Auto-Vectorization in Compilers: Leveraging SIMD for High-Performance Computing

Content

01

Background

02

Predicated SSA

PLDI22

Compilation and Compiler

- **<u>Compilation</u>** is the process of translating **high-level programming language** (e.g. C++, Python) source code into **low-level machine-executable** (e.g., assembly).

- A **<u>compiler</u>** is a software tool that translates source code into target machine code, often applying optimizations to improve performance and efficiency.

$$\begin{tikzcd}
\text{Source Code} \arrow[r] & \text{Abstract Syntax Tree (AST)} \arrow[r] & \text{Intermediate Representation} \arrow[r, "\textbf{Optimization Passes}"] & {} \arrow[r] & \text{Machine Code}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Source Code} \arrow[r, "\textbf{Clang}"] & \text{LLVM IR} \arrow[r, "\textbf{Opt. Passes}"] & {} \arrow[r] & {}
\end{tikzcd}$$

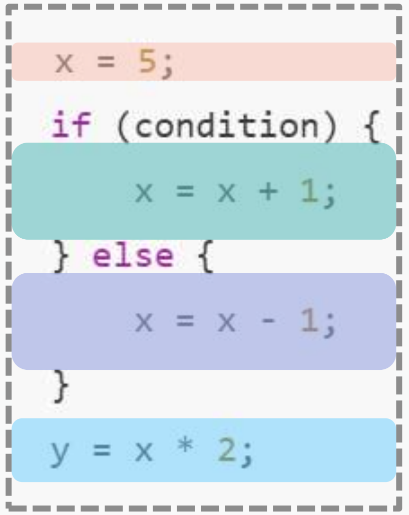
Some Concepts in Compiler

Control Flow Analysis

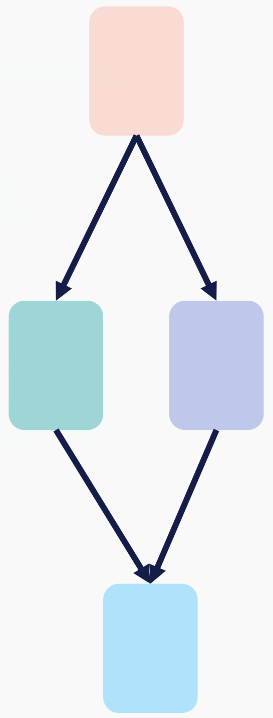
- **Basic Block** is a sequence of consecutive instructions in a program. These instructions execute in sequence under same condition.
  - Basic blocks form a program's Control Flow Graph (CFG), where edges represent possible execution paths between blocks.

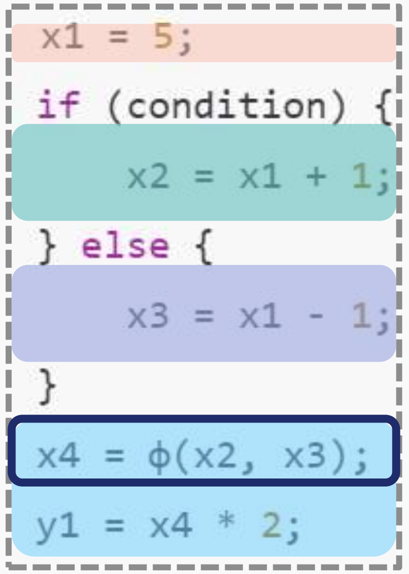
Dataflow Analysis

- **Static Single Assignment (SSA)** is an intermediate representation (IR) used in modern compilers to simplify optimization and analysis.
  - **LLVM IR follows SSA form.**
  - Key Property: Each variable is assigned (defined) exactly once in the program.
    - *Phi (Φ) functions* handle values coming from multiple control paths.



```
x = 5;
if (condition) {
    x = x + 1;
} else {
    x = x - 1;
}
y = x * 2;
```





```
x1 = 5;
if (condition) {
    x2 = x1 + 1;
} else {
    x3 = x1 - 1;
}
x4 = ϕ(x2, x3);
y1 = x4 * 2;
```

SIMD

- processors now include SIMD registers and instructions, to enable more parallelism.
  - Single Instruction Multiple Data
  - vector registers are extra wide
    - a 512-bit wide register can hold 16 32-bit words.
  - Instructions on these registers perform the **same operation** on each of the packed data types.

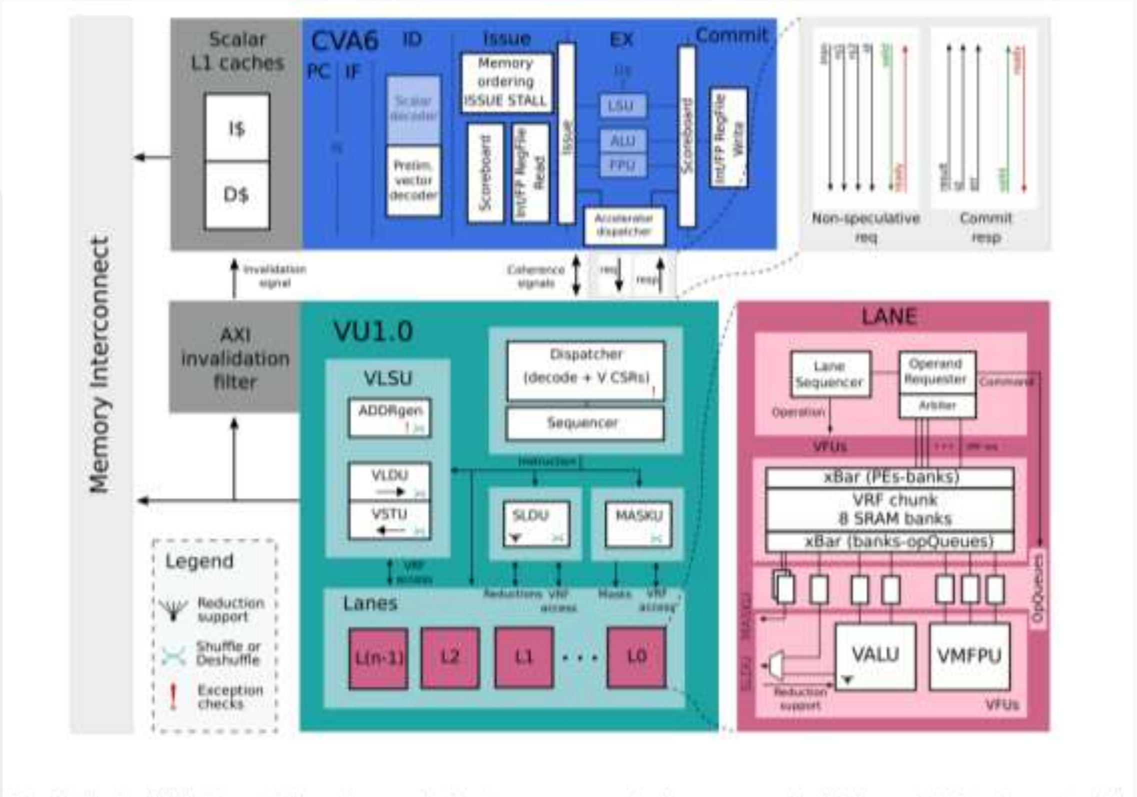


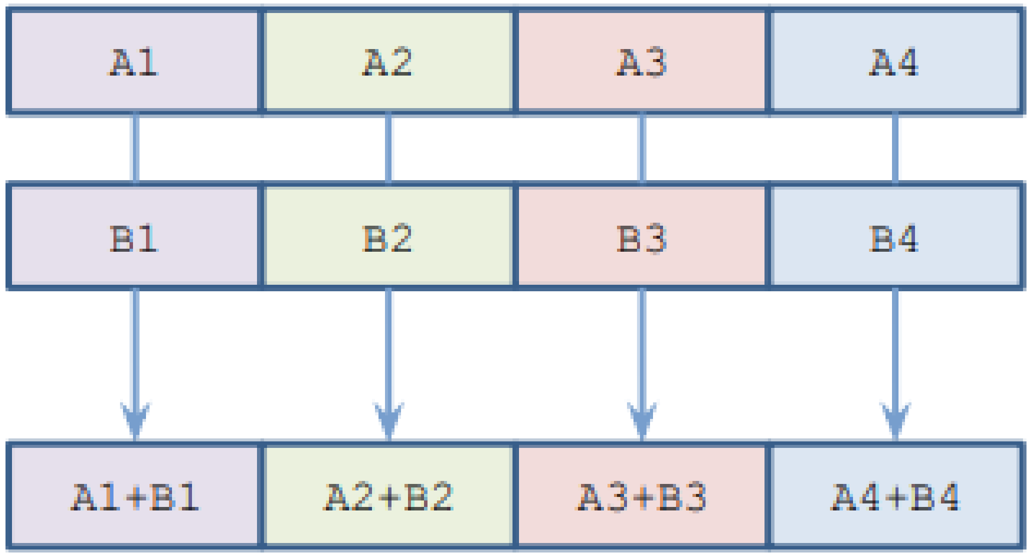
Figure 1. Top-Level block diagram of (the new) system with the vector co-processor marked in green, a more detailed diagram of the lane in magenta, and the host scalar core CVA6 in blue.

M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli and L. Benini, "A “New Ara” for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design," 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP), Gothenburg, Sweden, 2022, pp. 43-51, doi: 10.1109/ASAP54787.2022.00017. keywords: {Program processors;Source coding;Systems architecture;Computer architecture;Throughput;Vector processors;Kernel;RISC-V;ISA;Vector;Efficiency},

SIMD

```c
void vec_add_rvv(vint32m1_t *a, vint32m1_t *b, vint32m1_t *c, size_t vl) {
 *c = vadd_vv_i32m1(*a, *b, vl);
}
```

```
1020:        0287ec87                vl1re32.v       v25,(a5)
1024:        fe043783                ld      a5,-32(s0)
1028:        0287ec07                vl1re32.v       v24,(a5)
102c:        fd043783                ld      a5,-48(s0)
1030:        0507f057                vsetvli zero,a5,e32,m1,ta,mu
1034:        039c0c57                vadd.vv v24,v25,v24
1038:        fd843783                ld      a5,-40(s0)
103c:        02878c27                vs1r.v  v24,(a5)
```



Automatic Vectorization

- Loop Vectorize

1024 scalar addition

```
for (i = 0; i < 1024; i++)
    c[i] = a[i] * b[i];
```

256 vector addition

```
for (i = 0; i += 4)
    c[i:i+3] = a[i:i+3] * b[i:i+3];
```

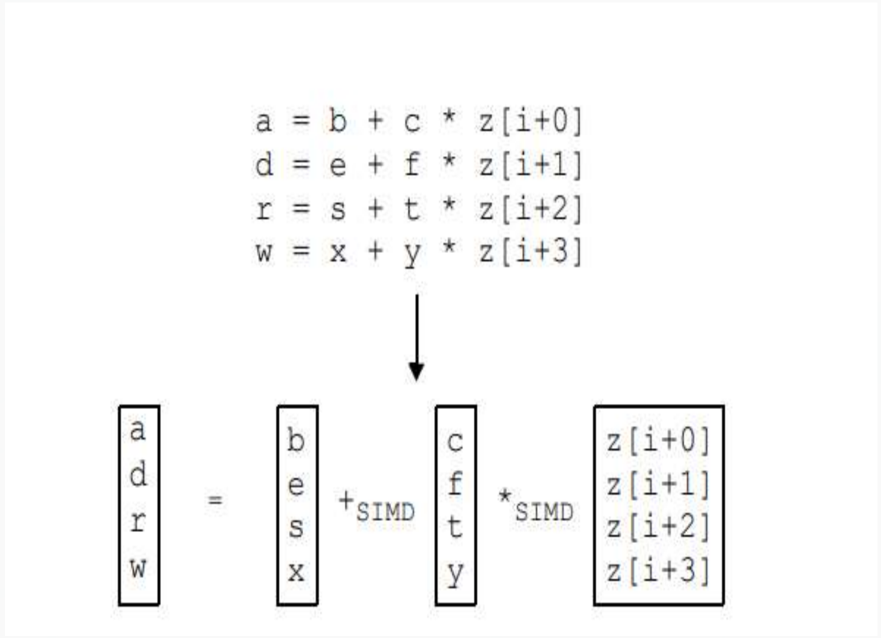
- SLP(superword level parallelism) Vectorize

```
a = b + c * z[i+0]
d = e + f * z[i+1]
r = s + t * z[i+2]
w = x + y * z[i+3]
```

$$\boxed{\begin{matrix}a\\d\\r\\w\end{matrix}} = \boxed{\begin{matrix}b\\e\\s\\x\end{matrix}} +_{\mathrm{SIMD}} \boxed{\begin{matrix}c\\f\\t\\y\end{matrix}} *_{\mathrm{SIMD}} \boxed{\begin{matrix}z[i+0]\\z[i+1]\\z[i+2]\\z[i+3]\end{matrix}}$$

Auto-Vectorization: SLP

- The operations execute in lock-step.
- The inputs and outputs of the operations reside in **packed storage** (usually implemented as vector registers).
- The operations are **isomorphic**.
- The operations are applied **elementwise**.

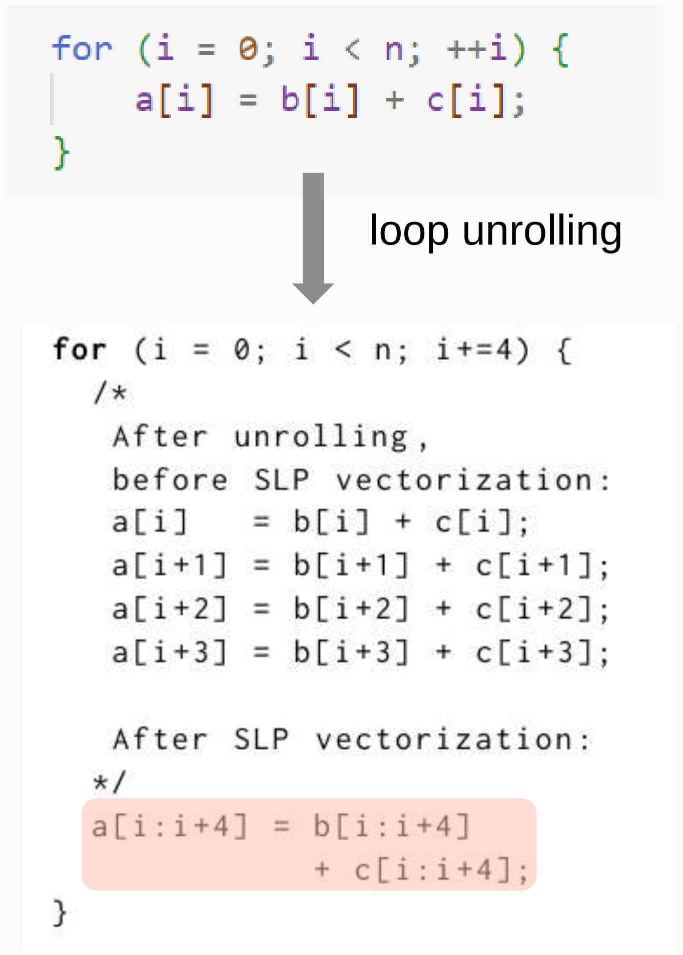


Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00). Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

All You Need Is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP

All You Need is SLP

- <u>SLP vectorization is simpler</u>: not requiring loop dependence analysis to reason with loop-carried dependence
- <u>SLP vectorization is more flexible:</u> more heuristic regarding new SIMD instructions can be used to extend the search space
- Loop Level Parallelism (LLP) can be exploited with Superword Level Parallelism (SLP) after unrolling with appropriate factor.



```
for (i = 0; i < n; ++i) {
    a[i] = b[i] + c[i];
}
```

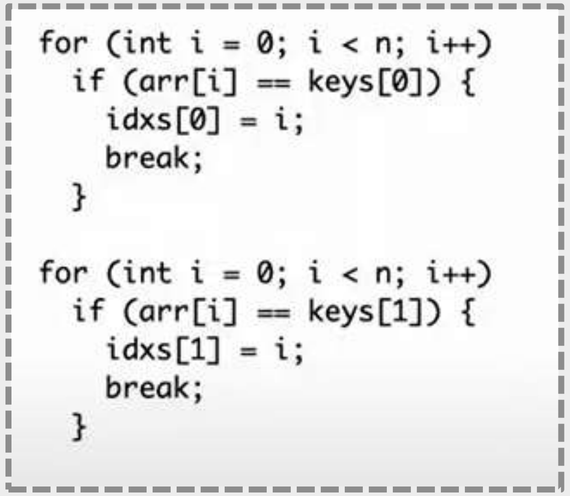
```
for (i = 0; i < n; i+=4) {
  /*
   After unrolling,
   before SLP vectorization:
   a[i]   = b[i] + c[i];
   a[i+1] = b[i+1] + c[i+1];
   a[i+2] = b[i+2] + c[i+2];
   a[i+3] = b[i+3] + c[i+3];

   After SLP vectorization:
  */
  a[i:i+4] = b[i:i+4]
               + c[i:i+4];
}
```

But SLP's capacity is limited on CFG-based IRs

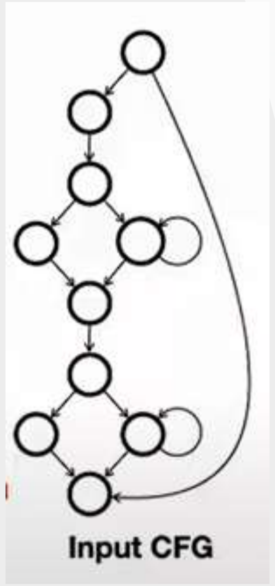
SLP's capacity is limited on CFG-based IRs

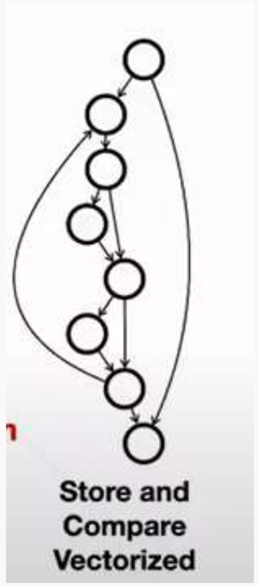
- **SLP vectorization is always limited in one Basic Block**
  - Packing multiple instructions into one is a lot about code motion.
  - Vectorizing instructions in different basic block requires code motion across *block boundaries* and thus requires control flow transformation.

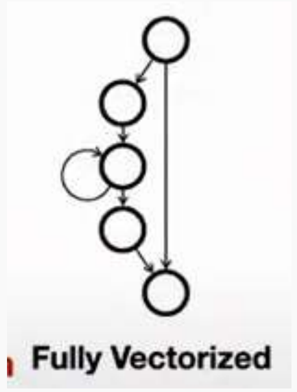


```
for (int i = 0; i < n; i++)
  if (arr[i] == keys[0]) {
    idxs[0] = i;
    break;
  }

for (int i = 0; i < n; i++)
  if (arr[i] == keys[1]) {
    idxs[1] = i;
    break;
  }
```







Need global code motion and CFG reconstruction to exploit the parallelism

Predicated SSA

An IR to Facilitate Code Motion.

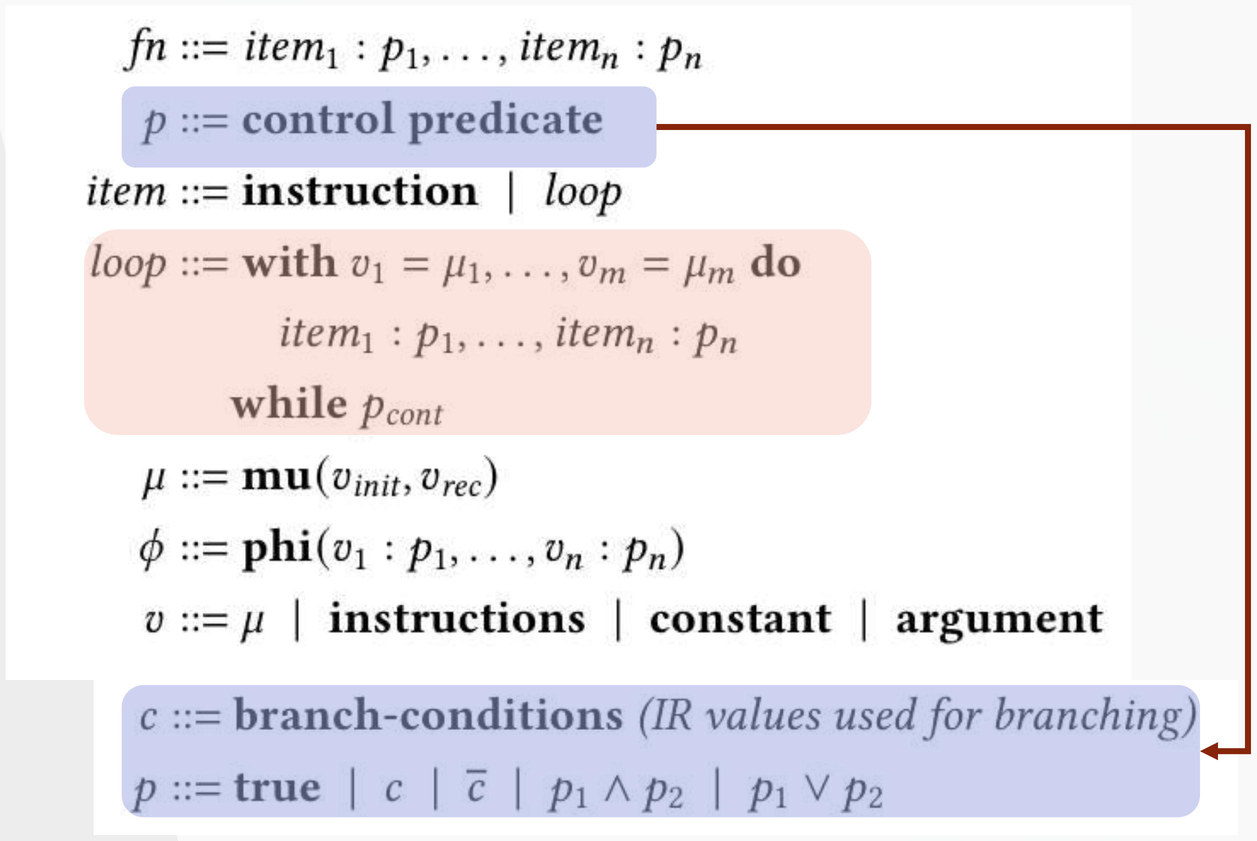
- Transform CFG to straight-line code.
- Encode control flow with ***Predicate,*** bool expression indicating whether the instruction can execute.

control predicate *Item: Instruction | Loop*

- *Item: Instruction | Loop*
- *[CP]: control predicate*
  - *True* indicate the *Item* executes unconditionally

Loops are defined in a nested manner

- *Mu instructions* and the *latch predicate* preserve the control relation of loop



$$fn ::= item_1 : p_1, \ldots, item_n : p_n$$

$$p ::= \textbf{control predicate}$$

$$item ::= \textbf{instruction} \mid loop$$

$$\begin{aligned} loop ::= {} & \textbf{with } v_1 = \mu_1, \ldots, v_m = \mu_m \textbf{ do} \\ & \quad item_1 : p_1, \ldots, item_n : p_n \\ & \textbf{while } p_{cont} \end{aligned}$$

$$\mu ::= \textbf{mu}(v_{init}, v_{rec})$$

$$\phi ::= \textbf{phi}(v_1 : p_1, \ldots, v_n : p_n)$$

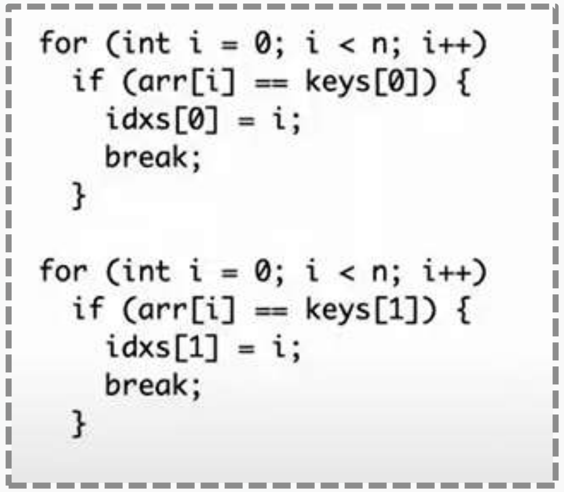
$$v ::= \mu \mid \textbf{instructions} \mid \textbf{constant} \mid \textbf{argument}$$

$$c ::= \textbf{branch-conditions} \ \textit{(IR values used for branching)}$$

$$p ::= \textbf{true} \mid c \mid \bar{c} \mid p_1 \wedge p_2 \mid p_1 \vee p_2$$

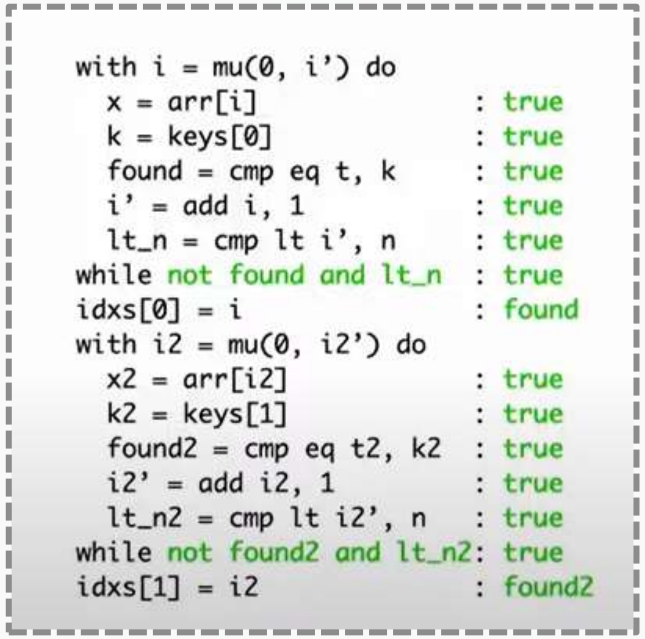
Predicated SSA

- Convert CFG of basic block into ‘super BB’ of *Items*.
  - Wider space to do code motion
- Cover control flow information in *Predicate.*
  - **Make vectorization between different loops practical.**
    - *Loop Fusion*
    - *Loop Co-iteration*: loops with similar iterate mode can be transferred in to one loop

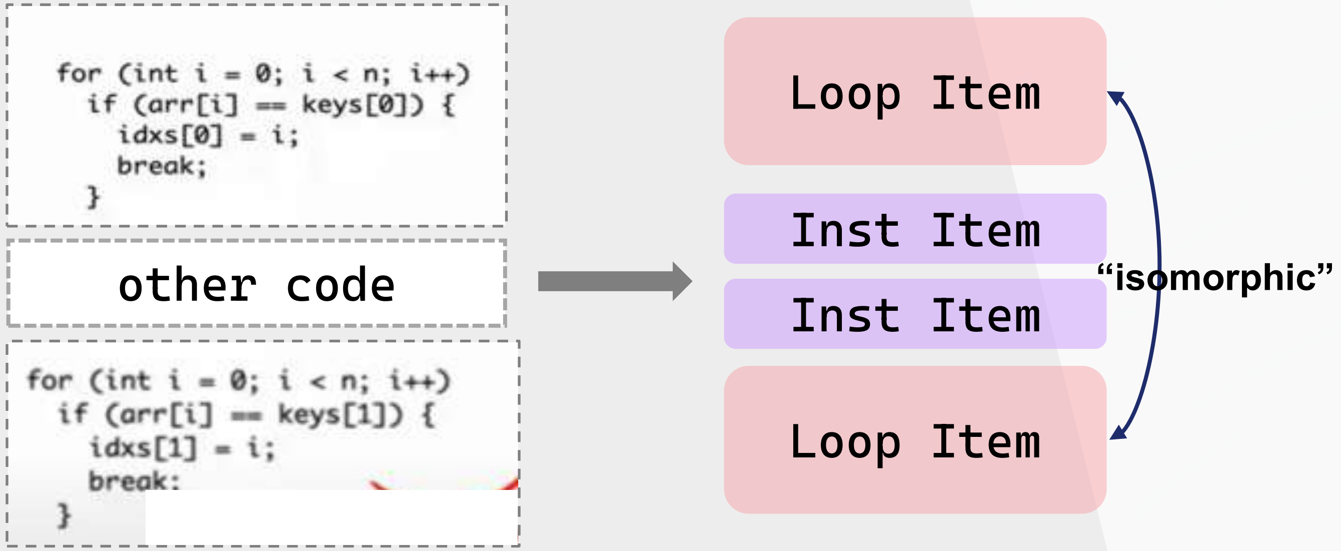


```
for (int i = 0; i < n; i++)
  if (arr[i] == keys[0]) {
    idxs[0] = i;
    break;
  }

for (int i = 0; i < n; i++)
  if (arr[i] == keys[1]) {
    idxs[1] = i;
    break;
  }
```



```
with i = mu(0, i’) do
  x = arr[i]                  : true
  k = keys[0]                 : true
  found = cmp eq t, k         : true
  i’ = add i, 1               : true
  lt_n = cmp lt i’, n         : true
while not found and lt_n      : true
idxs[0] = i                   : found
with i2 = mu(0, i2’) do
  x2 = arr[i2]                : true
  k2 = keys[1]                : true
  found2 = cmp eq t2, k2      : true
  i2’ = add i2, 1             : true
  lt_n2 = cmp lt i2’, n       : true
while not found2 and lt_n2: true
idxs[1] = i2                  : found2
```



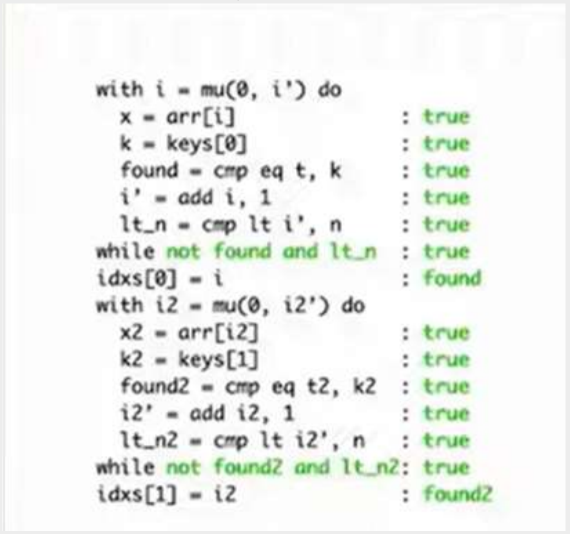
```
for (int i = 0; i < n; i++)
  if (arr[i] == keys[0]) {
    idxs[0] = i;
    break;
  }
```

```
for (int i = 0; i < n; i++)
  if (arr[i] == keys[1]) {
    idxs[1] = i;
    break:
  }
```

Vectorization Pipeline with Predicated SSA

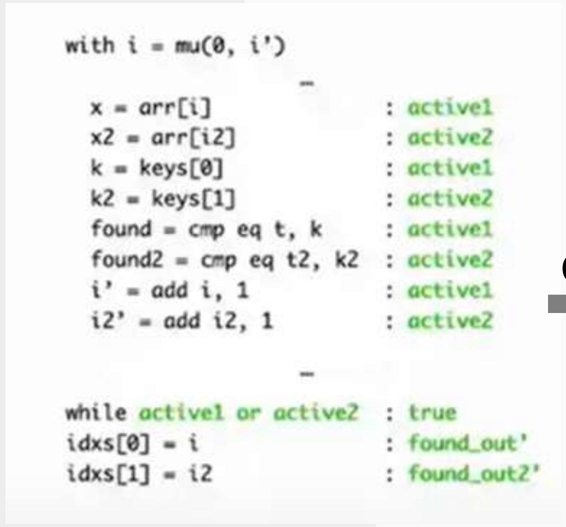
LLVM IR

Convert to PSSA



```
with i = mu(0, i') do
  x = arr[i]                  : true
  k = keys[0]                 : true
  found = cmp eq t, k         : true
  i' = add i, 1               : true
  lt_n = cmp lt i', n         : true
while not found and lt_n      : true
idxs[0] = i                   : found
with i2 = mu(0, i2') do
  x2 = arr[i2]                : true
  k2 = keys[1]                : true
  found2 = cmp eq t2, k2      : true
  i2' = add i2, 1             : true
  lt_n2 = cmp lt i2', n       : true
while not found2 and lt_n2: true
idxs[1] = i2                  : found2
```

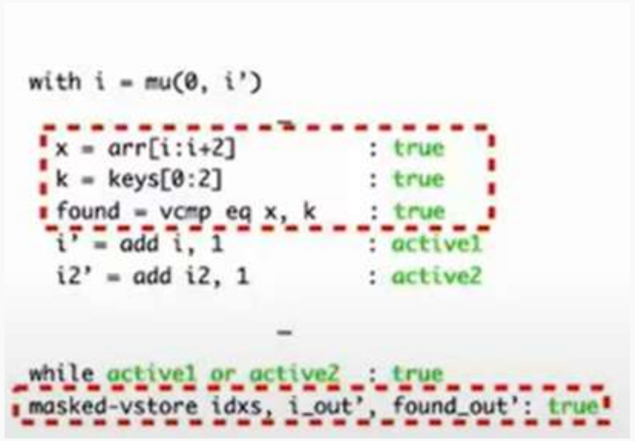
Scheduling
(code motion)



```
with i = mu(0, i')
                  …
  x = arr[i]                 : active1
  x2 = arr[i2]               : active2
  k = keys[0]                : active1
  k2 = keys[1]               : active2
  found = cmp eq t, k        : active1
  found2 = cmp eq t2, k2     : active2
  i' = add i, 1              : active1
  i2' = add i2, 1            : active2

                  …
while active1 or active2     : true
idxs[0] = i                  : found_out'
idxs[1] = i2                 : found_out2'
```

Vector
Instruction
Generation



```
with i = mu(0, i')

  x = arr[i:i+2]                  : true
  k = keys[0:2]                   : true
  found = vcmp eq x, k            : true
  i' = add i, 1                   : active1
  i2' = add i2, 1                 : active2

                    …
while active1 or active2    : true
masked-vstore idxs, i_out', found_out': true
```

Convert to
CFG-based IR

LLVM IR

Evaluation

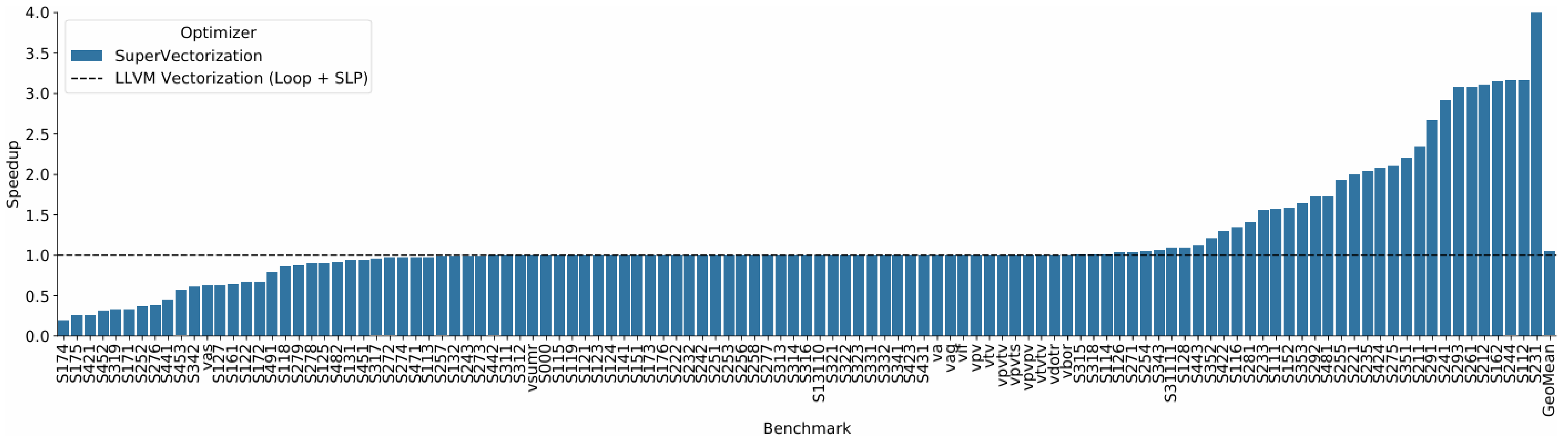


Figure 13. Speedup over LLVM scalar optimizations (-O3) on TSVC. The benchmarks are sorted (in increasing order) according to speedup from SuperVectorization over LLVM's scalar optimizations.

Evaluation

benchmark for polyhedral optimizations

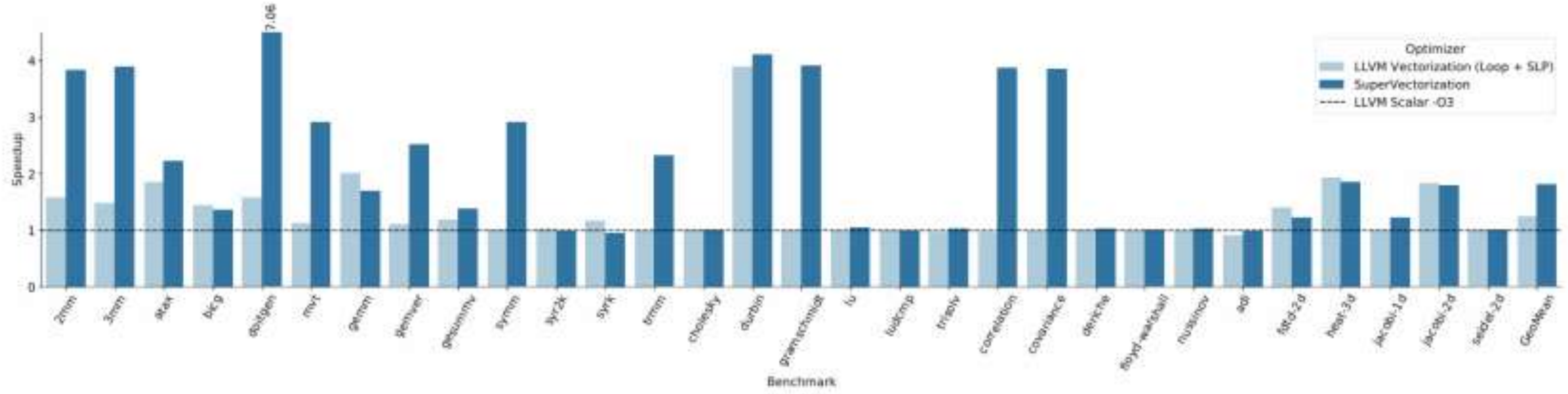


Figure 14. Speedup over LLVM scalar optimizations (-O3) on PolyBench

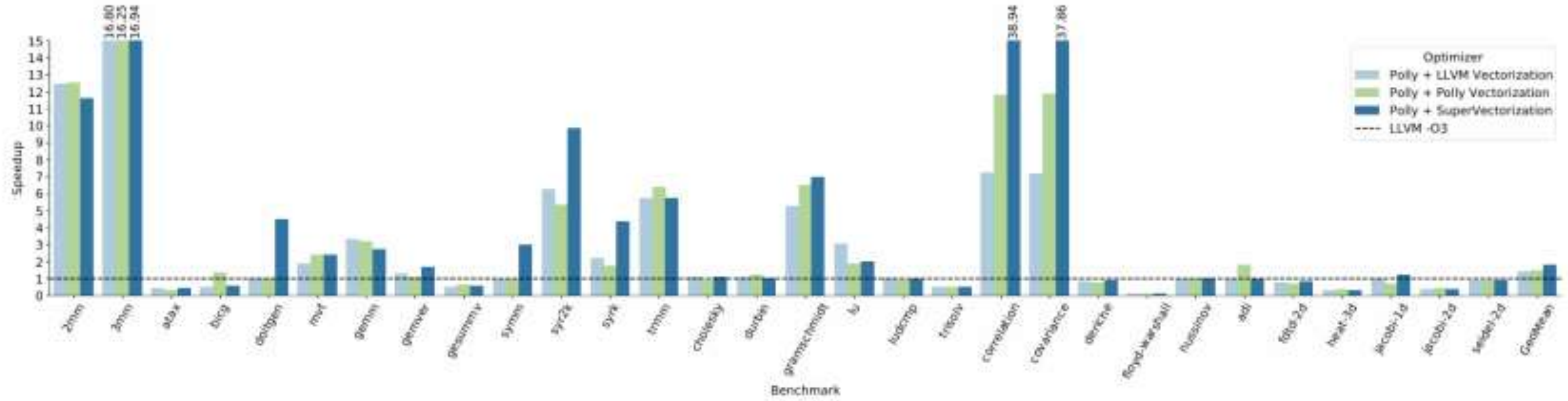
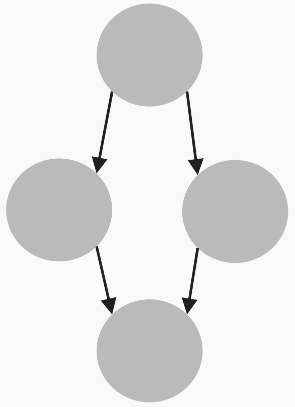


Figure 15. Speedup over full LLVM optimizations (-O3 + vectorization) on PolyBench using Polly

Some Thoughts

- **Vectorization has a lot to do with code motion (scheduling)**
  - Independent instructions in a basic block can be reordered and transform into vector instruction
  - Instructions in different basic block can be viewed as have some kind of dependency.



We care about <u>dependence analysis</u>!

Contains pseudo dependence

- Control equivalent instructions in different blocks has pseudo dependence

- **Dependence**
  - Control dependence
  - Use-define data dependence
  - Memory access data dependence

SSA make this trivial globally

Gotten from the control flow and the memory access pattern of instructions in each block

Some Thoughts

- **Dependence**
  - Control dependence → Contains pseudo dependence
    - Control equivalent instructions in different blocks has pseudo dependence
  - Use-define data dependence
  - Memory access data dependence

Control dependence can be transferred to data dependence using predication

- That’s what PSSA do!
- And emitting the pseudo dependence in loop conditions is the season why it can perform loop fusion and co-iteration well.

- PSSA is an IR to Facilitate Code Motion.
- Personally, I prefer to call it an IR to facilitate dependence analysis. 😳

Thank you for listening