Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements

Vasileios Porpodas; Rodrigo C. O. Rocha; Evgueni Brevnov; Luís F. W. Góes; Timothy Mattson

Group 5*: **Hongru Lu; Yining Wang; Yichao Yuan***

Table of Contents

- **Introduction & Motivation**
  Vectorization & SIMD
  Super - Level Parallelism (SLP)
- **Super Node SLP Algorithm**
  Algebraic Background
  Examples: Leave Node & Trunk Node
  Algorithm
- **Results & Evaluations**
- **Commentary**

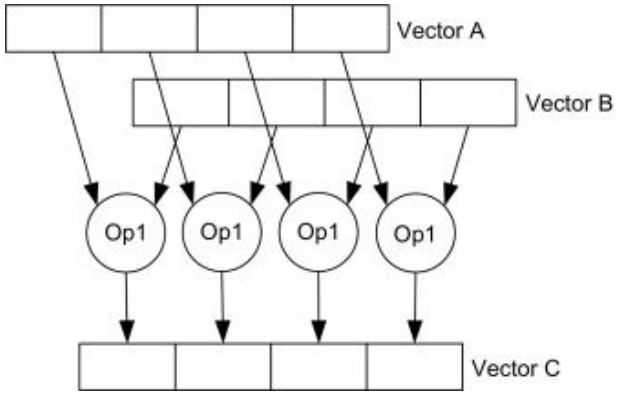
Introduction - Vectorization & SIMD

```
b = a[i+0]
c = 5
d = b+c
e = a[i+1]
f = 6
g = e+f
```

```
b = a[i+0]
e = a[i+1]

c = 5
f = 6

d = b + c
g = e + f
```



Single Instruction, Multiple Data (SIMD)

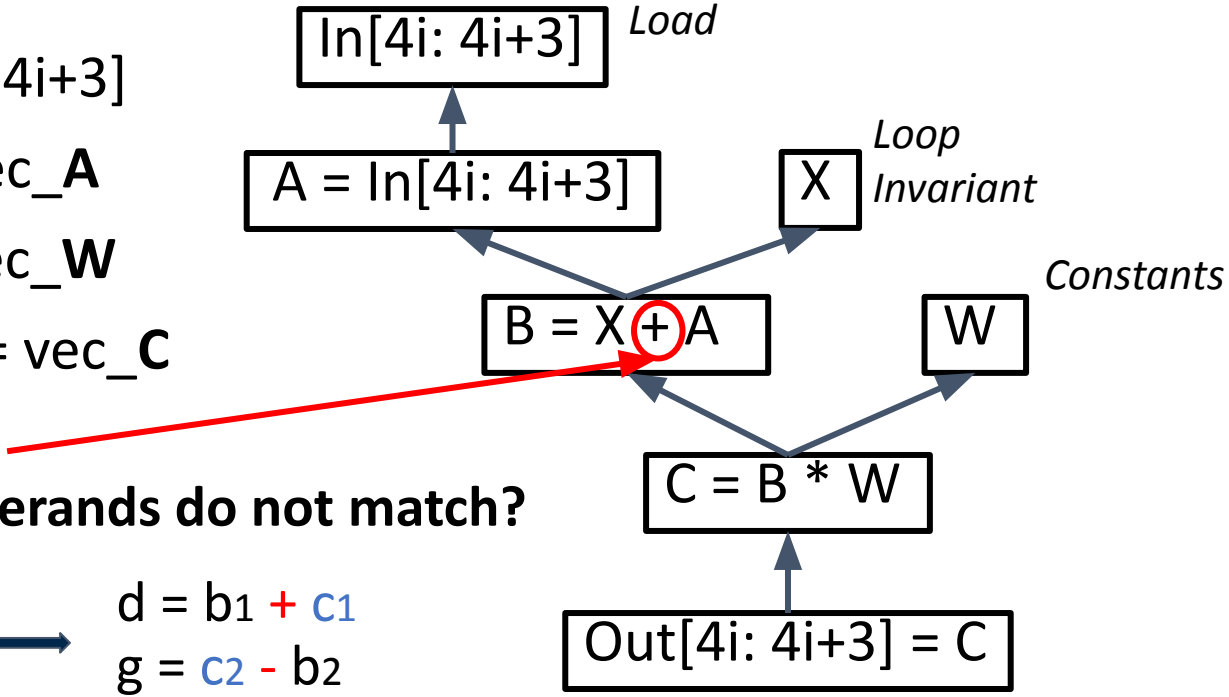
Introduction - Superword-Level Parallelism (SLP)

vec_**A** = vec_**In**[4i: 4i+3]

vec_**B** = vec_**X** + vec_**A**

vec_**C** = vec_**B** * vec_**W**

vec_**Out**[4i: 4i+3] = vec_**C**



What if operators/operands do not match?

$d = b_1 + c_1$
$g = b_2 + c_2$

$\longrightarrow$

$d = b_1 + c_1$
$g = c_2 - b_2$

SuperNode SLP - Algebraic Background

Commutative and associative operators allows:

- Substitution with inverse operators

A + B **- C**  A + B + (**-C**)

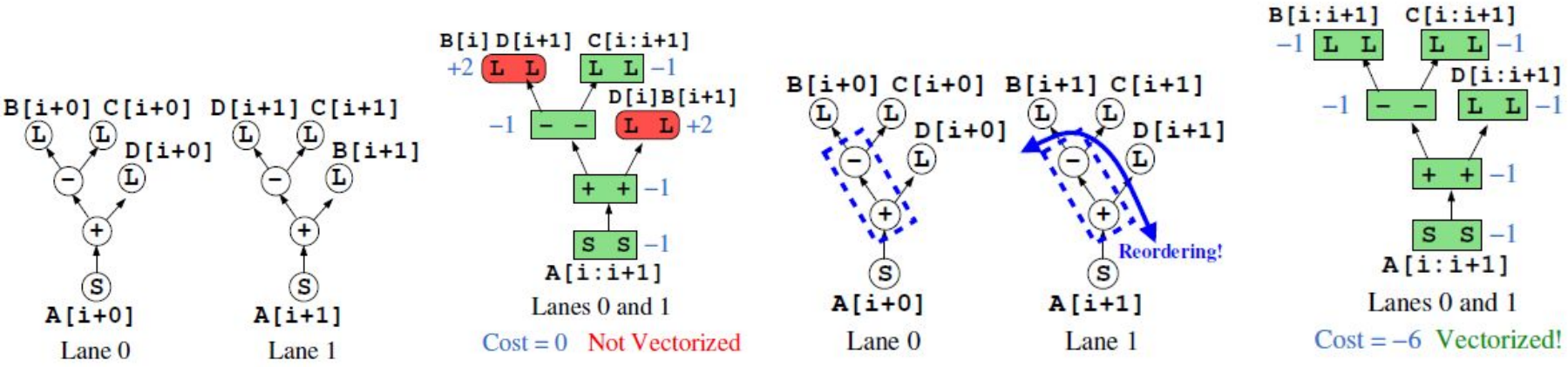
- Reordering

A + B **- C**  A **- C** + B

Super Node SLP - Leaf Node

```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=D[i+1]-C[i+1]+B[i+1];
```

Source



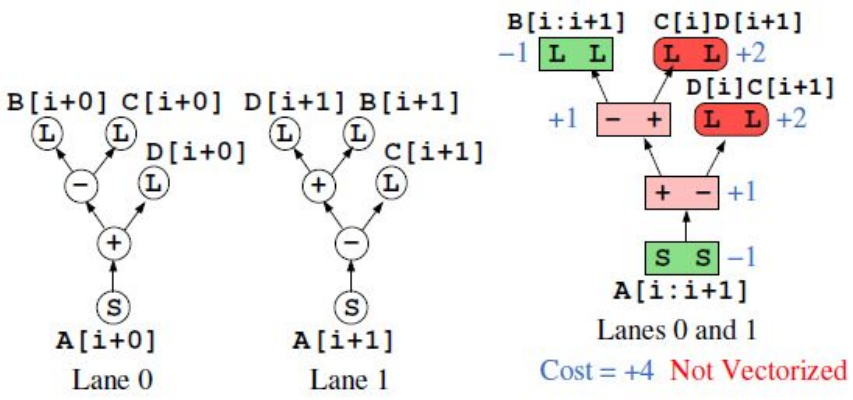
Original SLP

Super Node SLP

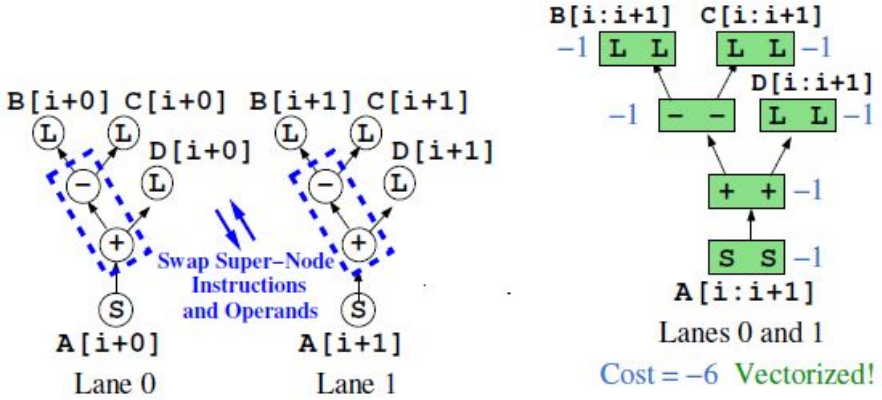
Super Node SLP - Trunk Node

```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[i+1];
```

Source

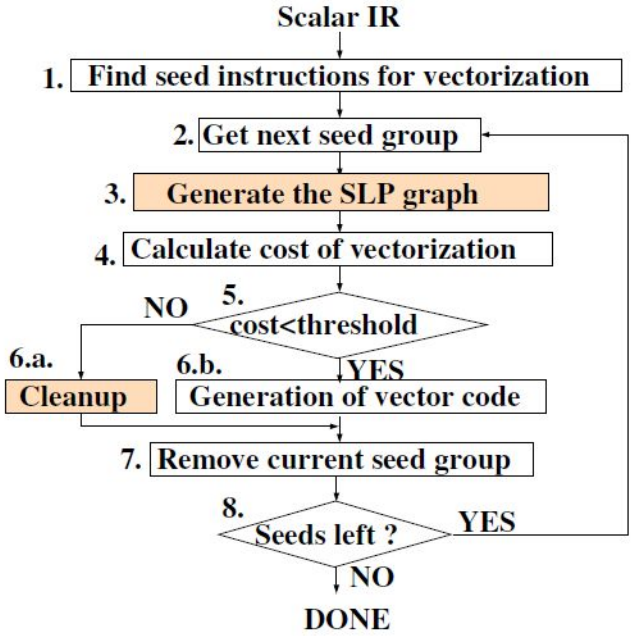


Original SLP



Super Node SLP

Super Node SLP - Algorithm



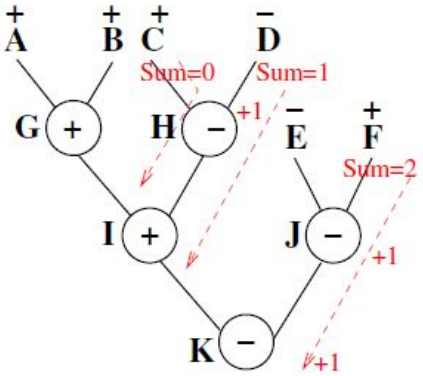
Two Key Steps:

- Construction of Super-Node

- Reordering

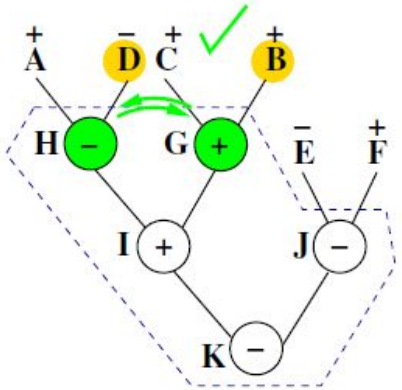
Problem:
How to determine which reordering is legal?

Super Node SLP - Legal Motions

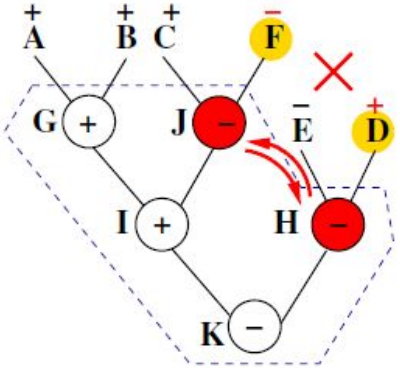


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	–
E	1	–
F	2	+
G	0	+
H	0	+
I	0	+
J	1	–
K	0	+

Calculating APO



Legal Motion



Illegal Motion

Evaluation

- Testbench: SPEC06 (*this paper also test SN-SLP's motivating examples)
- Configuration: SN-SLP, Look-ahead SLP (LSLP) and –O3

Kernel	Benchmark	Filename:Line
433-mult-su3-mat-hwvec	433.milc	m_mat_hwvec.c:23
433-mult-su3-mat-vec	433.milc	m matvec.c:64
433-mult-su3-nn	433.milc	m mat nn.c:90
453-minvers	453.provray	matcies.cpp:1331
454-solveSparseColumns	454.calculix	SubMtx solveH.c:257
454-solveDenseSubColumns	454.calculix	SubMtx solveH.c:9

Speedup on Kernels

- SN-SLP outperforms LSLP and –O3.
- On average about 1.25x speedup over O3 optimization.
- Significant speedup on motivating examples.

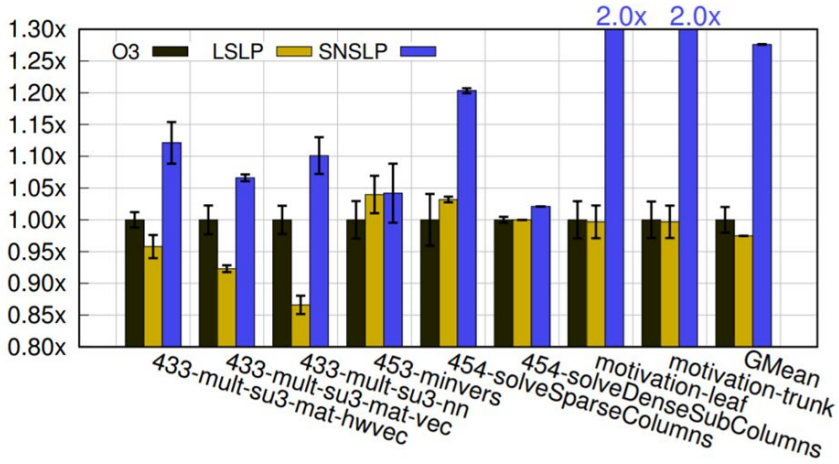


Fig. 5. Execution speedup, normalized to O3

Speedup on Benchmarks

- No significant speedup
- Only run 2% faster than LSLP in one out of 6 benchmarks.
- The kernels optimized are not hot spot

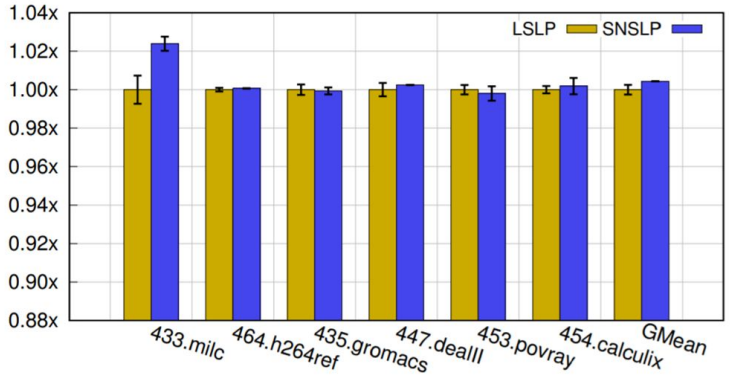


Fig. 8. Execution speedup normalized to LSLP.

Compilation Time

- No significant overhead
- Time save when there is a significant code size reduction: less work for remaining passes

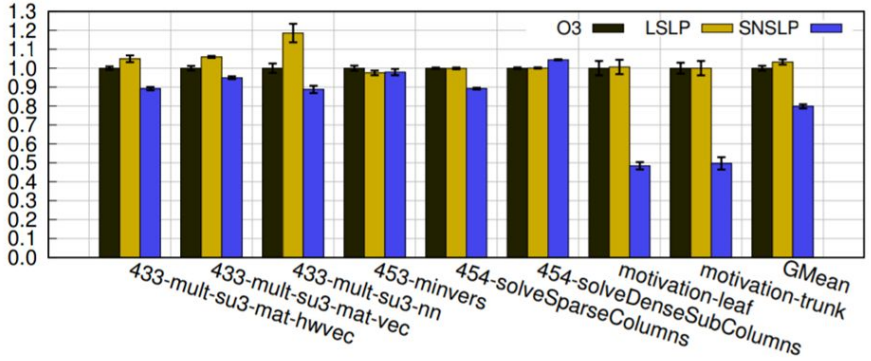


Fig. 11. Compilation time normalized to O3.

Commentary

- SNSLP brings the performance less significant benefit for the whole program.
- SNSLP grows larger SLP graphs for evaluation, but the cost model makes it still hard to trigger.
- Optimization on underlining architecture for SIMD may result in more significant results.

Q&A

[1] V. Porpodas, R. C. O. Rocha, E. Brevnov, L. F. W. Góes and T. Mattson, "Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements," *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 206-216, doi: 10.1109/CGO.2019.8661192.

[2] I. Rosen, D. Nuzman, and A. Zaks, “Loop-aware SLP in GCC,” in GCC Developers Summit, 2007