

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

Vasileios Porpodas¹, Rodrigo C. O. Rocha²
Evgueni Brevnov¹, Luís F. W. Góes³
and Timothy Mattson¹

Intel¹, University of Edinburgh², PUC Minas³

CGO 2019



THE UNIVERSITY of EDINBURGH
informatics





Legal Disclaimer & Optimization Notice

Performance results are based on testing as of 10/16/2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright 2018, Intel Corporation. All rights reserved. Intel, the Intel logo, Pentium, Xeon, Core, VTune, OpenVINO, Cilk, are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

Optimization Notice

Intels compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804



SLP: The Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen et al. PLDI'00]

SLP: The Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen et al. PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP [Rosen et al. GCC-DEV'07]



SLP: The Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen et al. PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP [Rosen et al. GCC-DEV'07]
- SLP and loop-vectorizer complement each other:

SLP: The Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen et al. PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP [Rosen et al. GCC-DEV'07]
- SLP and loop-vectorizer complement each other:
 - Unroll loop and vectorize with SLP
 - Even if loop-vectorizer fails, SLP could partly succeed



SLP: The Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen et al. PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP [Rosen et al. GCC-DEV'07]
- SLP and loop-vectorizer complement each other:
 - Unroll loop and vectorize with SLP
 - Even if loop-vectorizer fails, SLP could partly succeed
- Run SLP after the Loop Vectorizer



SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations



SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations

```
for (i=0; i<N; i+=4)
  A[i] = B[i]
  A[i+1] = B[i+1]
  A[i+2] = B[i+2]
  A[i+3] = B[i+3]
```

SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations



Loop Vectorization (LV) with VF = 4
for (i=0; i<N; i+=16)

```
for (i=0; i<N; i+=4)
  A[i] = B[i]
  A[i+1] = B[i+1]
  A[i+2] = B[i+2]
  A[i+3] = B[i+3]
```

SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations




Loop Vectorization (LV) with VF = 4
for (i=0; i<N; i+=16)

```
for (i=0; i<N; i+=4)
  A[i] = B[i]
  A[i+1] = B[i+1]
  A[i+2] = B[i+2]
  A[i+3] = B[i+3]
```

SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations

 **Loop Vectorization (LV) with VF = 4**

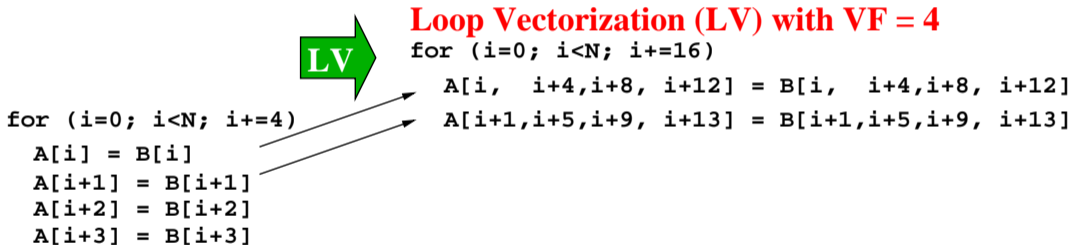
```
for (i=0; i<N; i+=4)
  A[i] = B[i]
  A[i+1] = B[i+1]
  A[i+2] = B[i+2]
  A[i+3] = B[i+3]
```

for (i=0; i<N; i+=16)
 A[i, i+4, i+8, i+12] = B[i, i+4, i+8, i+12]

An arrow points from the first four lines of the first code block to the single line of the second code block.

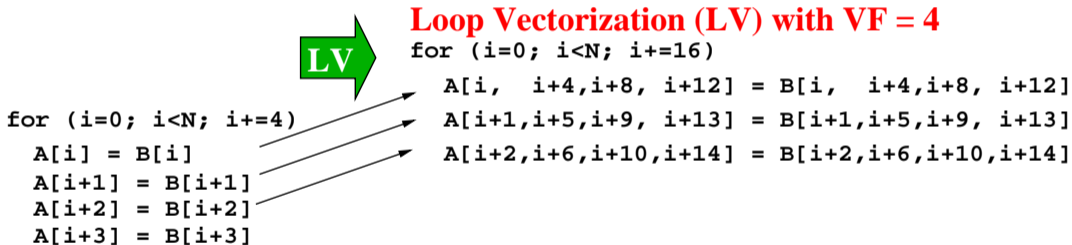
SLP compared to Loop Vectorization

- Vectorizes across instructions, *NOT* iterations



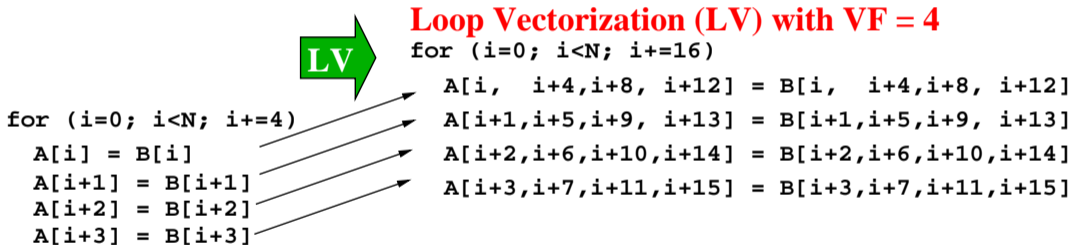
SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations



SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations



SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations

```
for (i=0; i<N; i+=4)
  A[i] = B[i]
  A[i+1] = B[i+1]
  A[i+2] = B[i+2]
  A[i+3] = B[i+3]
```



Loop Vectorization (LV) with VF = 4

```
for (i=0; i<N; i+=16)
```

```
  A[i, i+4,i+8, i+12] = B[i, i+4,i+8, i+12]
```

```
  A[i+1,i+5,i+9, i+13] = B[i+1,i+5,i+9, i+13]
```

```
  A[i+2,i+6,i+10,i+14] = B[i+2,i+6,i+10,i+14]
```

```
  A[i+3,i+7,i+11,i+15] = B[i+3,i+7,i+11,i+15]
```



SLP Vectorizer with VF = 4

```
for (i=0; i<N; i+=4)
```


SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations

```
for (i=0; i<N; i+=4)
  A[i] = B[i]
  A[i+1] = B[i+1]
  A[i+2] = B[i+2]
  A[i+3] = B[i+3]
```



Loop Vectorization (LV) with VF = 4

```
for (i=0; i<N; i+=16)
```

```
  A[i, i+4,i+8, i+12] = B[i, i+4,i+8, i+12]
```

```
  A[i+1,i+5,i+9, i+13] = B[i+1,i+5,i+9, i+13]
```

```
  A[i+2,i+6,i+10,i+14] = B[i+2,i+6,i+10,i+14]
```

```
  A[i+3,i+7,i+11,i+15] = B[i+3,i+7,i+11,i+15]
```



SLP Vectorizer with VF = 4

```
for (i=0; i<N; i+=4)
```

SLP compared to Loop Vectorization

- Vectorizes across instructions, ***NOT*** iterations

```
for (i=0; i<N; i+=4)
  A[i] = B[i]
  A[i+1] = B[i+1]
  A[i+2] = B[i+2]
  A[i+3] = B[i+3]
```



Loop Vectorization (LV) with VF = 4

```
for (i=0; i<N; i+=16)
```

```
  A[i, i+4,i+8, i+12] = B[i, i+4,i+8, i+12]
```

```
  A[i+1,i+5,i+9, i+13] = B[i+1,i+5,i+9, i+13]
```

```
  A[i+2,i+6,i+10,i+14] = B[i+2,i+6,i+10,i+14]
```

```
  A[i+3,i+7,i+11,i+15] = B[i+3,i+7,i+11,i+15]
```



SLP Vectorizer with VF = 4

```
for (i=0; i<N; i+=4)
```

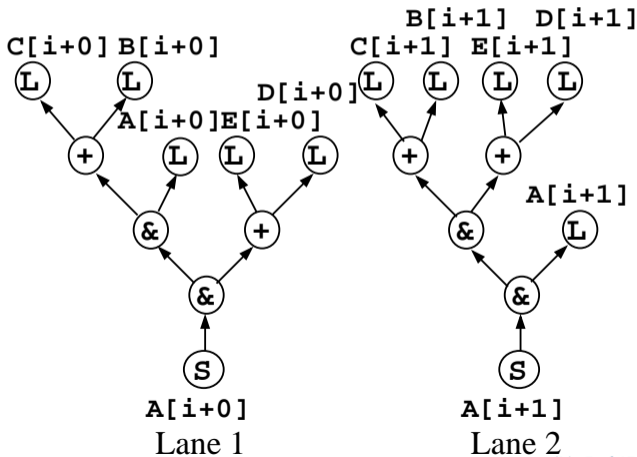
```
  A[i:i+3] = B[i:i+3]
```

State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic

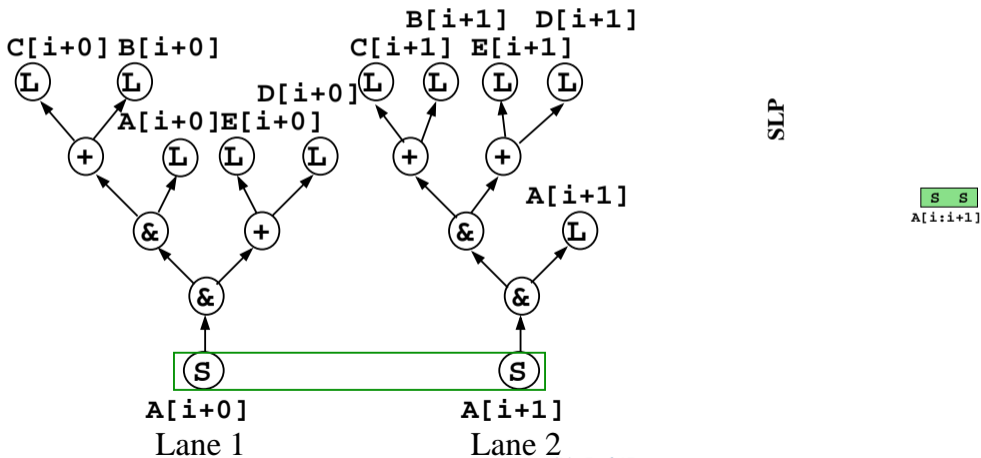
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



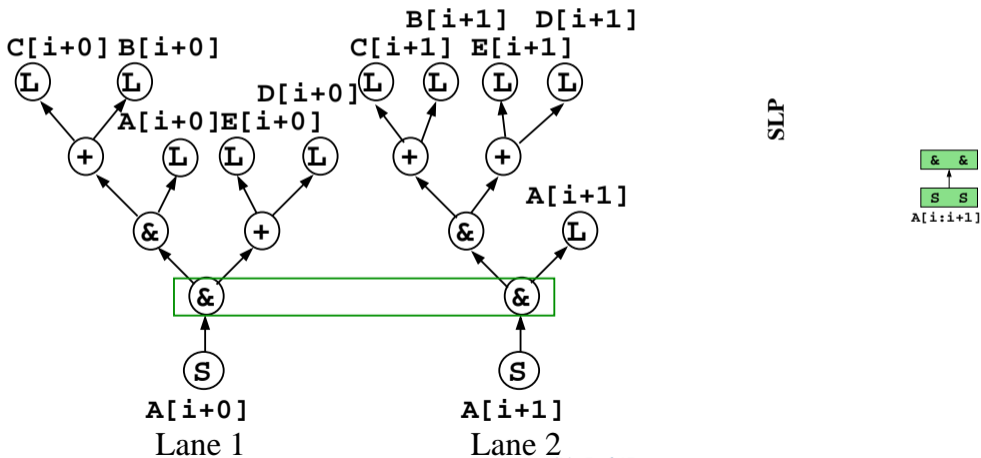
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



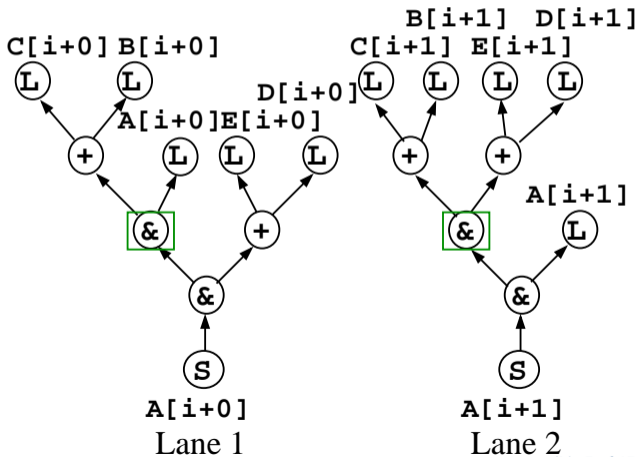
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic

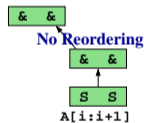


State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic

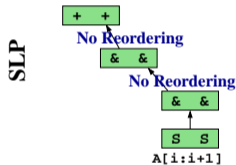
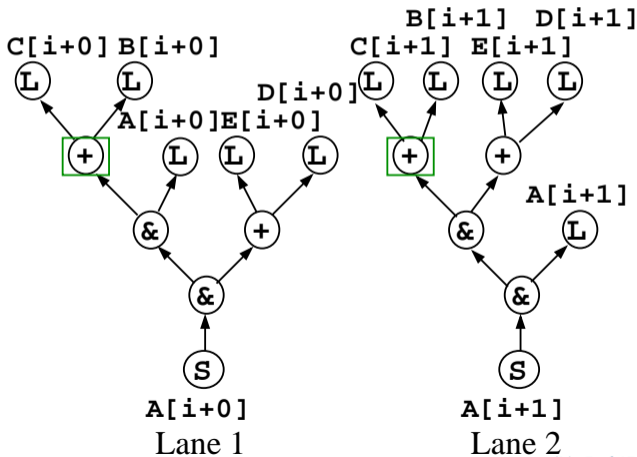


SLP



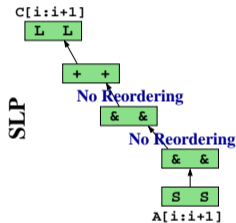
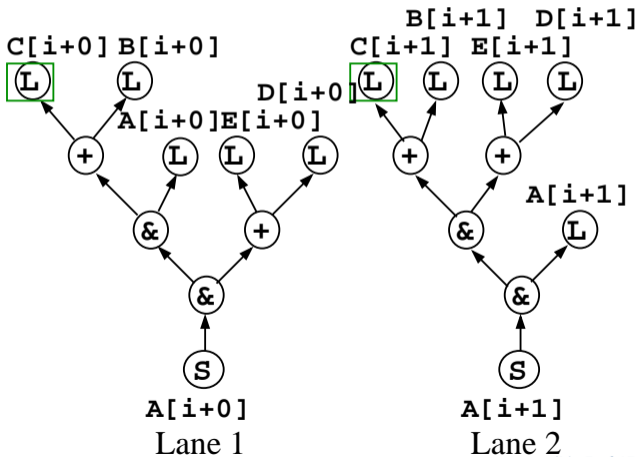
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



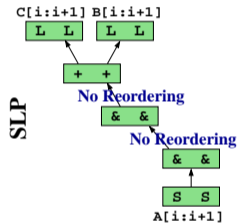
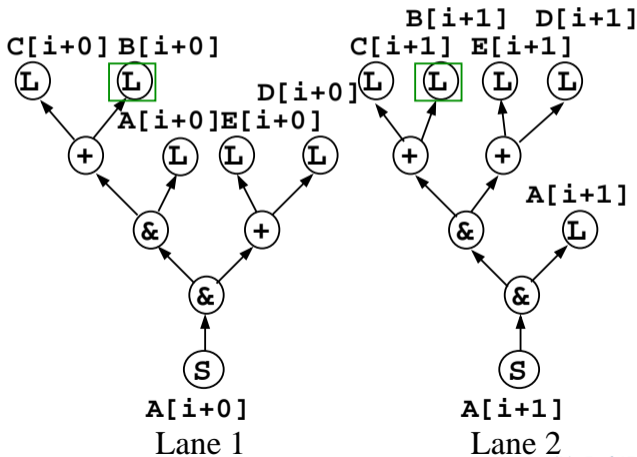
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



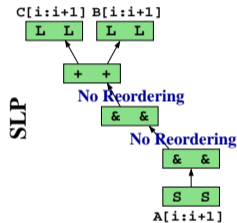
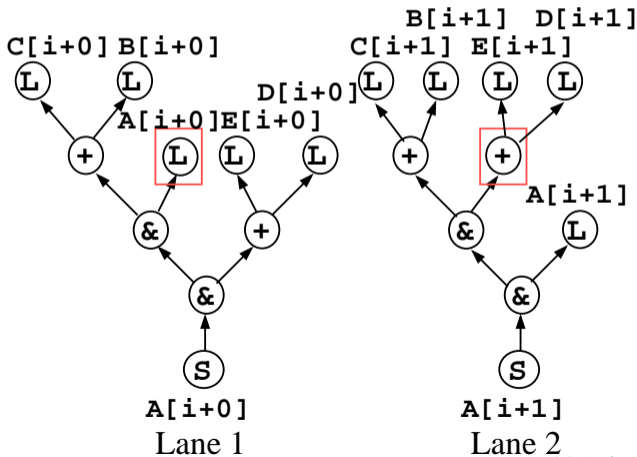
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



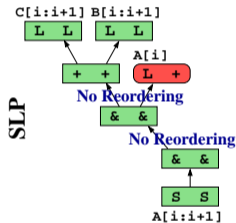
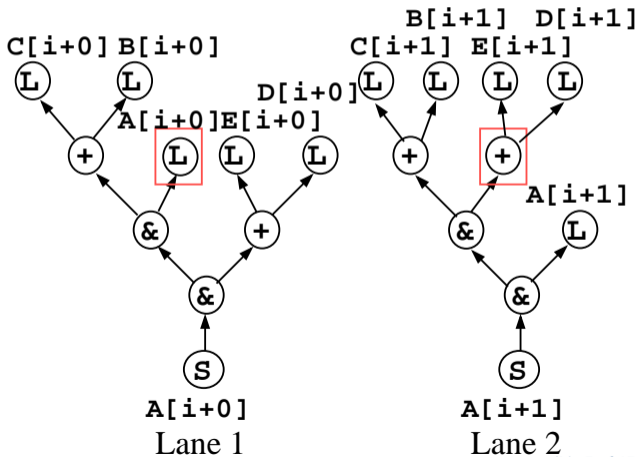
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



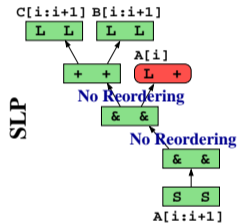
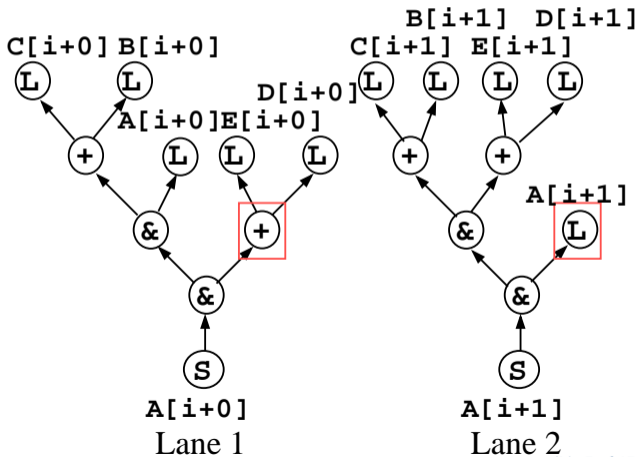
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



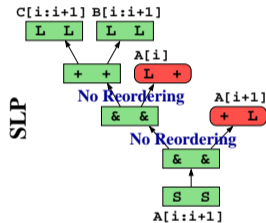
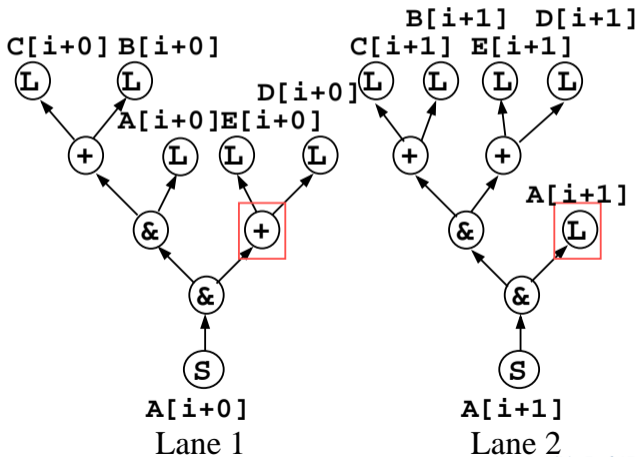
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



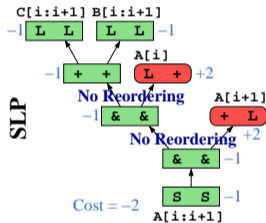
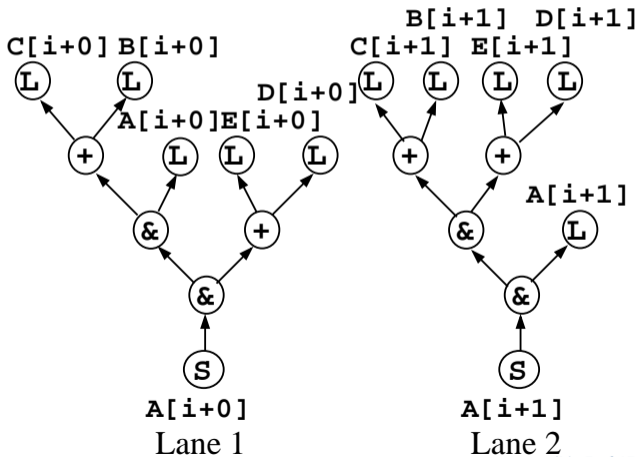
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



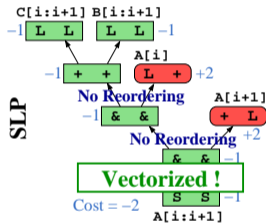
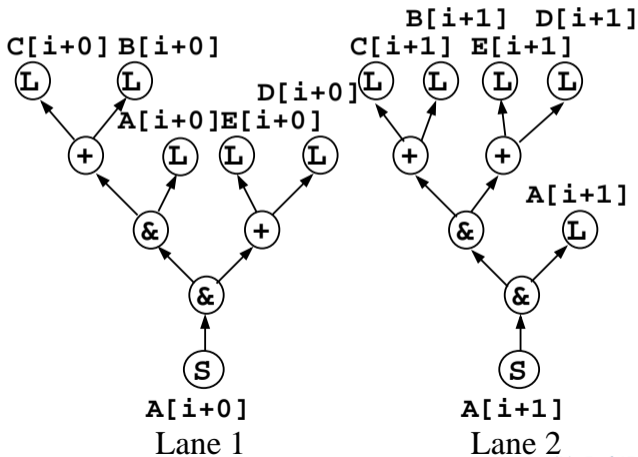
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



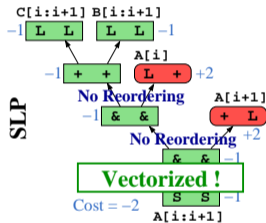
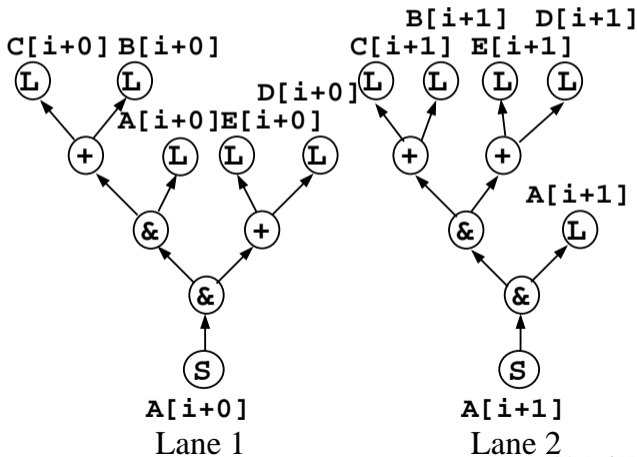
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



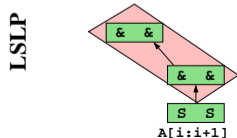
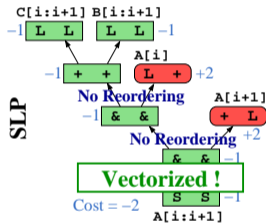
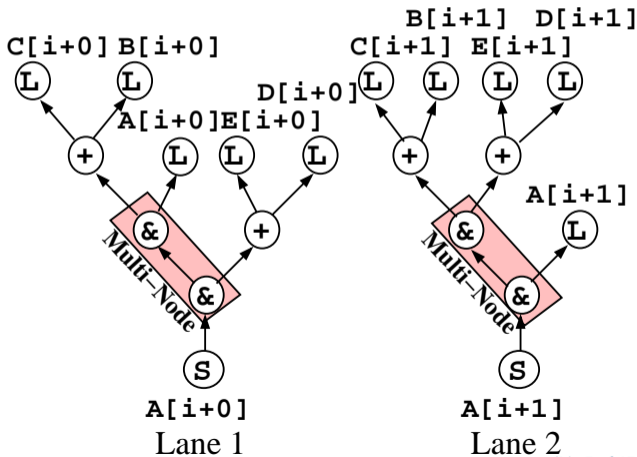
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



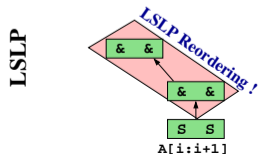
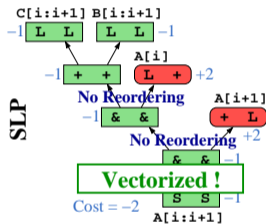
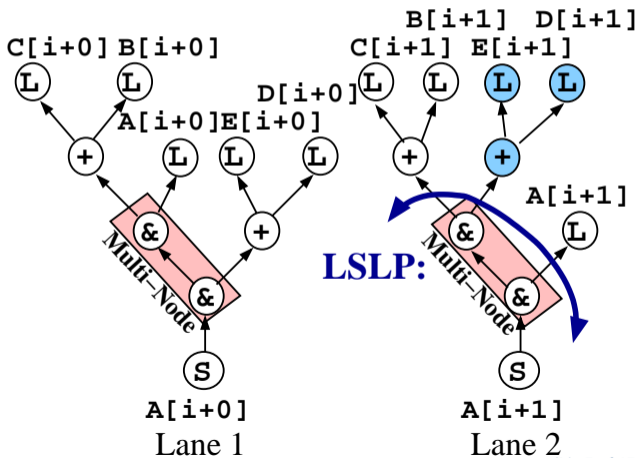
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



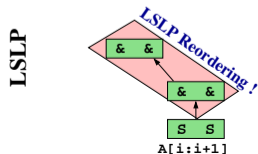
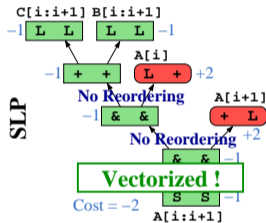
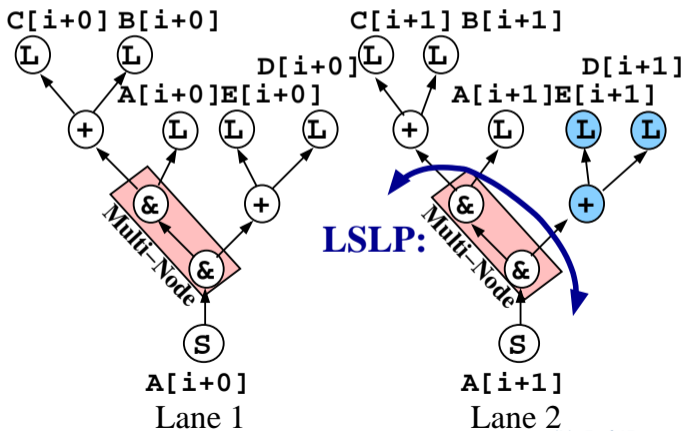
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



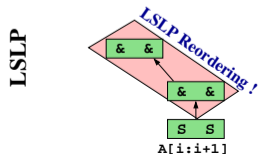
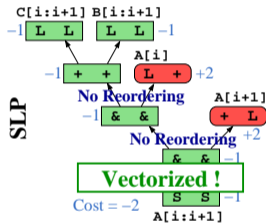
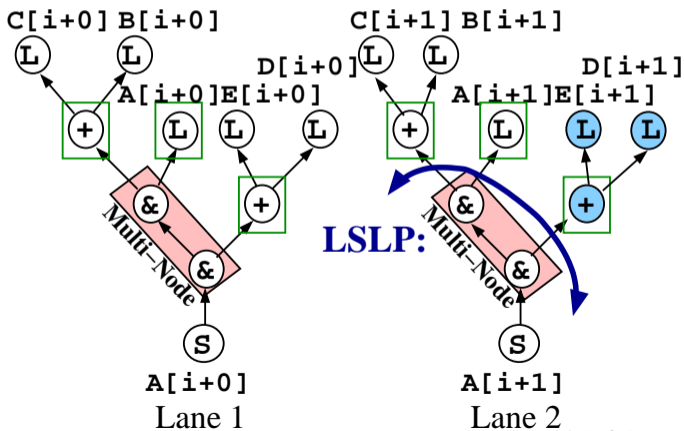
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



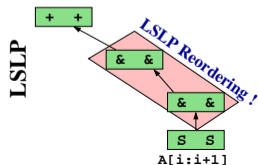
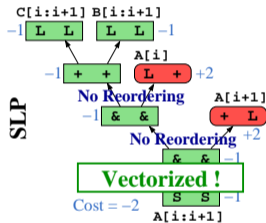
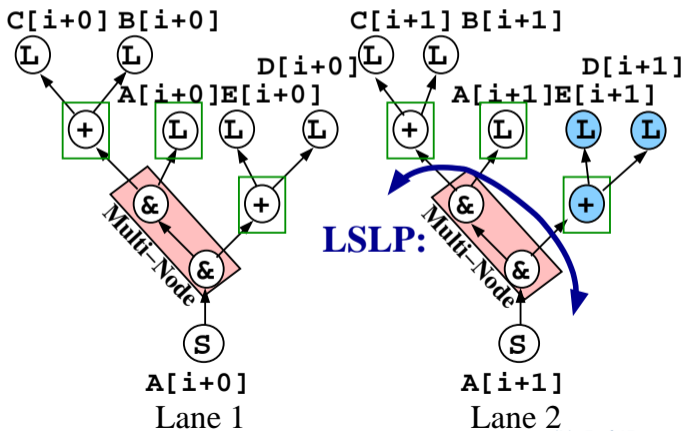
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



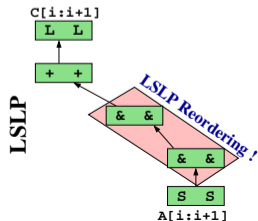
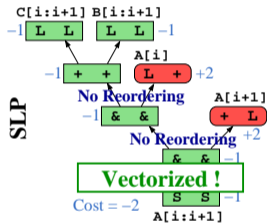
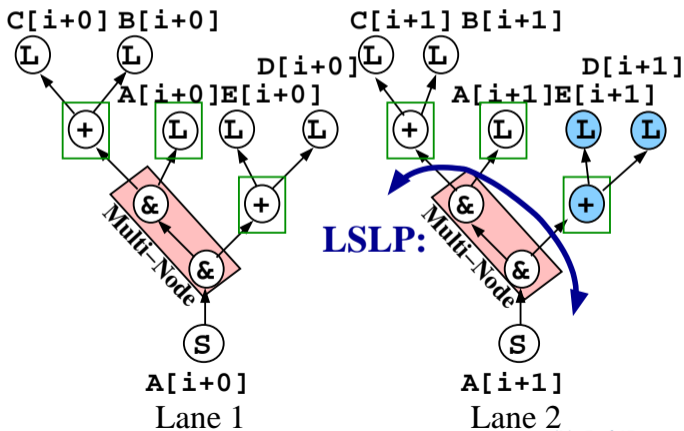
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



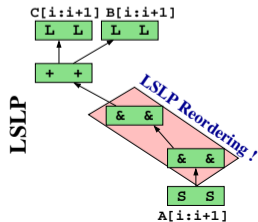
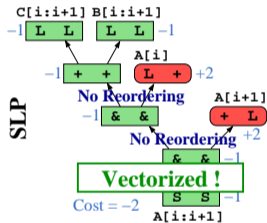
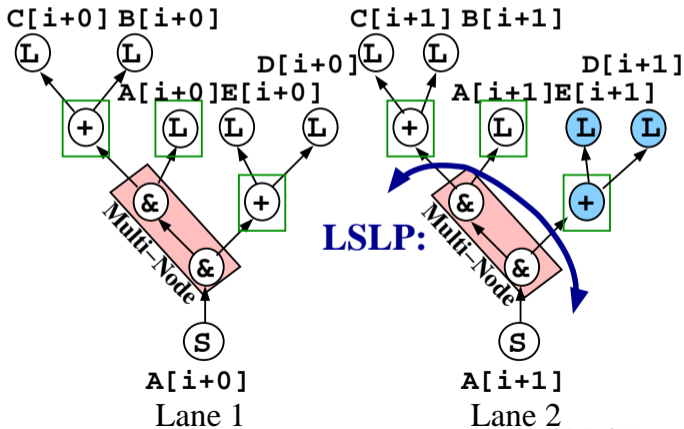
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



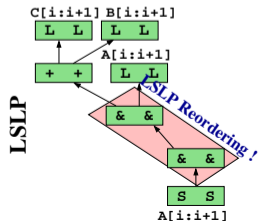
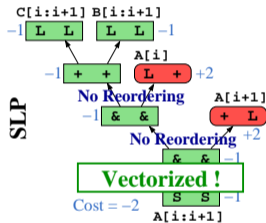
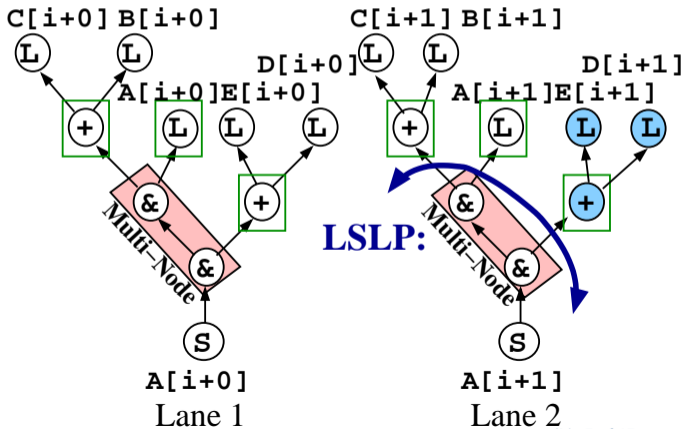
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



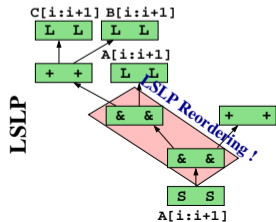
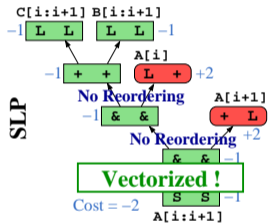
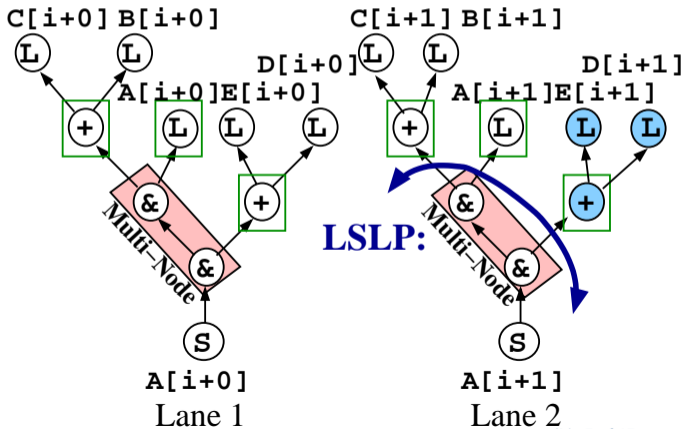
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



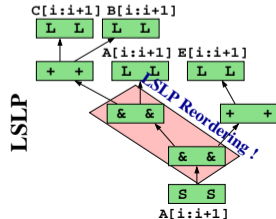
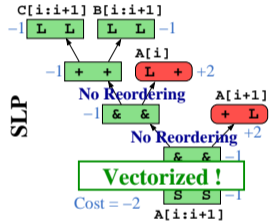
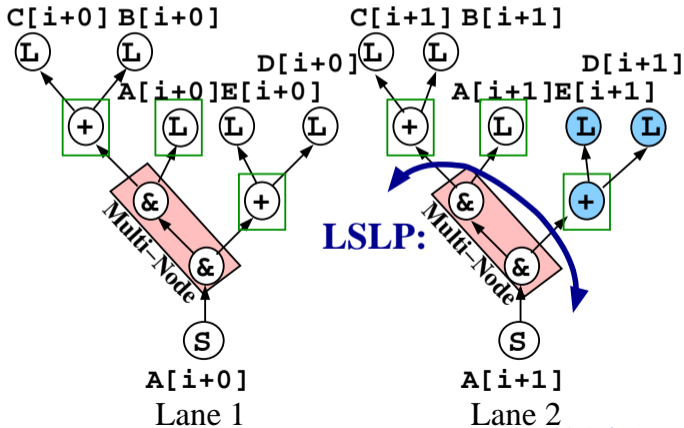
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



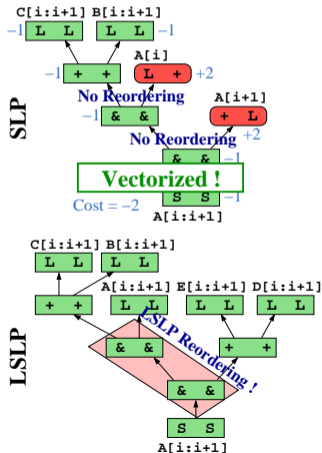
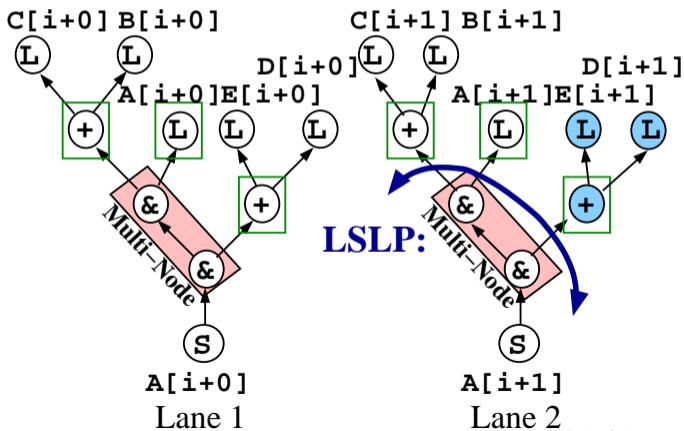
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



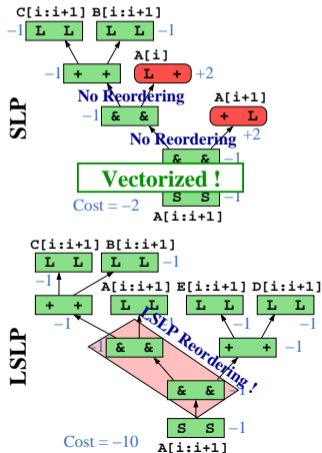
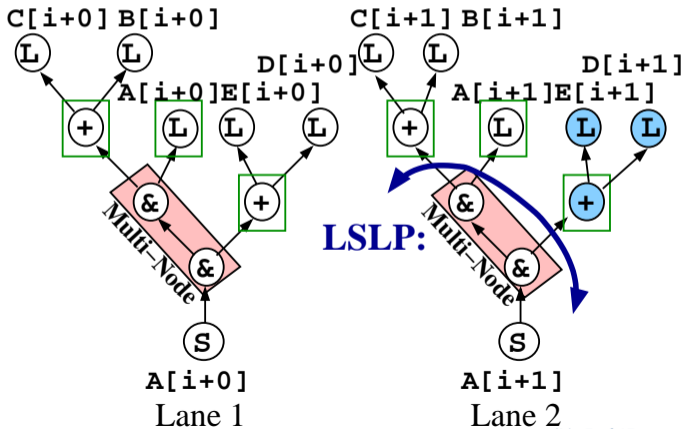
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



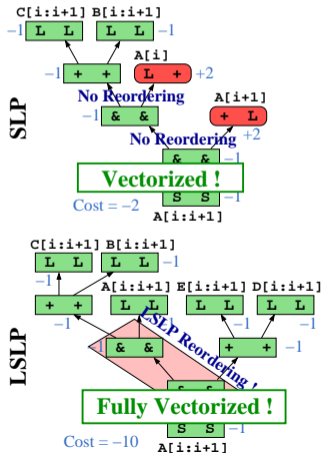
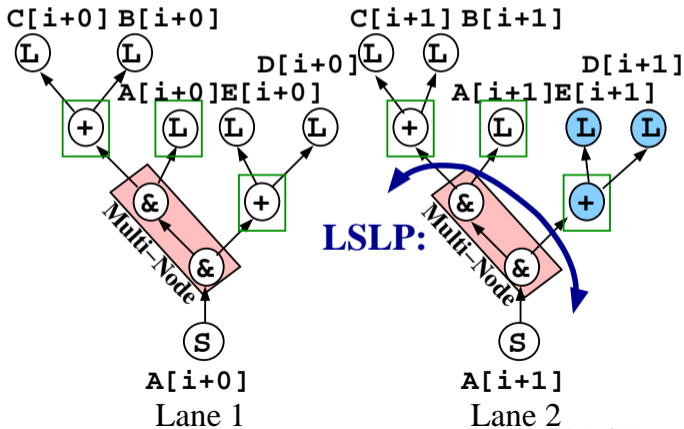
State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



State-of-the-art [Look-Ahead SLP CGO'18]

- Form Multi-Nodes and reorder operands with Look-Ahead heuristic



Multi-Node (LSLP) VS Super-Node (SuperNode-SLP)

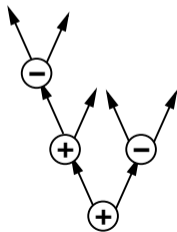
- The inverse element of $\text{ADD}(+)$ is $\text{SUB}(-)$

Multi-Node (LSLP) VS Super-Node (SuperNode-SLP)

- The inverse element of $ADD(+)$ is $SUB(-)$
- Multi-Nodes cannot handle inverse elements

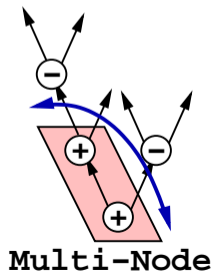
Multi-Node (LSLP) VS Super-Node (SuperNode-SLP)

- The inverse element of $ADD(+)$ is $SUB(-)$
- Multi-Nodes cannot handle inverse elements



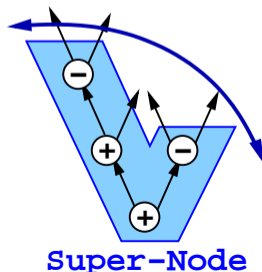
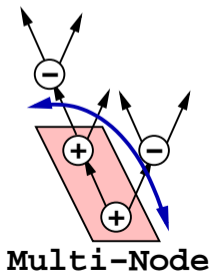
Multi-Node (LSLP) VS Super-Node (SuperNode-SLP)

- The inverse element of $ADD(+)$ is $SUB(-)$
- Multi-Nodes cannot handle inverse elements



Multi-Node (LSLP) VS Super-Node (SuperNode-SLP)

- The inverse element of $ADD(+)$ is $SUB(-)$
- Multi-Nodes cannot handle inverse elements
- Super-Nodes **can** reorder across them when legal

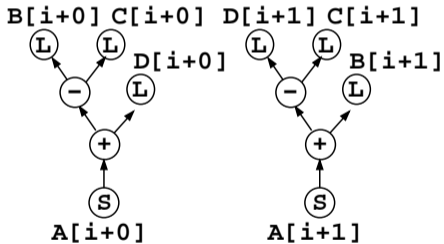


LSLP fails in the presence of inverse elements

```
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];
```

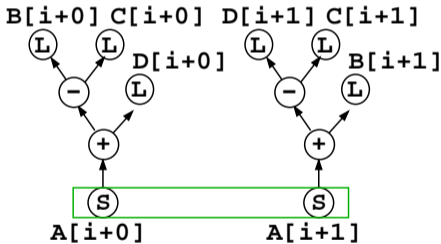
LSLP fails in the presence of inverse elements

```
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];
```



LSLP fails in the presence of inverse elements

```
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];
```



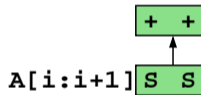
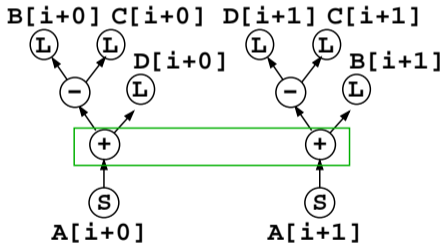
(L)SLP

A[i:i+1] S S

LSLP fails in the presence of inverse elements

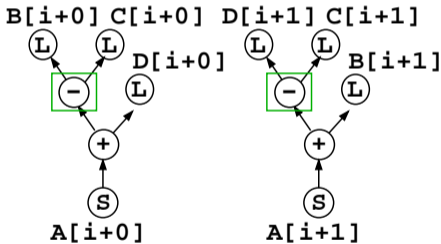
```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=D[i+1]-C[i+1]+B[i+1];
```

(L)SLP

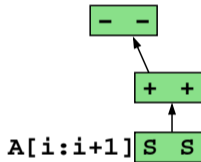


LSLP fails in the presence of inverse elements

```
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];
```

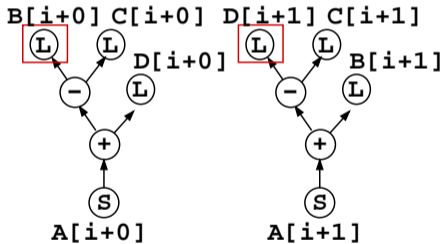


(L) SLP

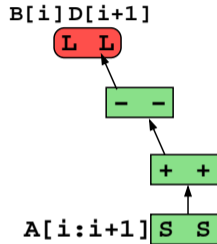


LSLP fails in the presence of inverse elements

```
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];
```

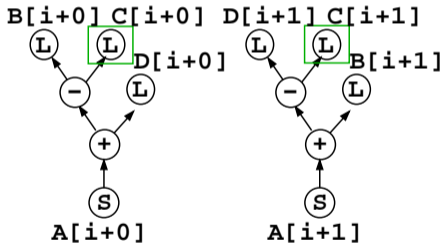


(L) SLP

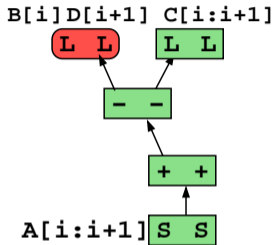


LSLP fails in the presence of inverse elements

```
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];
```

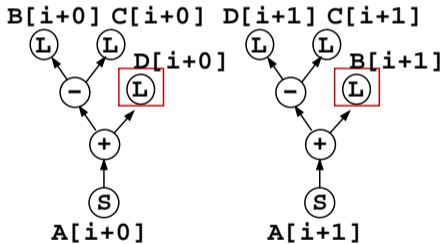


(L)SLP

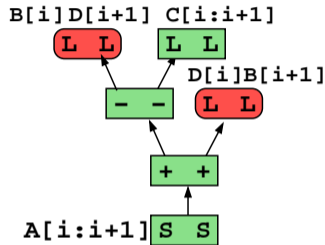


LSLP fails in the presence of inverse elements

```
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];
```

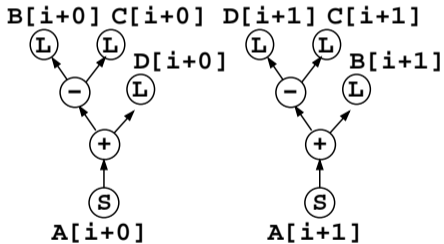


(L)SLP

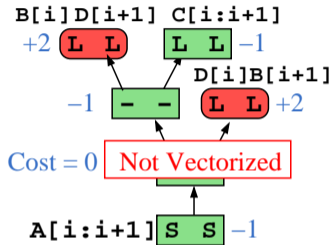


LSLP fails in the presence of inverse elements

```
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];
```

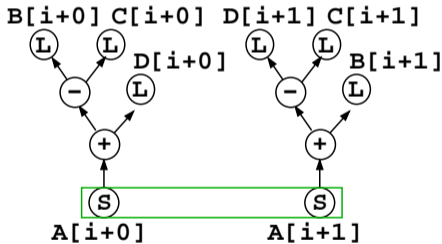


(L)SLP



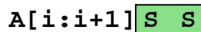
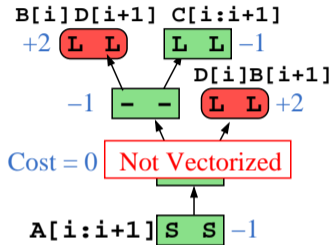
LSLP fails in the presence of inverse elements

```
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];
```



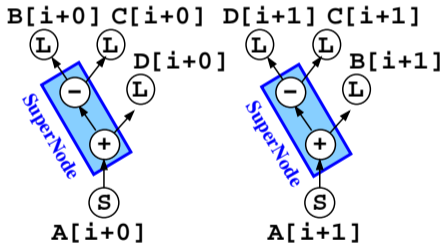
(L) SLP

SN-SLP



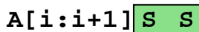
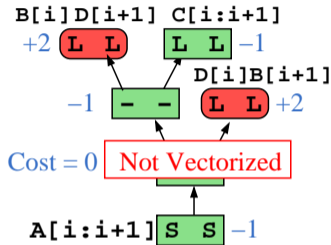
LSLP fails in the presence of inverse elements

```
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];
```



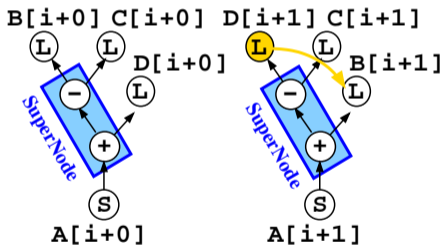
(L) SLP

SN-SLP



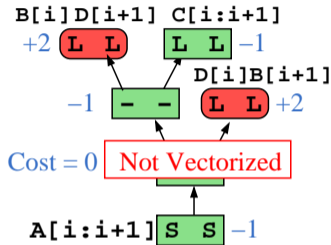
LSLP fails in the presence of inverse elements

```
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];
```



(L) SLP

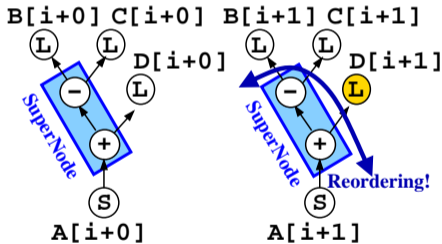
SN-SLP



A[i:i+1] S S

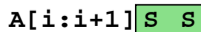
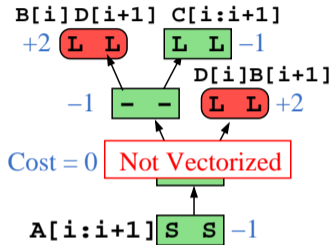
LSLP fails in the presence of inverse elements

```
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];
```



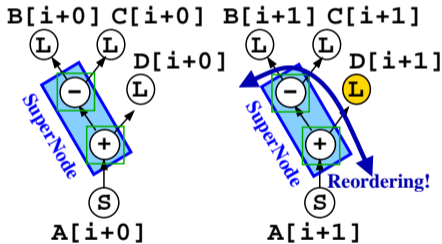
(L) SLP

SN-SLP

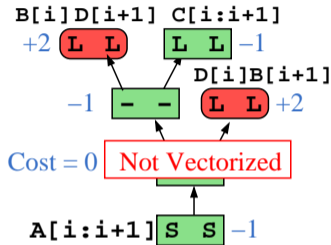


LSLP fails in the presence of inverse elements

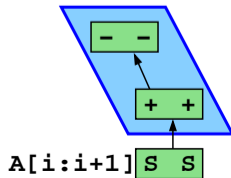
```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=D[i+1]-C[i+1]+B[i+1];
```



(L) SLP

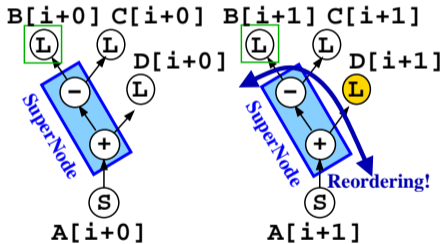


SN-SLP



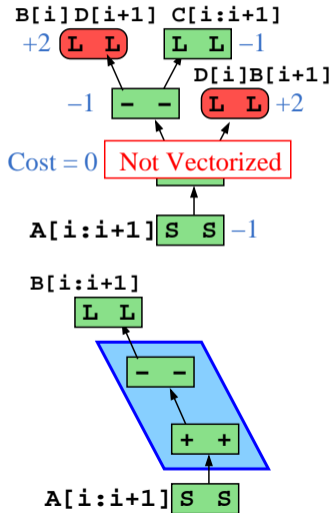
LSLP fails in the presence of inverse elements

```
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];
```



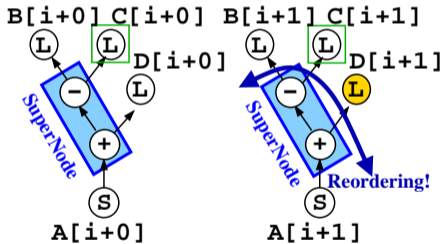
(L) SLP

SN-SLP



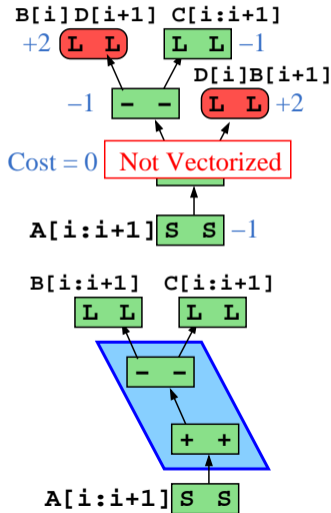
LSLP fails in the presence of inverse elements

```
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];
```



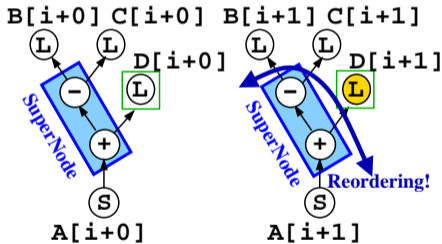
(L) SLP

SN-SLP



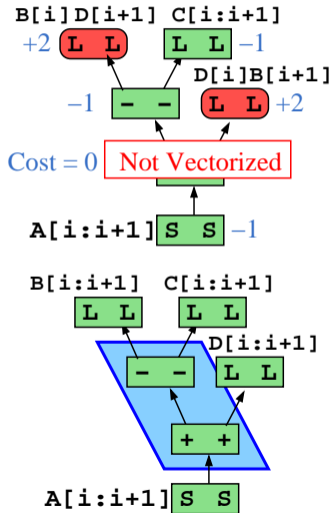
LSLP fails in the presence of inverse elements

```
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];
```



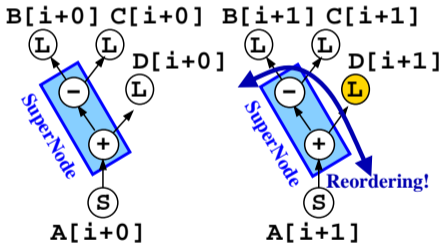
(L) SLP

SN-SLP



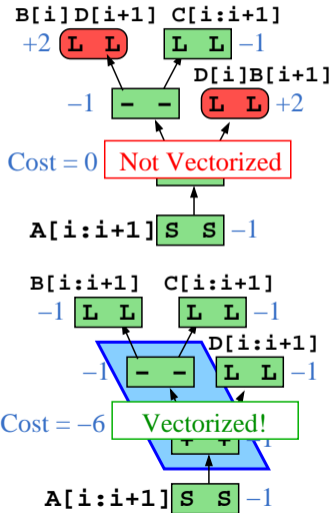
LSLP fails in the presence of inverse elements

```
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];
```



(L) SLP

SN-SLP



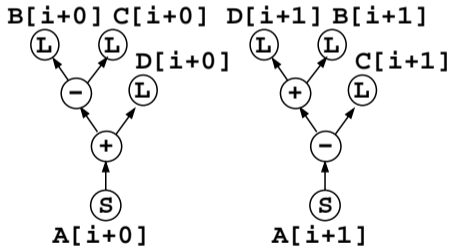


SuperNode internal nodes can be reordered too

```
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];
```

SuperNode internal nodes can be reordered too

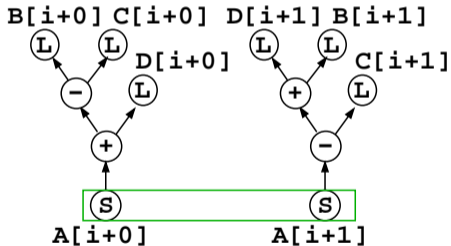
```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[i+1];
```



SuperNode internal nodes can be reordered too

```
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];
```

(L)SLP

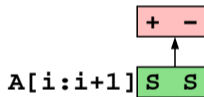
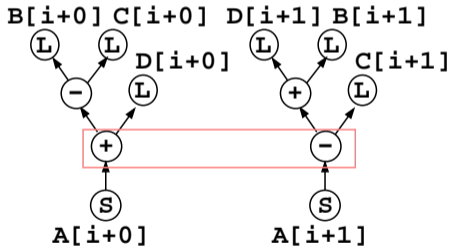


A[i:i+1] S S

SuperNode internal nodes can be reordered too

```
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];
```

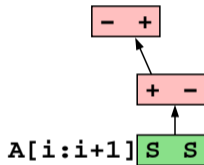
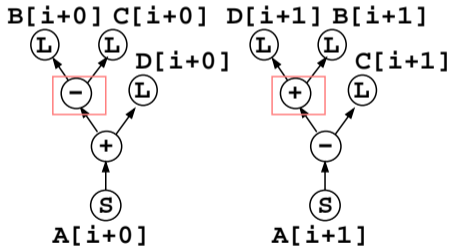
(L)SLP



SuperNode internal nodes can be reordered too

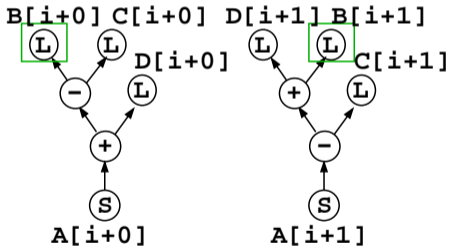
```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[i+1];
```

(L)SLP

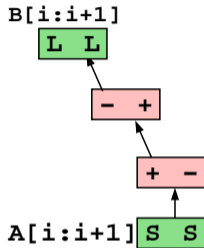


SuperNode internal nodes can be reordered too

```
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];
```



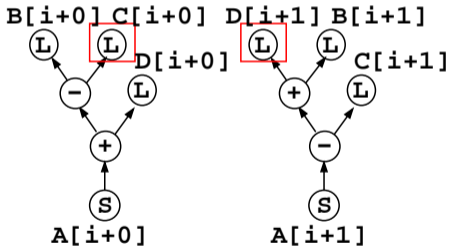
(L)SLP



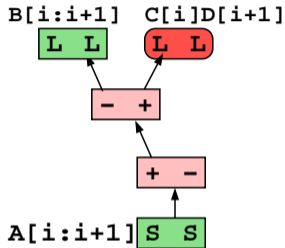
SuperNode internal nodes can be reordered too

```

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];
    
```

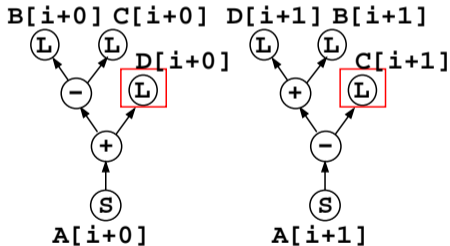


(L)SLP

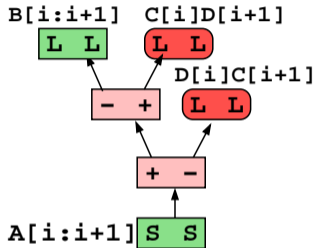


SuperNode internal nodes can be reordered too

```
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];
```

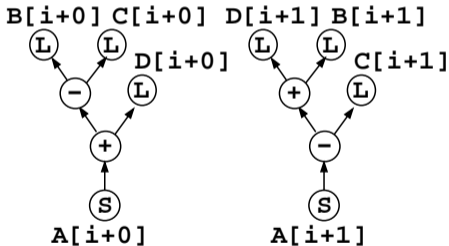


(L) SLP

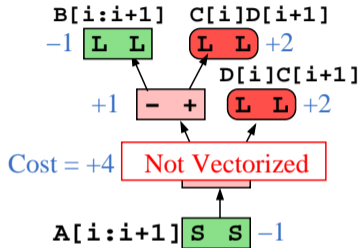


SuperNode internal nodes can be reordered too

```
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];
```

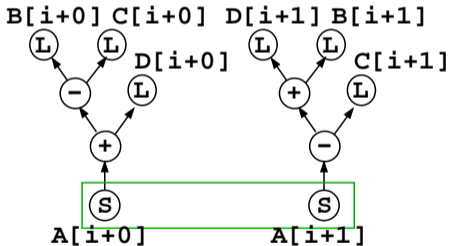


(L) SLP



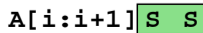
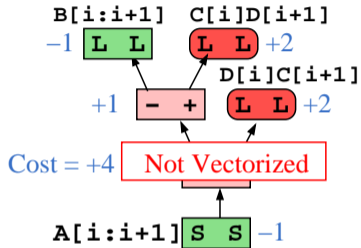
SuperNode internal nodes can be reordered too

```
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];
```



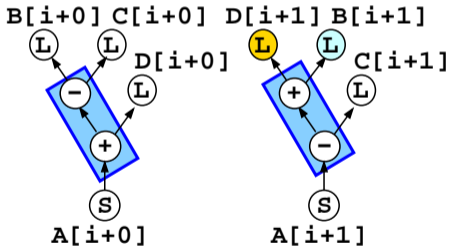
(L) SLP

SN-SLP



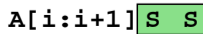
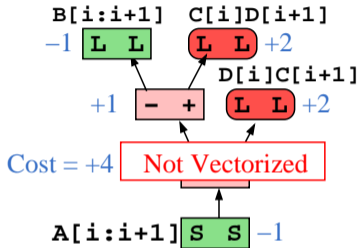
SuperNode internal nodes can be reordered too

```
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];
```



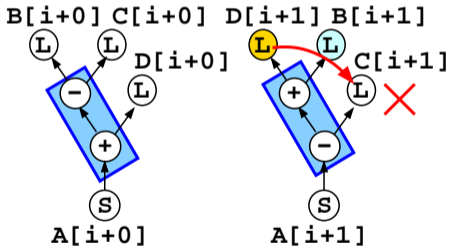
(L) SLP

SN-SLP



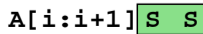
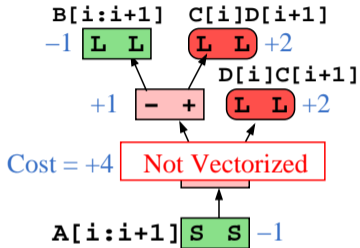
SuperNode internal nodes can be reordered too

```
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];
```



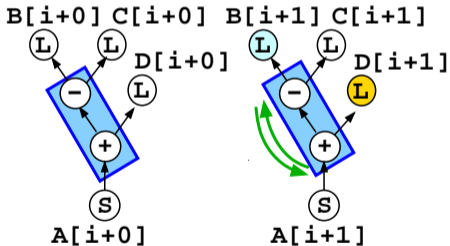
(L) SLP

SN-SLP



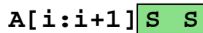
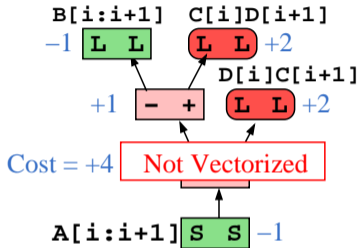
SuperNode internal nodes can be reordered too

```
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];
```



(L) SLP

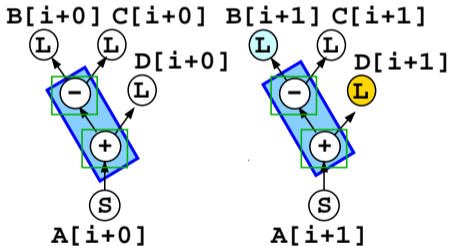
SN-SLP



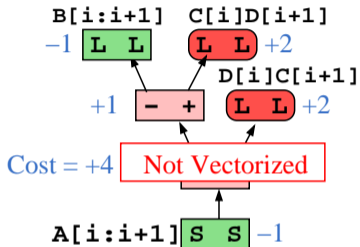
SuperNode internal nodes can be reordered too

```

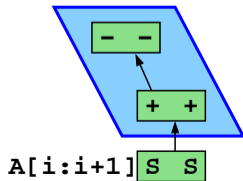
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[i+1];
    
```



(L) SLP

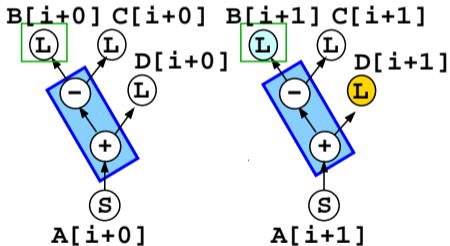


SN-SLP

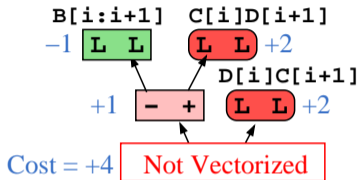


SuperNode internal nodes can be reordered too

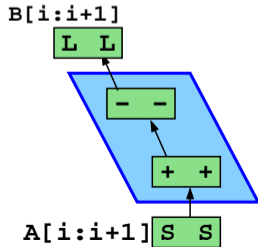
```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[i+1];
```



(L) SLP

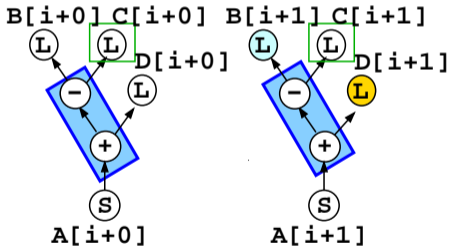


SN-SLP

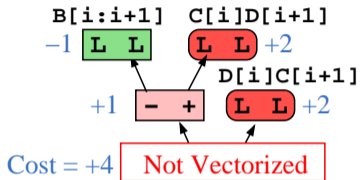


SuperNode internal nodes can be reordered too

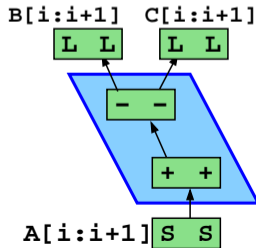
```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[i+1];
```



(L) SLP

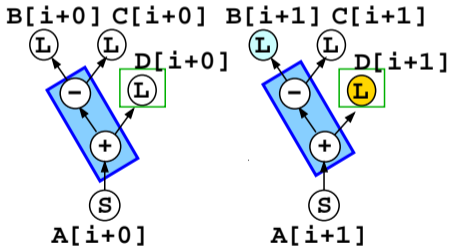


SN-SLP

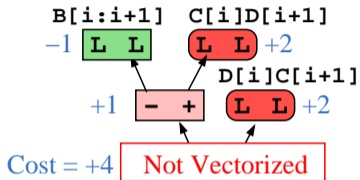


SuperNode internal nodes can be reordered too

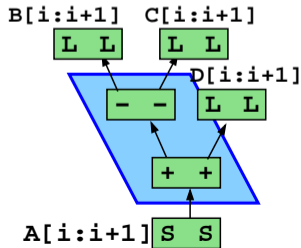
```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[i+1];
```



(L) SLP

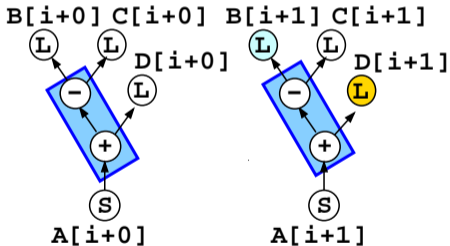


SN-SLP

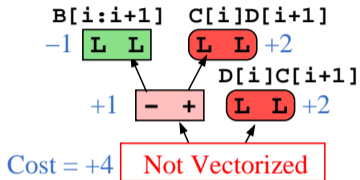


SuperNode internal nodes can be reordered too

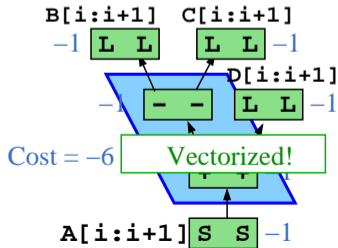
```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[i+1];
```



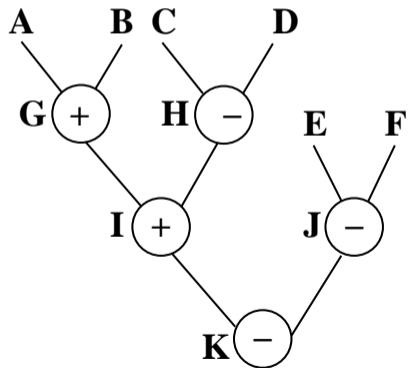
(L) SLP



SN-SLP

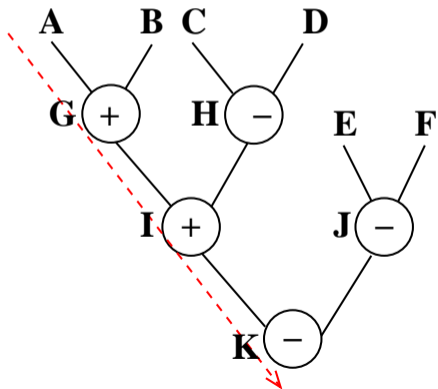


Legality



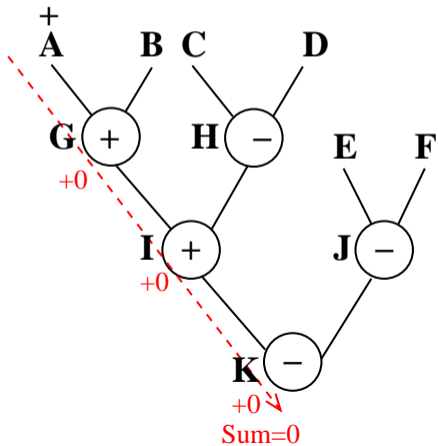
	Accumulated Path Operation (APO)	Linearized

Legality



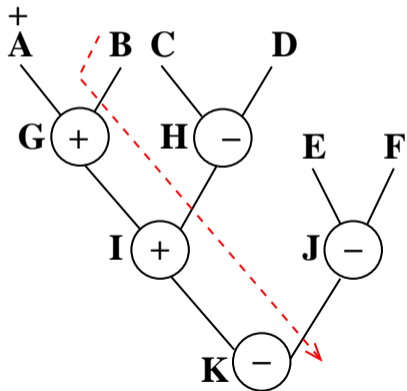
	Accumulated Path Operation (APO)	Linearized

Legality



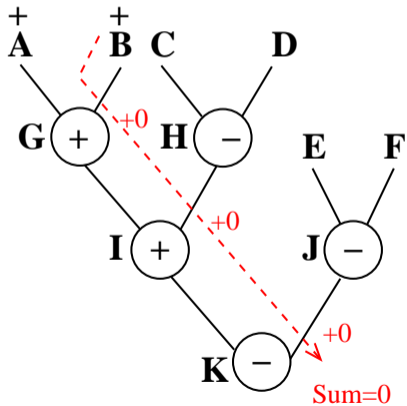
	Sum	Accumulated Path Operation (APO)	Linearized
A	0	+	

Legality



	Sum	Accumulated Path Operation (APO)	Linearized
A	0	+	$\begin{matrix} & \swarrow & \text{A} \\ + & & \end{matrix}$

Legality

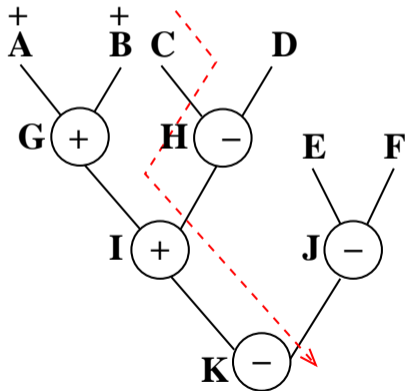


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+

Linearized



Legality

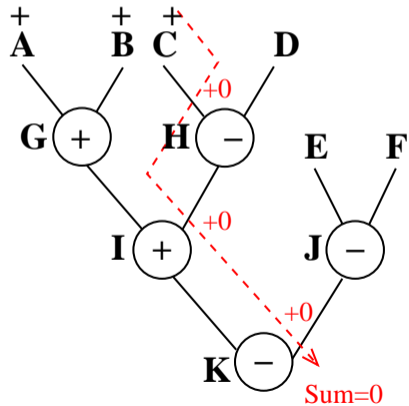


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+

Linearized

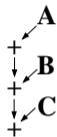


Legality

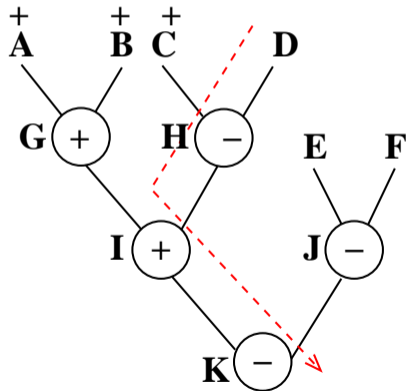


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+

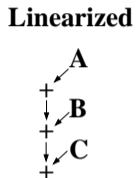
Linearized



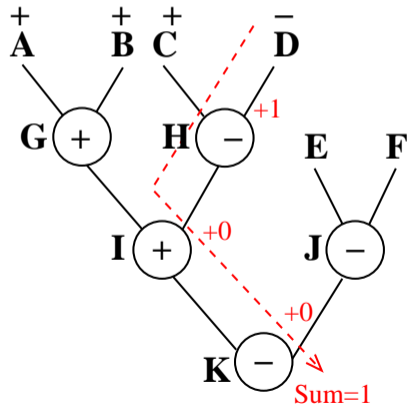
Legality



	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+

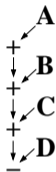


Legality

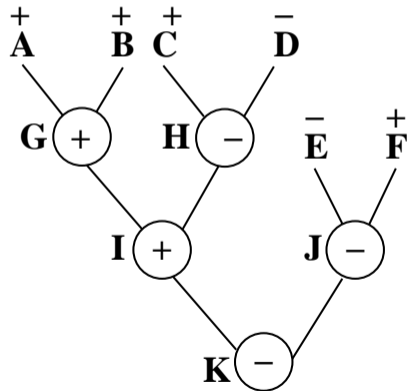


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	-

Linearized

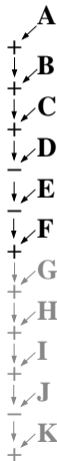


Legality

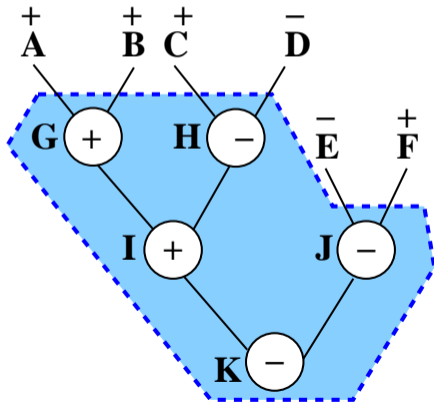


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	-
E	1	-
F	2	+
G	0	+
H	0	+
I	0	+
J	1	-
K	0	+

Linearized

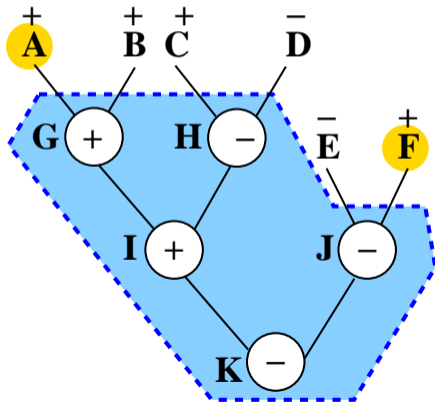


Legality



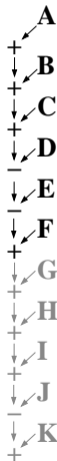
	Sum	Accumulated Path Operation (APO)	Linearized
A	0	+	A
B	0	+	+ B
C	0	+	+ + C
D	1	-	+ + + + - D
E	1	-	+ + + + - - E
F	2	+	+ + + + - - - + F
G	0	+	+ + + + - - - + + G
H	0	+	+ + + + - - - + + + H
I	0	+	+ + + + - - - + + + + I
J	1	-	+ + + + - - - + + + - - J
K	0	+	+ + + + - - - + + + - - - + K

Legality

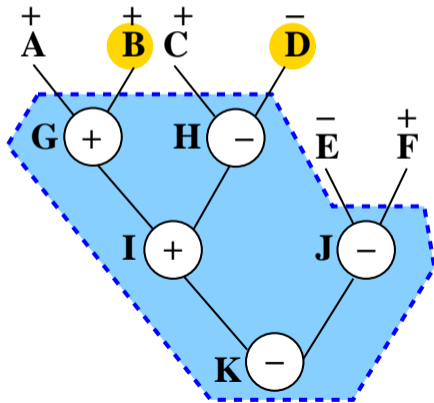


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	-
E	1	-
F	2	+
G	0	+
H	0	+
I	0	+
J	1	-
K	0	+

Linearized



Legality

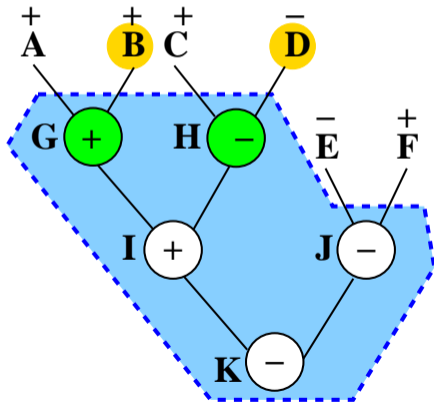


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	-
E	1	-
F	2	+
G	0	+
H	0	+
I	0	+
J	1	-
K	0	+

Linearized

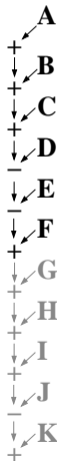


Legality

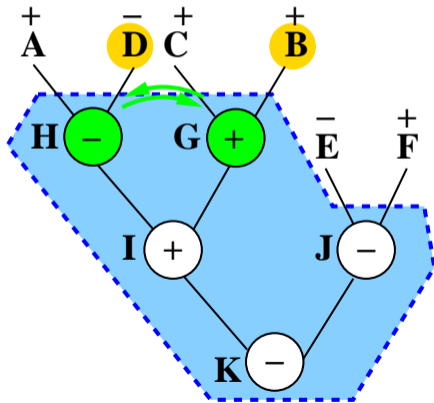


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	-
E	1	-
F	2	+
G	0	+
H	0	+
I	0	+
J	1	-
K	0	+

Linearized



Legality

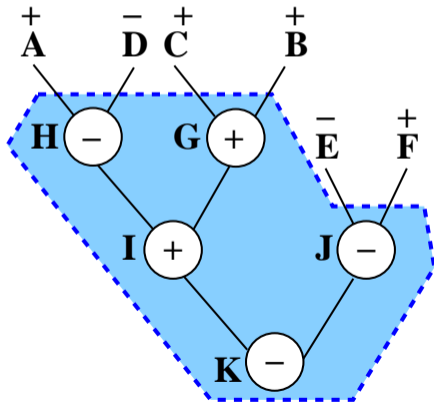


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	-
E	1	-
F	2	+
G	0	+
H	0	+
I	0	+
J	1	-
K	0	+

Linearized

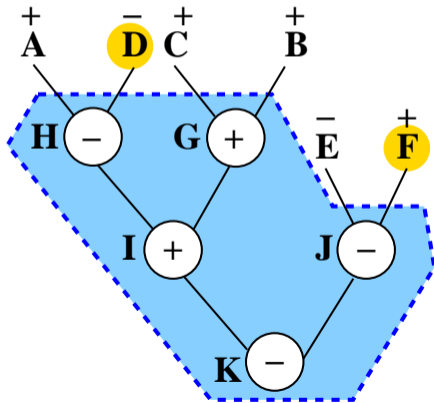


Legality



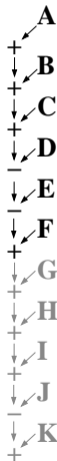
	Sum	Accumulated Path Operation (APO)	Linearized
A	0	+	A
B	0	+	B
C	0	+	C
D	1	-	D
E	1	-	E
F	2	+	F
G	0	+	G
H	0	+	H
I	0	+	I
J	1	-	J
K	0	+	K

Legality

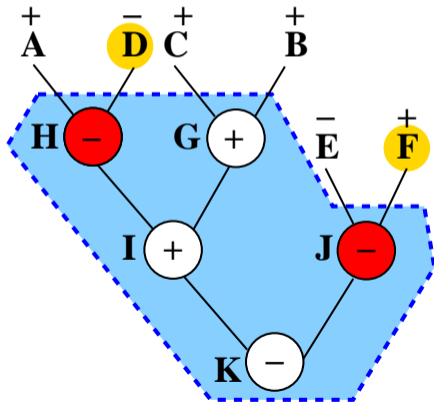


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	-
E	1	-
F	2	+
G	0	+
H	0	+
I	0	+
J	1	-
K	0	+

Linearized

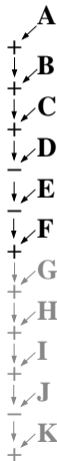


Legality

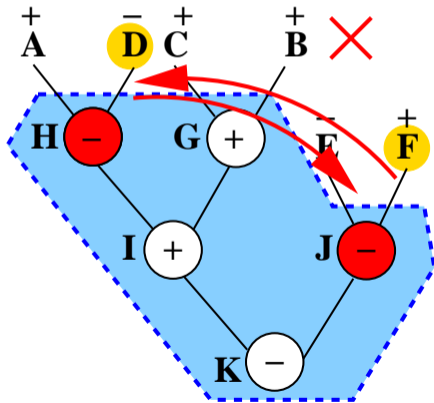


	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	-
E	1	-
F	2	+
G	0	+
H		+
I	0	+
J	0	-
K	0	+

Linearized



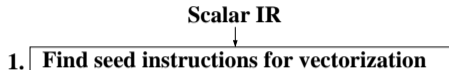
Legality



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

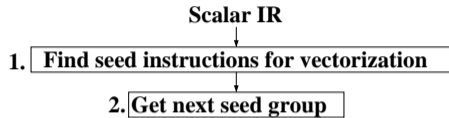
SN-SLP Algorithm

- Seed instructions are usually:
 - ① Consecutive Stores
 - ② Reductions



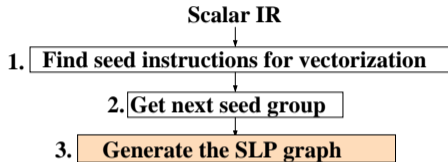
SN-SLP Algorithm

- Seed instructions are usually:
 - ① Consecutive Stores
 - ② Reductions



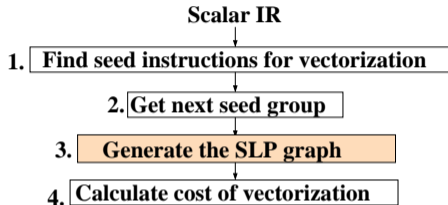
SN-SLP Algorithm

- Seed instructions are usually:
 - ① Consecutive Stores
 - ② Reductions
- Graph contains groups of vectorizable instructions



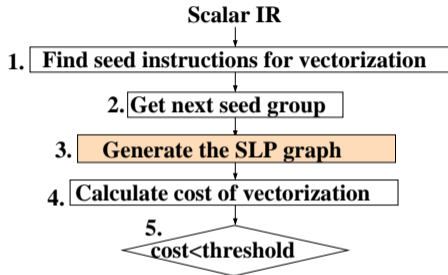
SN-SLP Algorithm

- Seed instructions are usually:
 - ① Consecutive Stores
 - ② Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count (TTI)



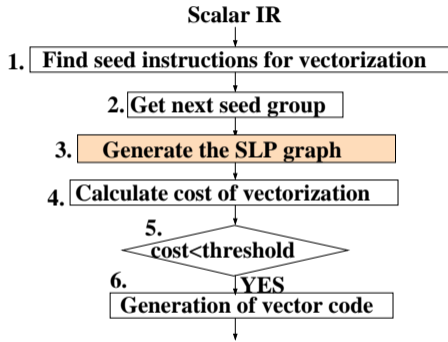
SN-SLP Algorithm

- Seed instructions are usually:
 - ① Consecutive Stores
 - ② Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count (TTI)
- Check overall profitability



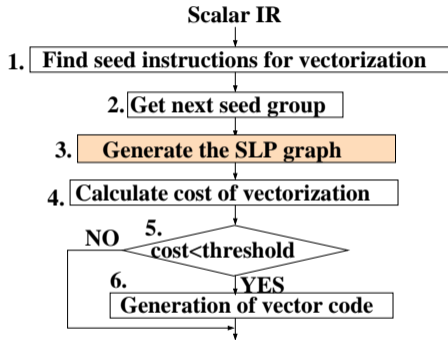
SN-SLP Algorithm

- Seed instructions are usually:
 - Consecutive Stores
 - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count (TTI)
- Check overall profitability
- Generate vector code



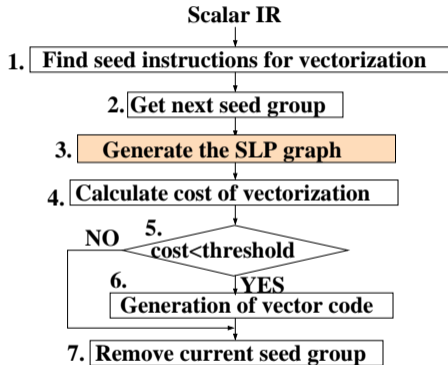
SN-SLP Algorithm

- Seed instructions are usually:
 - ① Consecutive Stores
 - ② Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count (TTI)
- Check overall profitability
- Generate vector code



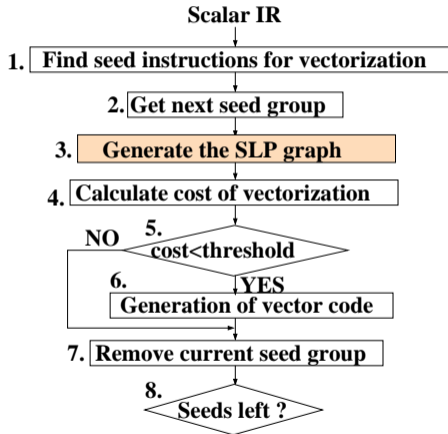
SN-SLP Algorithm

- Seed instructions are usually:
 - ① Consecutive Stores
 - ② Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count (TTI)
- Check overall profitability
- Generate vector code



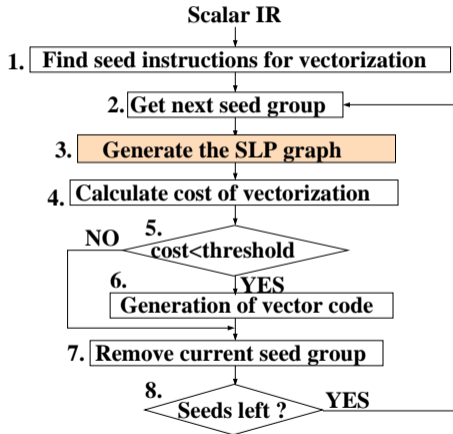
SN-SLP Algorithm

- Seed instructions are usually:
 - Consecutive Stores
 - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count (TTI)
- Check overall profitability
- Generate vector code



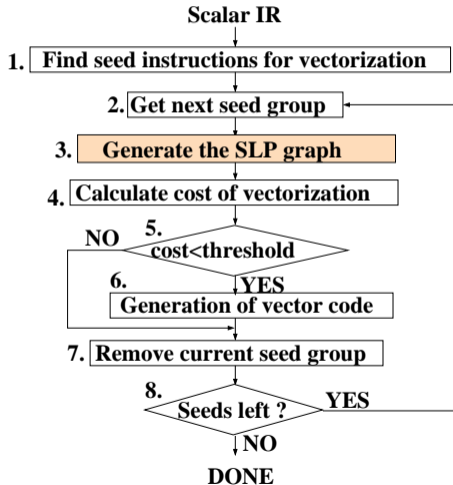
SN-SLP Algorithm

- Seed instructions are usually:
 - ① Consecutive Stores
 - ② Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count (TTI)
- Check overall profitability
- Generate vector code
- Repeat



SN-SLP Algorithm

- Seed instructions are usually:
 - Consecutive Stores
 - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count (TTI)
- Check overall profitability
- Generate vector code
- Repeat



Experimental Setup

- Implemented in LLVM trunk



Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU

Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU
- Compiler flags: `-O3 -ffast-math -march=native -mtune=native`

Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU
- Compiler flags: -O3 -ffast-math -march=native -mtune=native
- Kernels from unmodified functions of SPEC CPU2006

Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU
- Compiler flags: `-O3 -ffast-math -march=native -mtune=native`
- Kernels from unmodified functions of SPEC CPU2006
- We evaluated the following:

Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU
- Compiler flags: `-O3 -ffast-math -march=native -mtune=native`
- Kernels from unmodified functions of SPEC CPU2006
- We evaluated the following:
 - ① O3 : All vectorizers disabled

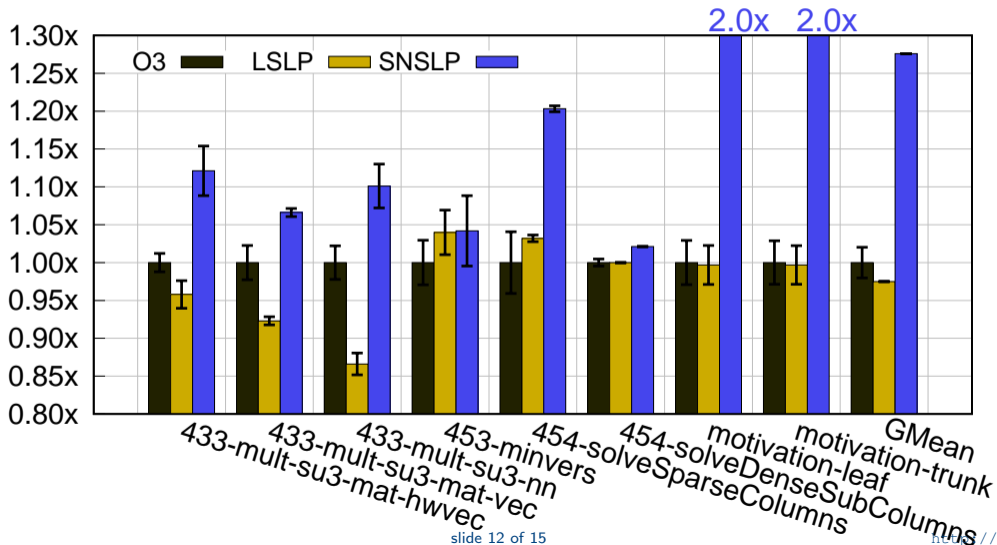
Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU
- Compiler flags: -O3 -ffast-math -march=native -mtune=native
- Kernels from unmodified functions of SPEC CPU2006
- We evaluated the following:
 - ① O3 : All vectorizers disabled
 - ② LSLP : O3 + LSLP

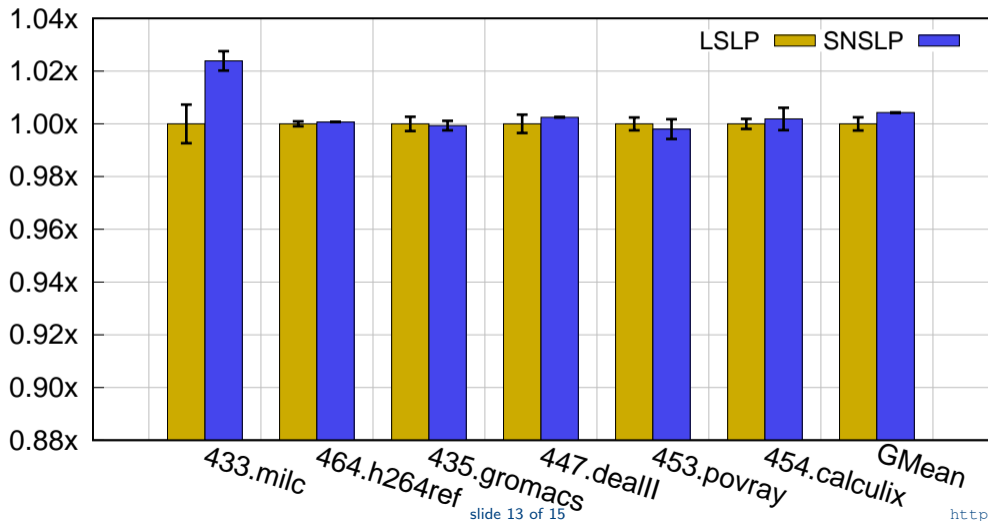
Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU
- Compiler flags: -O3 -ffast-math -march=native -mtune=native
- Kernels from unmodified functions of SPEC CPU2006
- We evaluated the following:
 - ① O3 : All vectorizers disabled
 - ② LSLP : O3 + LSLP
 - ③ SNSLP : O3 + SN-SLP

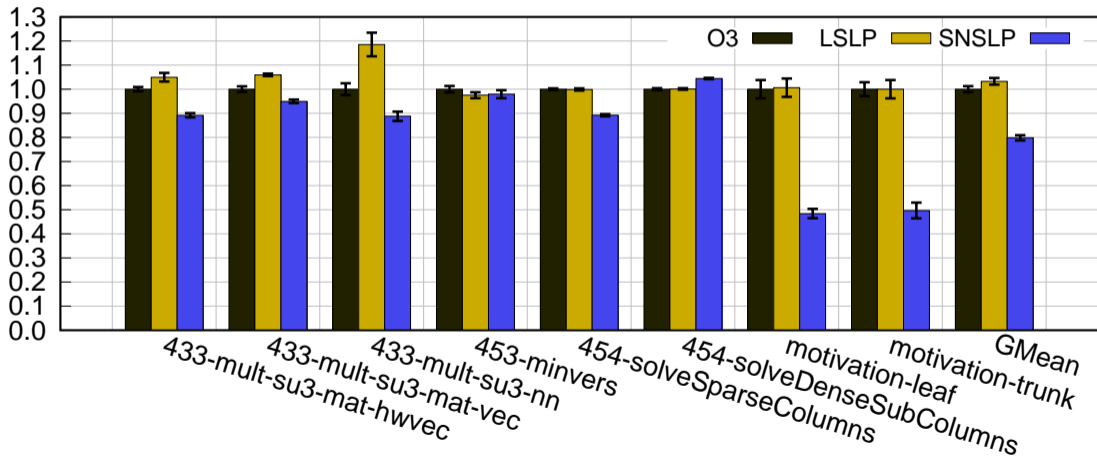
Performance of kernels



Performance (Full Benchmarks)



Total Compilation Time



Conclusion

- SN-SLP improves the effectiveness of SLP on code with inverse elements.

Conclusion

- SN-SLP improves the effectiveness of SLP on code with inverse elements.
 - ① It forms Super-Nodes of commutative operations and their inverse elements
 - ② It performs legal operand reordering, guided by the Look-Ahead heuristic

Conclusion

- SN-SLP improves the effectiveness of SLP on code with inverse elements.
 - ① It forms Super-Nodes of commutative operations and their inverse elements
 - ② It performs legal operand reordering, guided by the Look-Ahead heuristic
- Better at identifying isomorphism

Conclusion

- SN-SLP improves the effectiveness of SLP on code with inverse elements.
 - ① It forms Super-Nodes of commutative operations and their inverse elements
 - ② It performs legal operand reordering, guided by the Look-Ahead heuristic
- Better at identifying isomorphism
- Implemented in LLVM as an extension of SLP

Conclusion

- SN-SLP improves the effectiveness of SLP on code with inverse elements.
 - ① It forms Super-Nodes of commutative operations and their inverse elements
 - ② It performs legal operand reordering, guided by the Look-Ahead heuristic
- Better at identifying isomorphism
- Implemented in LLVM as an extension of SLP
- Improves performance with similar compilation time