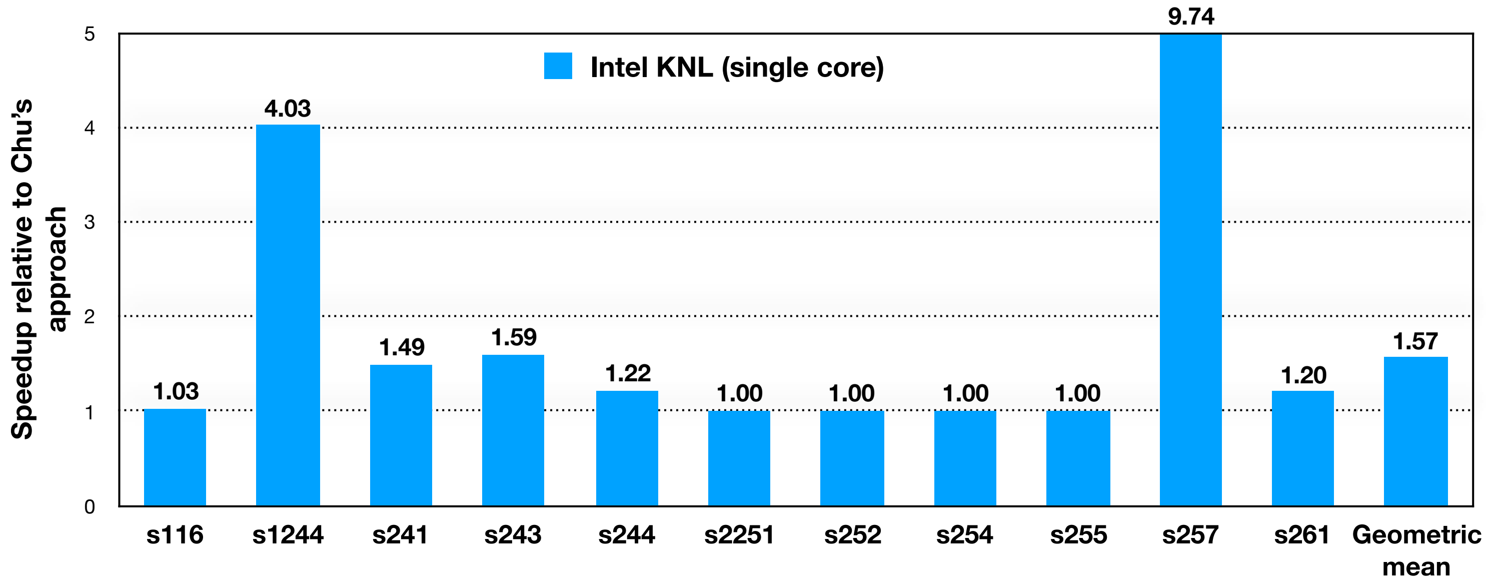


Calland's approach: SiVR

— Speedup's from s116 to s244
- Our approach chose SoVR for s116-s244 benchmarks since SoVR incurs less overhead compared to SiVR

3) Comparison with Chu's approach

- Chu's approach – considers both SoVR and SiVR
  - No global view of cycles and transformations that can break them



- s241, s243, s257, s261
  - Chu's approach applied redundant SoVR/SiVR transformations
- s116, s1244, s244
  - Chu's approach generates temporary arrays for SoVR

Coverage: Vectorization summary

kernel	ICC + O3 (Auto vec)	Calland's approach	Chu's approach	Our approach
s116	No	Yes	Yes	Yes
s1244	Yes	Yes	Yes	Yes
s241	No	Yes	Yes	Yes
s243	No	Yes	Yes	Yes
s244	No	Yes	Yes	Yes
s2251	No	Yes	Yes	Yes
s252	No	Yes	Yes	Yes
s254	No	Yes	Yes	Yes
s255	No	Yes	Yes	Yes
s257	No	Yes	Yes	Yes
s261	No	Yes	Yes	Yes
	4.61x	1.08x	1.57x	

Geometric mean speedup of our approach

Evaluation (GPU)

- We use the same 11 benchmarks from TSVC
- Experimental setup
  - On Nvidia Volta device
    - For consistency with single-threaded execution on KNL, we use single block (1024 threads)
    - Compiler & flags: NVCC v9.1 -O3 -arch=sm 70 -ccbin=icc
- Since original program cannot be run on GPU, we compare our approach with only past approaches (using code generated by our compiler in all cases)

Performance comparison on GPU

kernel	Speedup over Calland's approach	Speedup over Chu's approach
s116	1.29x	1.27x
s1244	1.57x	1.51x
s241	1.31x	1.70x
s243	1.47x	1.61x
s244	1.12x	1.32x
s2251	1.00x	1.00x
s252	1.00x	1.00x
s254	1.00x	1.00x
s255	1.00x	1.00x
s257	1.00x	1.08x
s261	1.00x	1.19x
	1.14x	**1.22x**

Geometric mean speedup of our approach

Our approach chose SoVR instead of SiVR

Redundant transformations, and generation of array temporaries

Conclusions & Future work

- Our contributions
  1. Unify multiple storage transformations to break cycles optimally
  2. Restricting additional space required by transformations
     - For details, we refer to the paper
  3. Evaluation in the context of mature loop-optimization framework

- Future work
  - Extend our framework with additional storage transformations
  - Systematically leverage storage transformations to enable loop optimizations

Any questions?