Unlocking Vectorization Scope: Extensible Vectorization via Unified Dependence Semantics

Shihan Fang

Shanghai Jiao Tong University
fang-account@sjtu.edu.cn

Overview

Modern processors deliver high throughput via SIMD execution, and auto-vectorization aims to automatically exploit this parallelism from ordinary code by identifying independent, isomorphic computations and legally reordering them for SIMD execution.

Current compiler vectorizers mainly rely on Superword Level Parallelism (SLP) [1] and Loop Level Parallelism (LLP) and improve performance in many cases, but they share a common weakness: the vectorization logic is *algorithm-centric*, embedding vectorization decisions inside specific vectorization algorithms. As a result, supporting new forms of parallelism often requires redesigning or adding new algorithms, making these approaches difficult to extend.

Key insight:

- Vectorization should be *dependence-centric*, driven by dependence structure rather than hard-coded vectorization patterns.
- *Extensibility* should be a first-class design goal for vectorization: vectorization opportunities evolve, frameworks should remain stable.

We introduce a ***dependence-centric***, ***extensible*** vectorization framework built on VIR, an intermediate representation extending PSSA [2] that unifies data and control dependences, and enables systematic integration of new vectorization strategies beyond prior approaches.

Methodology

Design Principles:

- ***Dependence-centric.*** A unified IR explicitly encodes data and control dependences as the semantic foundation for vectorization.
- ***Extensible.*** Express vectorization as generic traversals over semantic object, enabling new strategies without re-designing the framework.

1. VIR: Unified Dependence Representation

A *predicated* IR that replaces basic blocks with a list of

(`Control Predicate`, `Iterator`, `Item`)

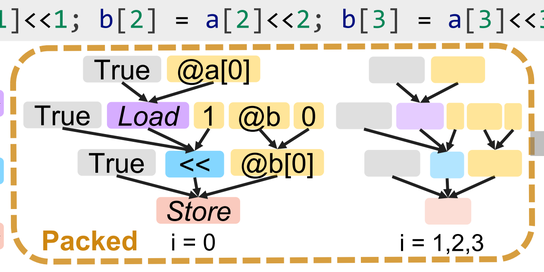
- `Control Predicate`: boolean formula (from PSSA [2]) that guards execution, encoding control.
- `Iterator`: explicit loop iteration pattern.
- `Item`: `Instruction` or `Loop` (a nested layer)

```
for (int i = 0; i < LENGTH; i = i + 2)
  dst[i] = src[i] + 1;
```

```
[CP]: (bool true)
-> int length = load ptr @LENGTH
[CP]: (bool true)
-> cmp bool loop_cond = 0 < int length
[CP]: (bool loop_cond)
-> with
     int {iter} = iota( 0, {iter_rec}, + 2)
   do
     [CP]: (bool true) | [0, int length) + 2
     -> int load_value = load ptr <@src[{iter}]>
     [CP]: (bool true) | [0, int length) + 2
     -> int store_value = int load_value + 1
     [CP]: (bool true) | [0, int length) + 2
     -> store int store_value ptr <@dst[{iter}]>
     [CP]: (bool true) | [0, int length) + 2
     -> {iter_rec} = {iter} + 2
     [CP]: (bool true) | [0, int length) + 2
     -> cmp bool latch = {iter_rec} < int length
   while [CP]: (bool latch)
```

Control Predicate / *Iterator* / *Instruction* / *Item* / *Loop Item* / *An Entry*

```
b[0] = a[0]<<1; b[1] = a[1]<<1; b[2] = a[2]<<2; b[3] = a[3]<<3;
[CP]: (bool true)
-> int x = load ptr <@a[i]>
[CP]: (bool true)
-> int x_ = int x << i
[CP]: (bool true)
-> store int x_ ptr <@b[i]>
```

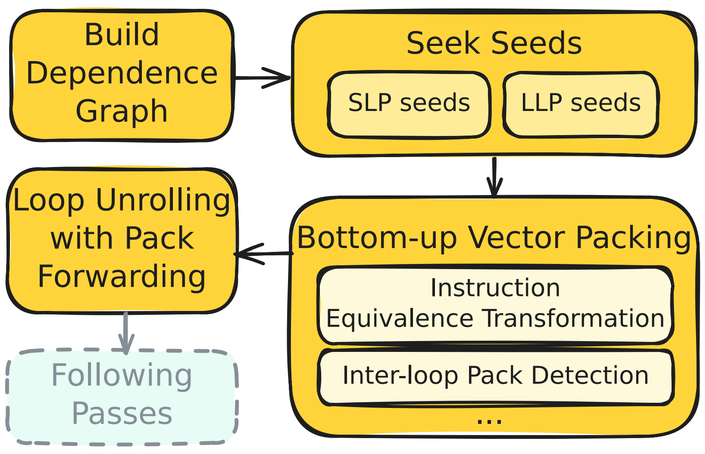



```
[CP]: (bool true)                          Vectorized VIR
-> vec(4) v = load ptr <@a[0]>
[CP]: (bool true)
-> vec(4) c_v = vec(4) v << vec<1,1,2,3>
[CP]: (bool true)
-> store vec(4) c_v ptr <@b[0]>
```

2. Dependence-centric Framework

Dependence graph.

- Hierarchical, tree-like representation on VIR
- Unifies data, control, and iteration dependences
- Vectorization is performed by traversing and matching structure on it.

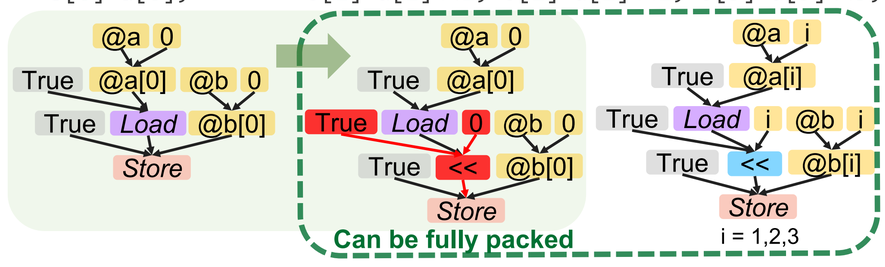


- Packing begins from *seeds*
  - SLP *seeds*: adjacent stores
  - LLP *seeds*: stores across loop iterations
- Proceeds along the tree. Merge nodes into *packs* if they are independent and isomorphic.
- Explores subtrees recursively.

Extensible mechanisms to uncover additional vectorization during vector packing.

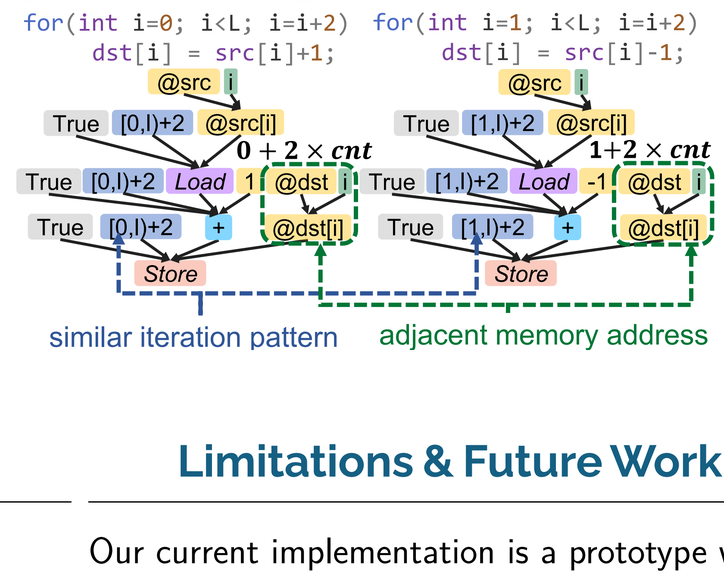
▷ **Instruction Equivalence Transformation.**
Apply pluggable, semantically equivalent transformations to enable packing.

```
b[0]=a[0];        b[1]=a[1]<<1; b[2]=a[2]<<2; b[3]=a[3]<<3;
```



▷ **Inter-loop Pack Detection.**
Pack across loops by resolving *iterators* to detect adjacent memory accesses, enabling fusion and vectorization.

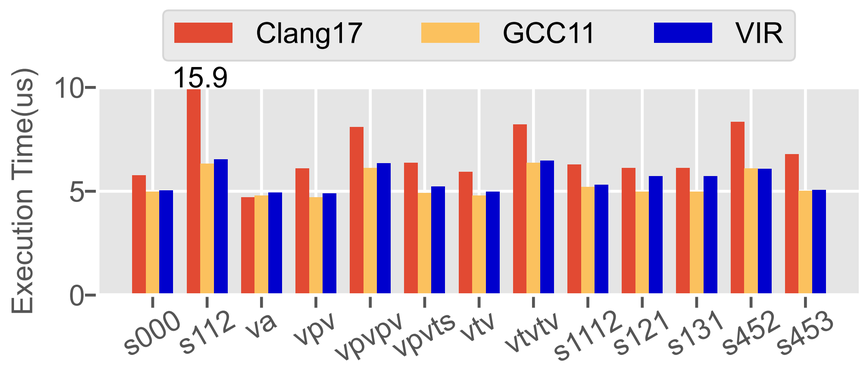
```
for(int i=0; i<L; i=i+2)      for(int i=1; i<L; i=i+2)
  dst[i] = src[i]+1;            dst[i] = src[i]-1;
```



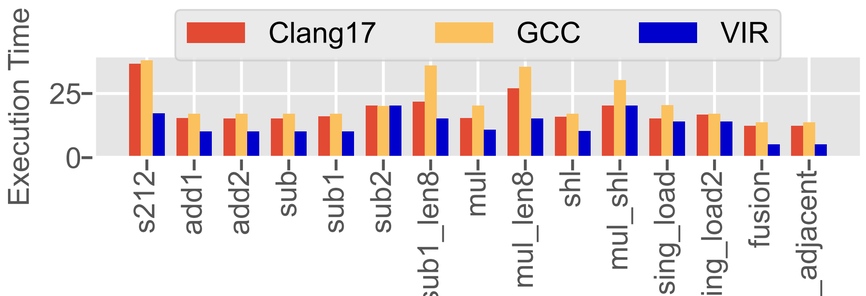
Experiments

We used Clang 17 and GCC 11 as baselines, compiling to x86 AVX2 (`-mavx2 -O3`) with inlining disabled. Experiments ran on an Intel Ultra 7 PC (96GB RAM) with Turbo Boost off.

▷ **Test cases from `tsvc` [3].** Achieves +27% vs Clang (avg. +15%) and -5% vs GCC on average.



▷ **Test cases for image pixel processing.** Achieves up to +53% vs Clang (avg. +27%) and up to +58% vs GCC (avg. +39%).



▷ **A real word case from `raylib` [4].**

```
color[0] = (hexValue >> 24) & 255;
color[1] = (hexValue >> 16) & 255;
color[2] = (hexValue >> 8) & 255;
color[3] = hexValue & 255;
```

(a) Source Code

```
%1 = load i32, ptr @hexValue, align 4
%2 = lshr i32 %1, 24
store i32 %2, ptr @color, align 16
%3 = lshr i32 %1, 16
%4 = and i32 %3, 255
%5 = getelementptr [4 x i32], ptr @color,i64 0,i64 1
store i32 %4, ptr %5, align 4
%6 = lshr i32 %1, 8
%7 = and i32 %6, 255
%8 = getelementptr [4 x i32], ptr @color,i64 0,i64 2
store i32 %7, ptr %8, align 8
%9 = and i32 %1, 255
%10 = getelementptr [4 x i32], ptr @color,i64 0,i64 3
store i32 %9, ptr %10, align 4
```

Clang17

```
%1 = load i32, ptr @hexValue, align 4
%2 = insertelement <4 x i32>
     <i32 0, i32 0, i32 0, i32 0>, i32 %1, i32 0
%3 = insertelement <4 x i32> %2, i32 %1, i32 1
%4 = insertelement <4 x i32> %3, i32 %1, i32 2
%5 = insertelement <4 x i32> %4, i32 %1, i32 3
%6 = ashr <4 x i32> %5, <i32 24,i32 16,i32 8,i32 0>
%7 = and <4xi32> %6,<i32 255,i32 255,i32 255,i32 255>
%8 = getelementptr [4 x i32], ptr @color,i32 0,i32 0
store <4 x i32> %7, ptr %8, align 4
```

VIR

(b) LLVM IR Generated by Clang 17 and VIR

Limitations & Future Work

Our current implementation is a prototype with a limited set of vectorization strategies. We plan to

- Integrate the framework into the LLVM toolchain to evaluate its effectiveness on real-world applications and existing optimization pipelines.
- Add richer vectorization strategies as plugins.
- Develop pruning and ranking heuristics to scale to larger and more complex programs.

References

[1] Samuel Larsen et al., "Exploiting superword level parallelism with multimedia instruction sets", Acm Sigplan Notices, 2000.

[2] Yishen Chen et al., "All you need is superword-level parallelism: systematic control-flow vectorization with SLP", PLDI, 2022.

[3] TSVC: Test Suite for Vectorizing Compilers, https://github.com/UoB-HPC/TSVC_2, accessed Jan 2026.

[4] raylib, https://github.com/raysan5/raylib, accessed Apr 2025.