Look-Ahead SLP: Auto-vectorization in the Presence of Commutative Operations

CGO 2018

Vasileios Porpodas
Intel Corporation
USA

Rodrigo C. O. Rochas
University of Edinburgh
UK

Luis F. W. Goes
PUC Minas
Brazil

EECS 583

Cangsu Lei, Shihao Liu
Yangming Ke, Zhensheng Jiang

12-04-2019

Contents

- Background
- Motivation
- Look-Ahead SLP
- Result & Conclusion

Background

SLP: The Straight-Line Code vectorizer

- Superword Level Parallelism
- Vectorizes across instructions

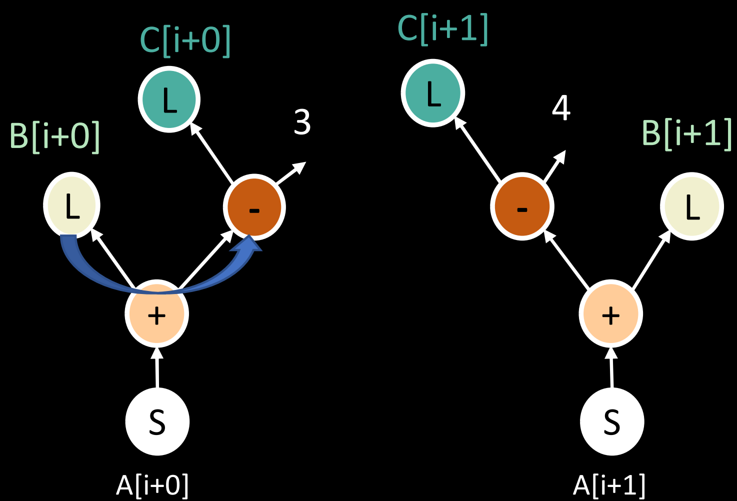
```
A[i] = B[i];
A[i+1] = B[i+1];
A[i+2] = B[i+2];
A[i+3] = B[i+3];
```

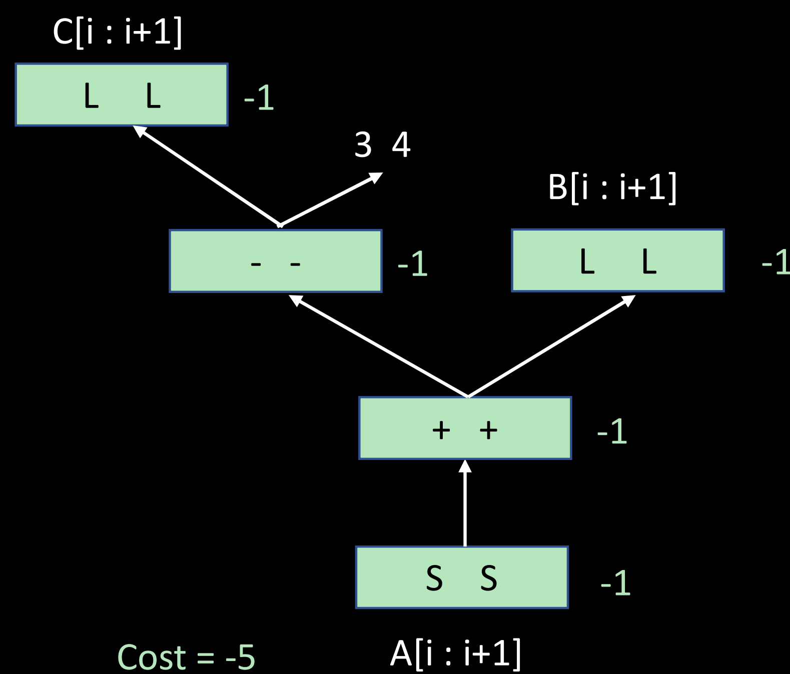
$\xrightarrow{\text{SLP}}$

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

How SLP works

```
long A[], B[], C[];
A[i+0] = B[i+0]         +  (C[i+0] – 3);
A[i+1] = (C[i+1] – 4)  +  B[i+1];
```





Motivation

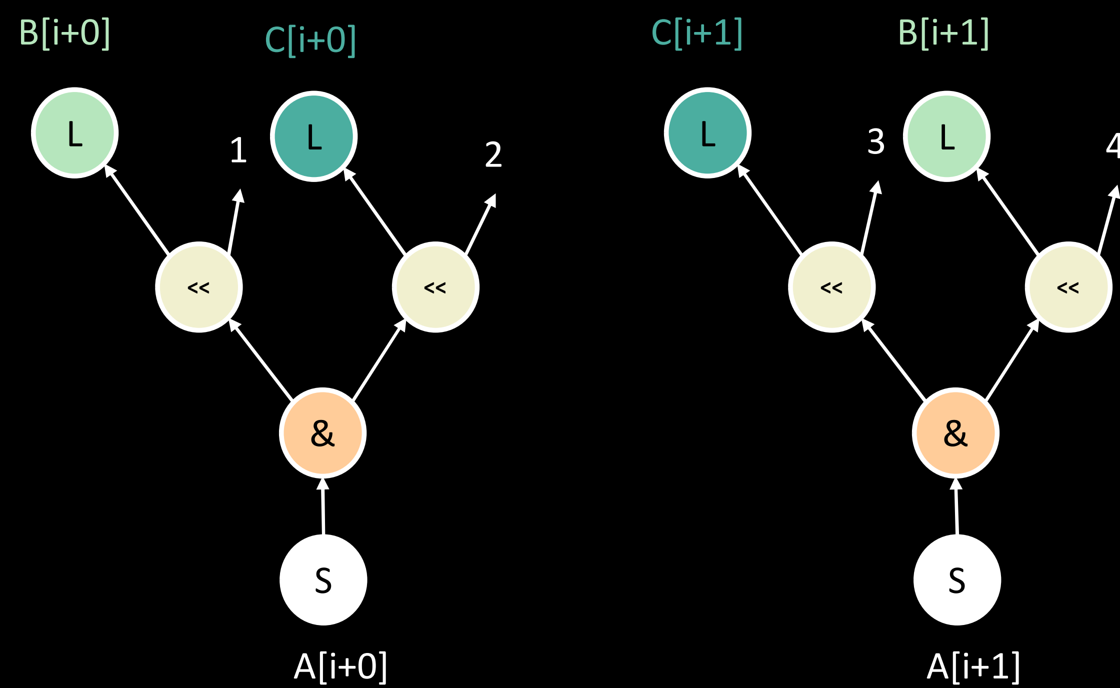
- Load Address Mismatch
- Opcode Mismatch
- Associativity Mismatch and Multi-Node

SLP Failure: Load Address Mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```

SLP Failure: Load Address Mismatch

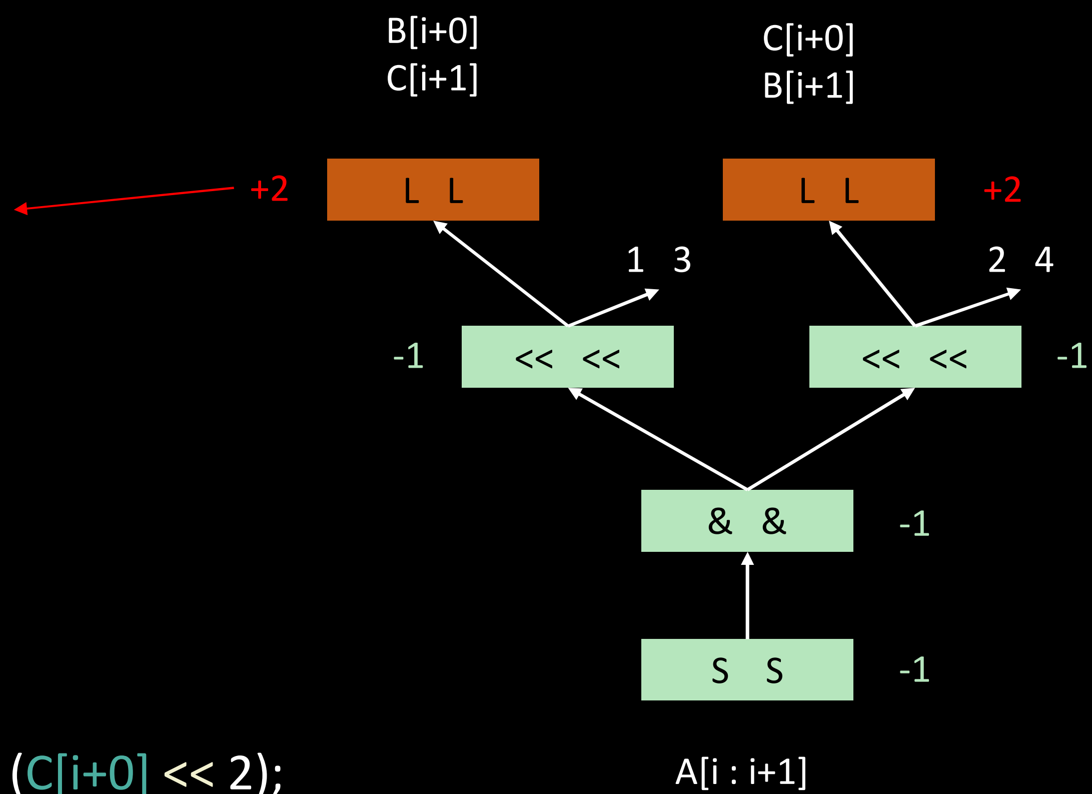
Use-def Chains DAG




```
1 long A[], B[], C[];
2 A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
3 A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```

SLP Failure: Load Address Mismatch

Extra cost for gathering and aggregating



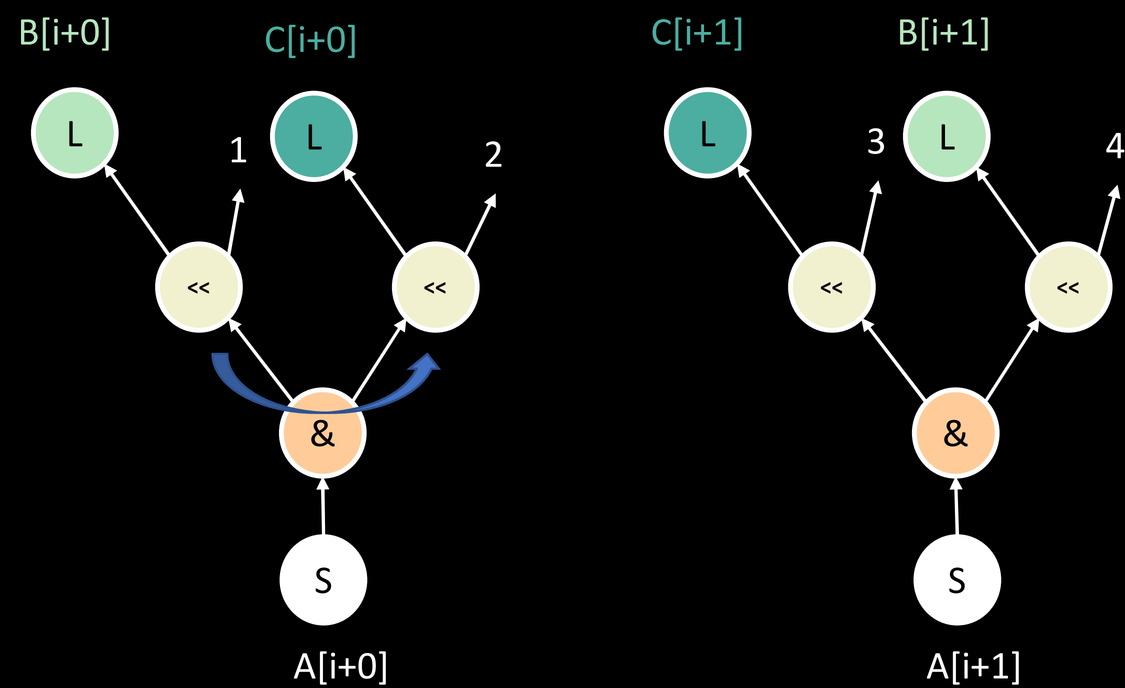
Total cost: 0

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```

SLP

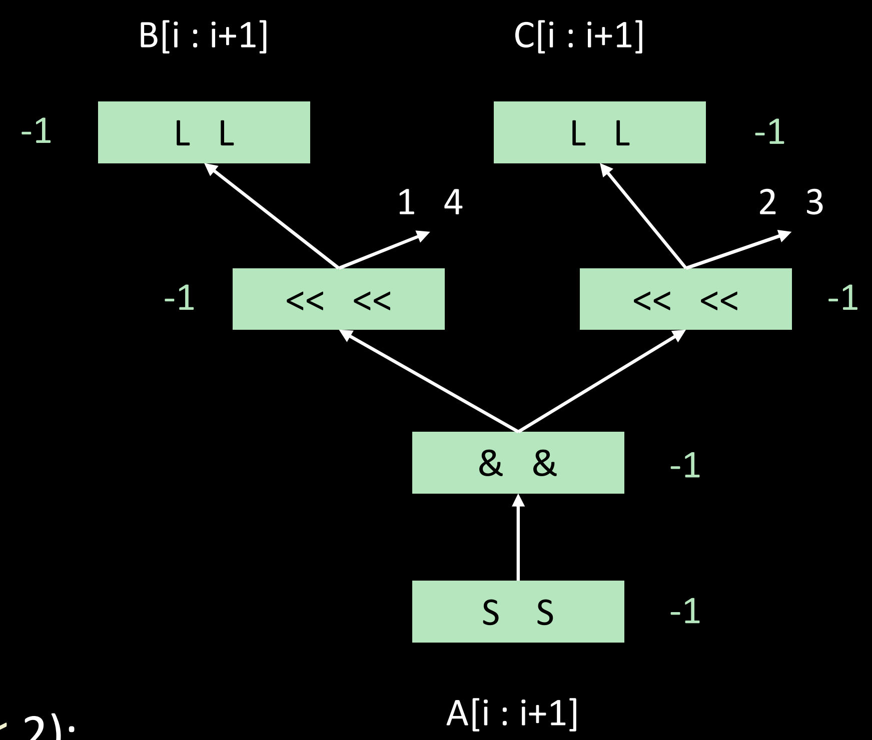
SLP Failure: Load Address Mismatch

Use-def Chains DAG




```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```

SLP Failure: Load Address Mismatch



Total cost: -6

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```

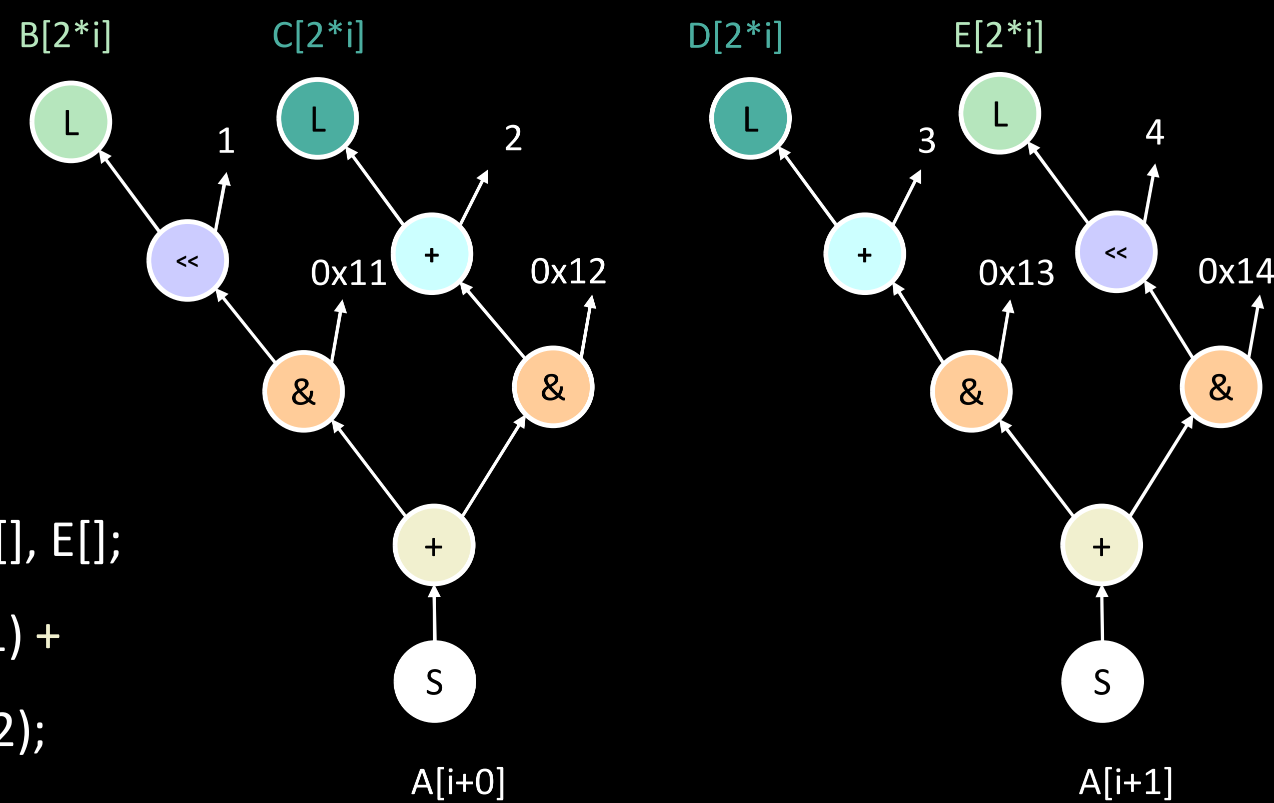
LSLP

SLP Failure: Opcode Mismatch

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = ((B[2*i] << 1) & 0x11) +
         ((C[2*i]  +   2) & 0x12);
A[i+1] = ((D[2*i]  +   3) & 0x13) +
         ((E[2*i] << 4) & 0x14);
```

SLP Failure: Opcode Mismatch

Use-def Chains DAG

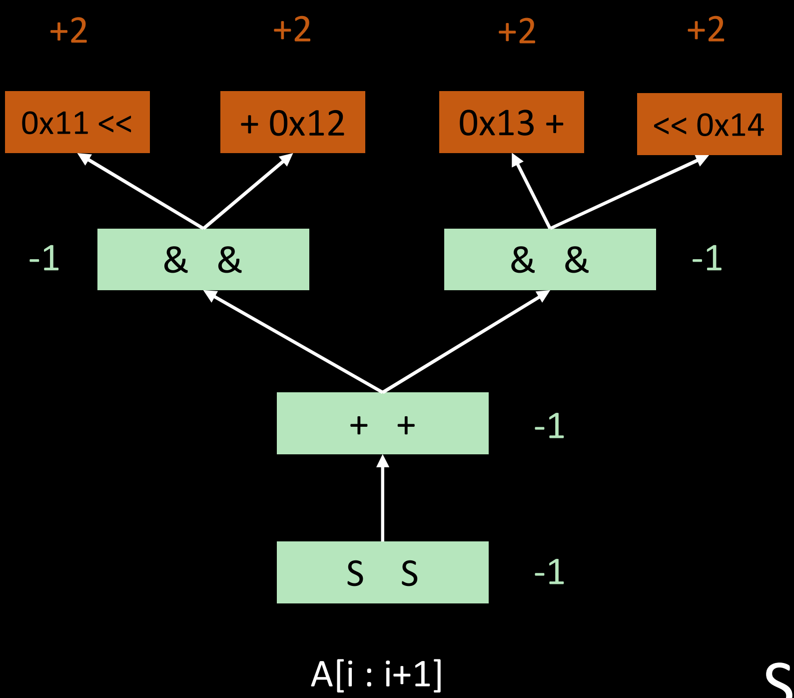



```
1 unsigned long A[], B[], C[], D[], E[];
2 A[i+0] = ((B[2*i] << 1) & 0x11) +
           ((C[2*i]  +   2) & 0x12);
3 A[i+1] = ((D[2*i]  +   3) & 0x13) +
           ((E[2*i] << 4) & 0x14);
```

SLP Failure: Opcode Mismatch

Total cost: +4

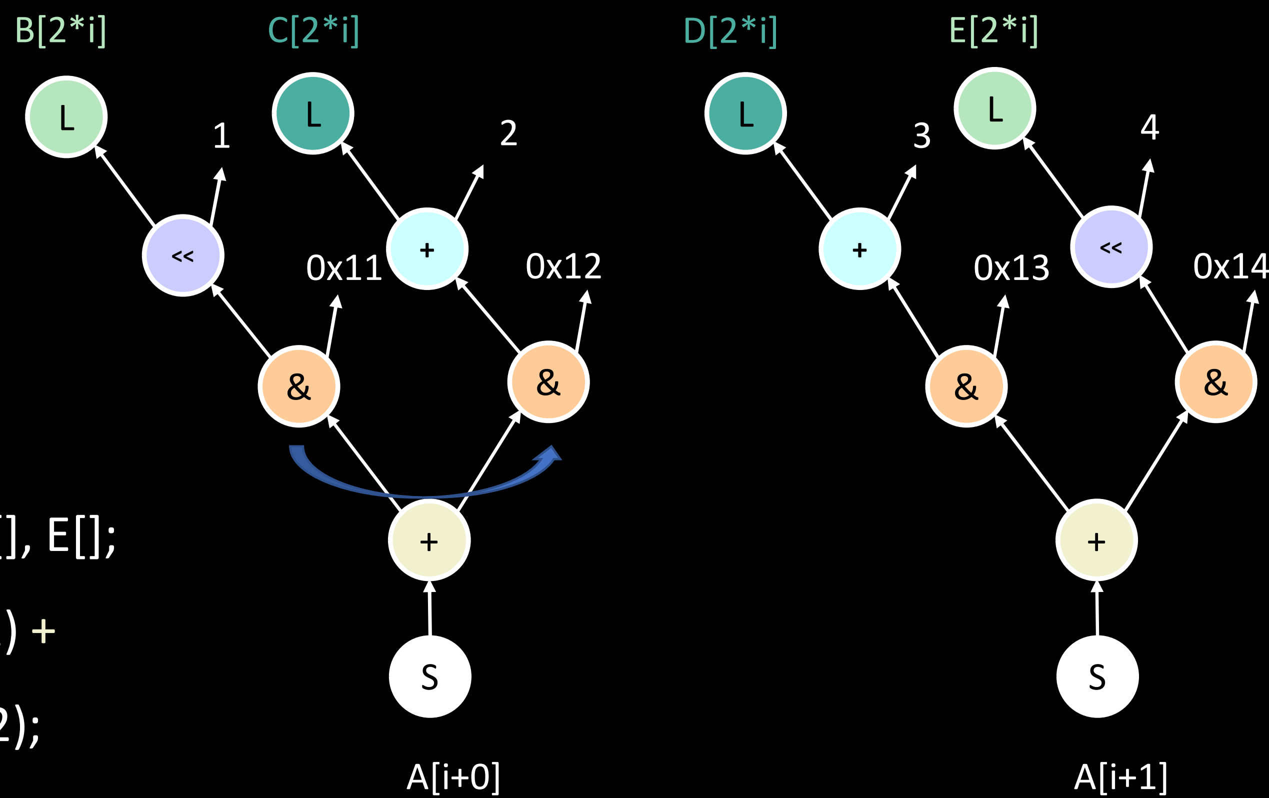
```
1  unsigned long A[], B[], C[], D[], E[];
2  A[i+0] = ((B[2*i] << 1) & 0x11) +
            ((C[2*i]  +  2) & 0x12);
3  A[i+1] = ((D[2*i]  +  3) & 0x13) +
            ((E[2*i] << 4) & 0x14);
```



SLP

SLP Failure: Opcode Mismatch

Use-def Chains DAG

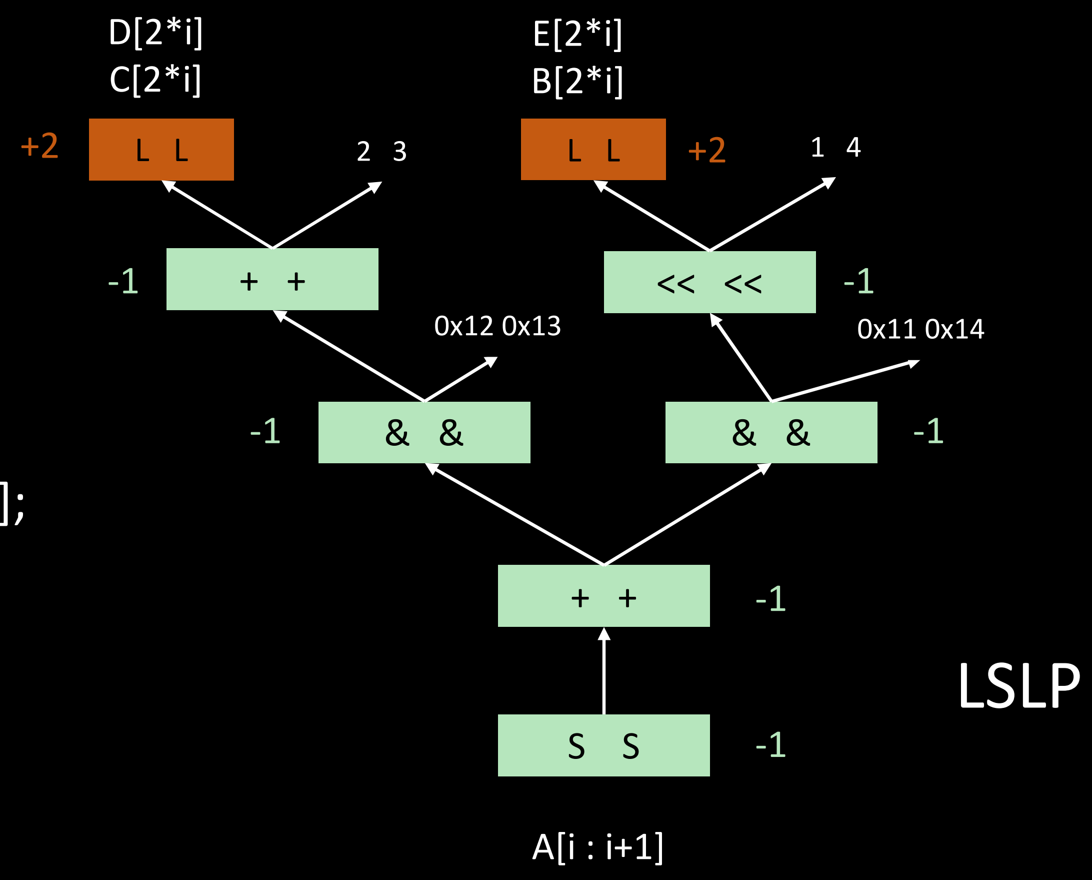



```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = ((B[2*i] << 1) & 0x11) +
         ((C[2*i]  +  2) & 0x12);
A[i+1] = ((D[2*i]  +  3) & 0x13) +
         ((E[2*i] << 4) & 0x14);
```

SLP Failure: Opcode Mismatch

Total cost: -2

```
1 unsigned long A[], B[], C[], D[], E[];
2 A[i+0] = ((B[2*i] << 1) & 0x11) +
           ((C[2*i]  +  2) & 0x12);
3 A[i+1] = ((D[2*i]  +  3) & 0x13) +
           ((E[2*i] << 4) & 0x14);
```

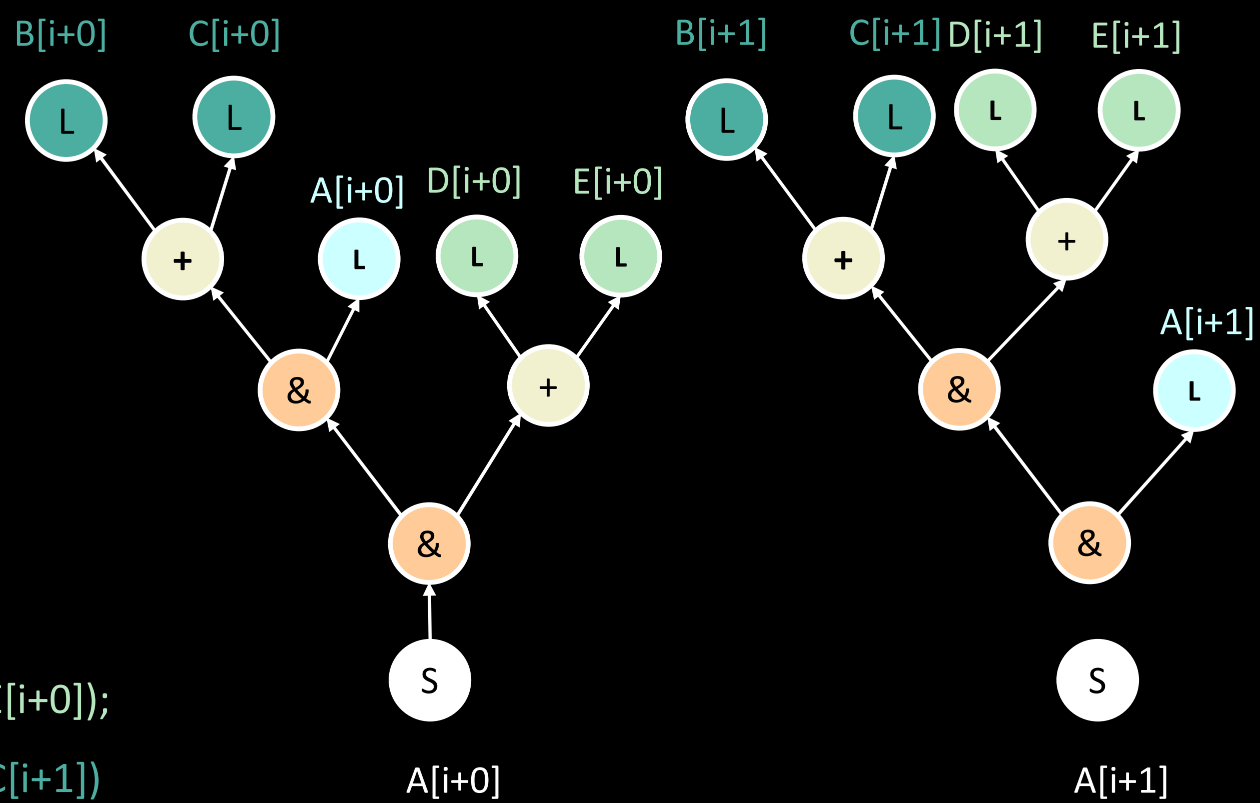


SLP Failure: Associativity Mismatch and Multi-Node

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = A[i+0] &
            (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1])
            & A[i+1];
```

SLP Failure: Associativity Mismatch and Multi-Node

Use-def Chains DAG

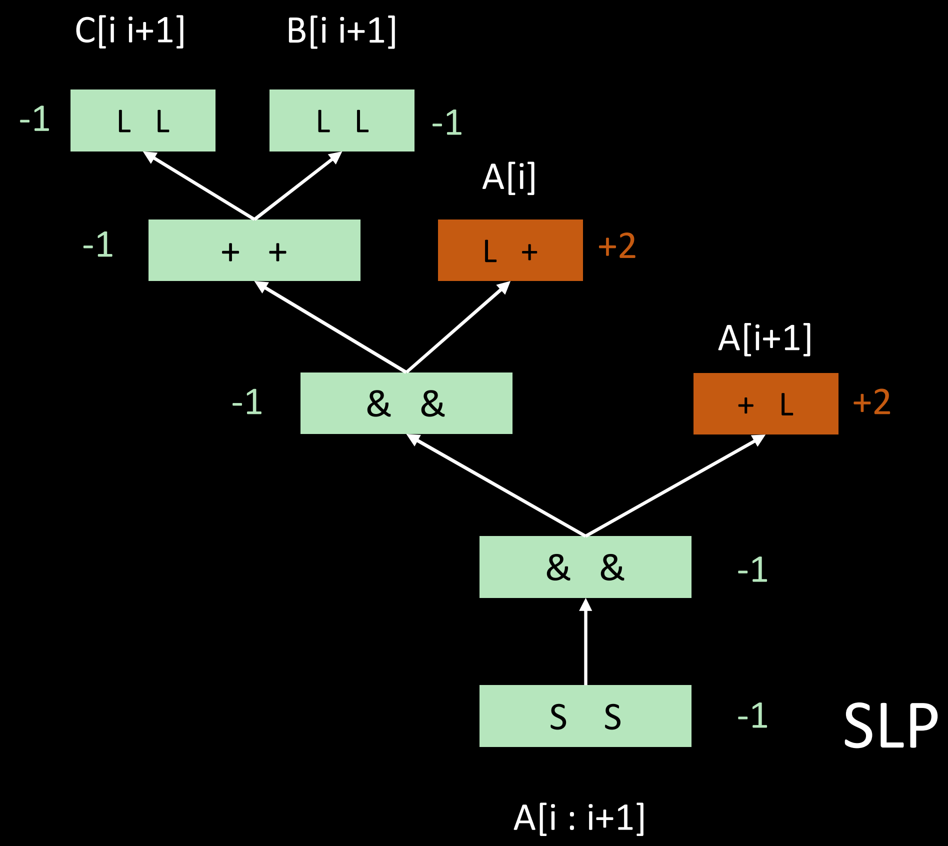



```
1 unsigned long A[], B[], C[], D[], E[];
2 A[i+0] = A[i+0] &
          (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
3 A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1])
          & A[i+1];
```

SLP Failure: Associativity Mismatch and Multi-Node

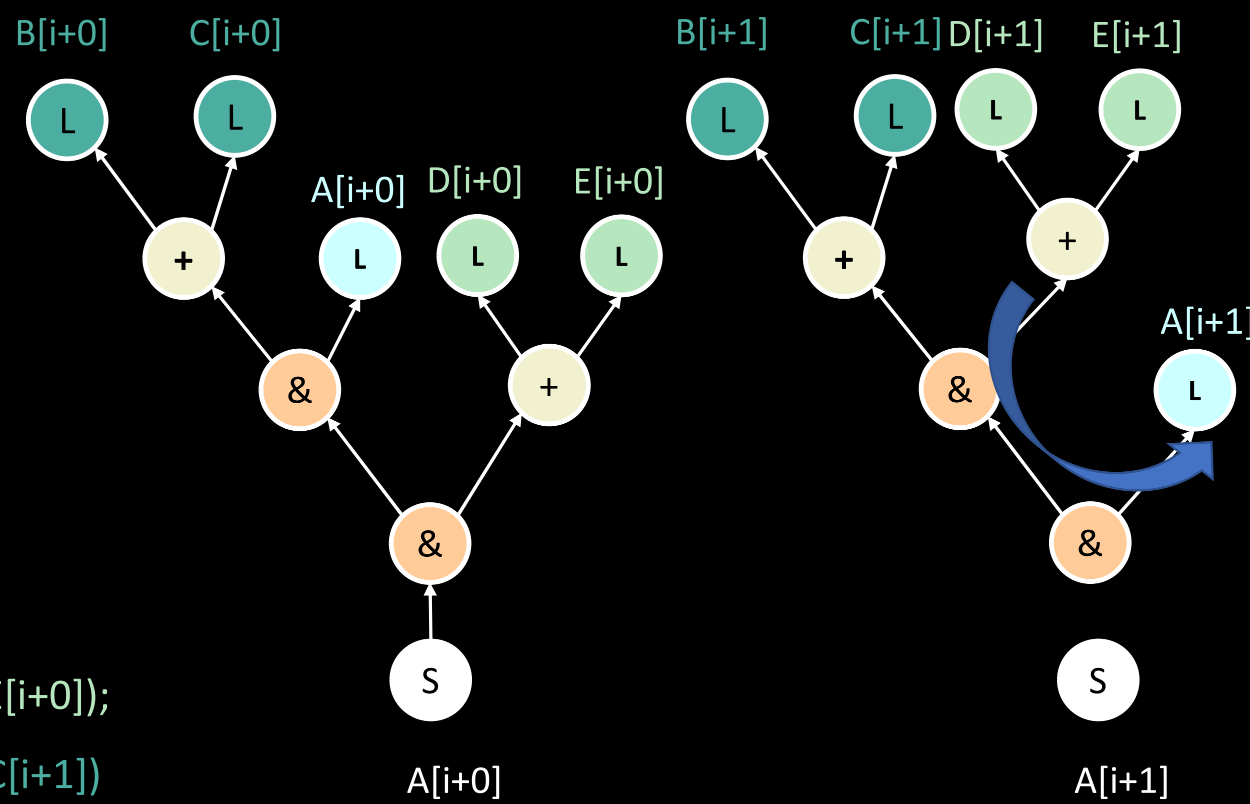
Total cost: -2

```
1 unsigned long A[], B[], C[], D[], E[];
2 A[i+0] = A[i+0] &
          (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
3 A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1])
          & A[i+1];
```



SLP Failure: Associativity Mismatch and Multi-Node

Use-def Chains DAG

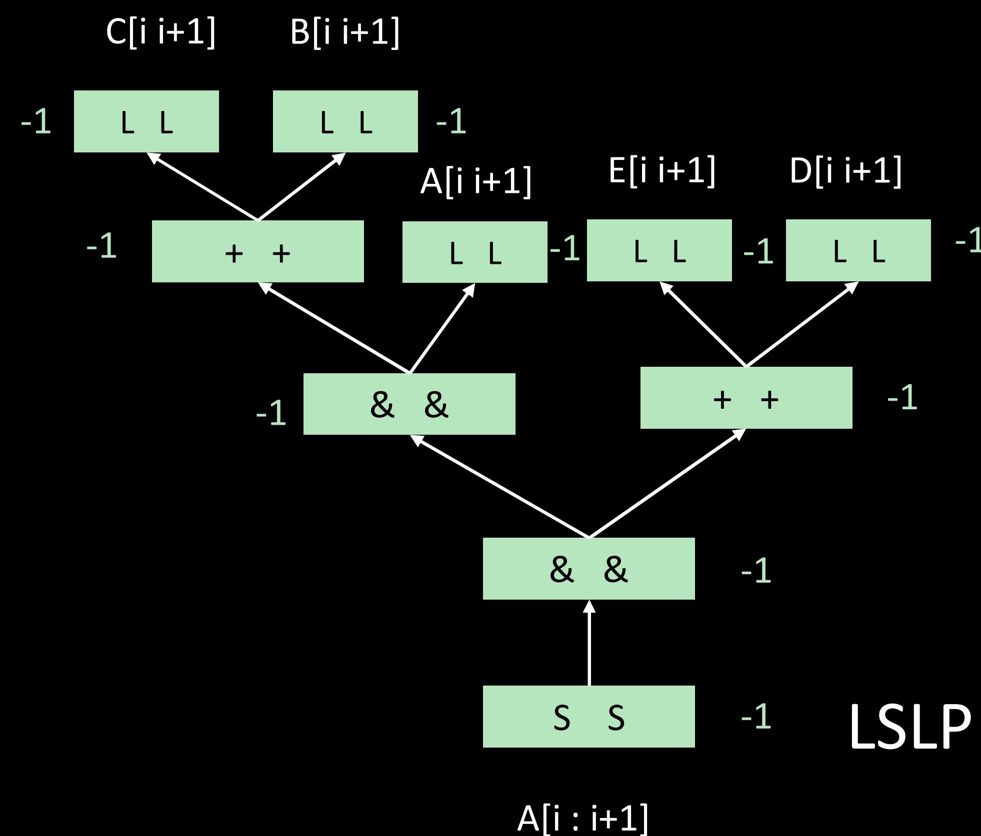



```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = A[i+0] &
        (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1])
        & A[i+1];
```

SLP Failure: Associativity Mismatch and Multi-Node

Total cost: -10

```
1 unsigned long A[], B[], C[], D[], E[];
2 A[i+0] = A[i+0] &
           (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
3 A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1])
           & A[i+1];
```



LSLP

Motivation

- Load Address Mismatch

  SLP: 0 LSLP: -6
- Opcode Mismatch

  SLP: +4 LSLP: -2
- Associativity Mismatch and Multi-Node

  SLP: -2 LSLP: -10

Look-Ahead SLP – Overview

1. Multi-Node Formation
   - Multi-Node: Chained multiple commutative operations;
   - e.g.: A[i+0] & (B[i+0] + C[i+0]) & (D[i+0] + E[i+0])
2. Top-Level Operand Reordering
   - Use Look-Ahead score for non-trivial case;
3. Run (1) recursively on reordered operands.

Multi-Node Formation

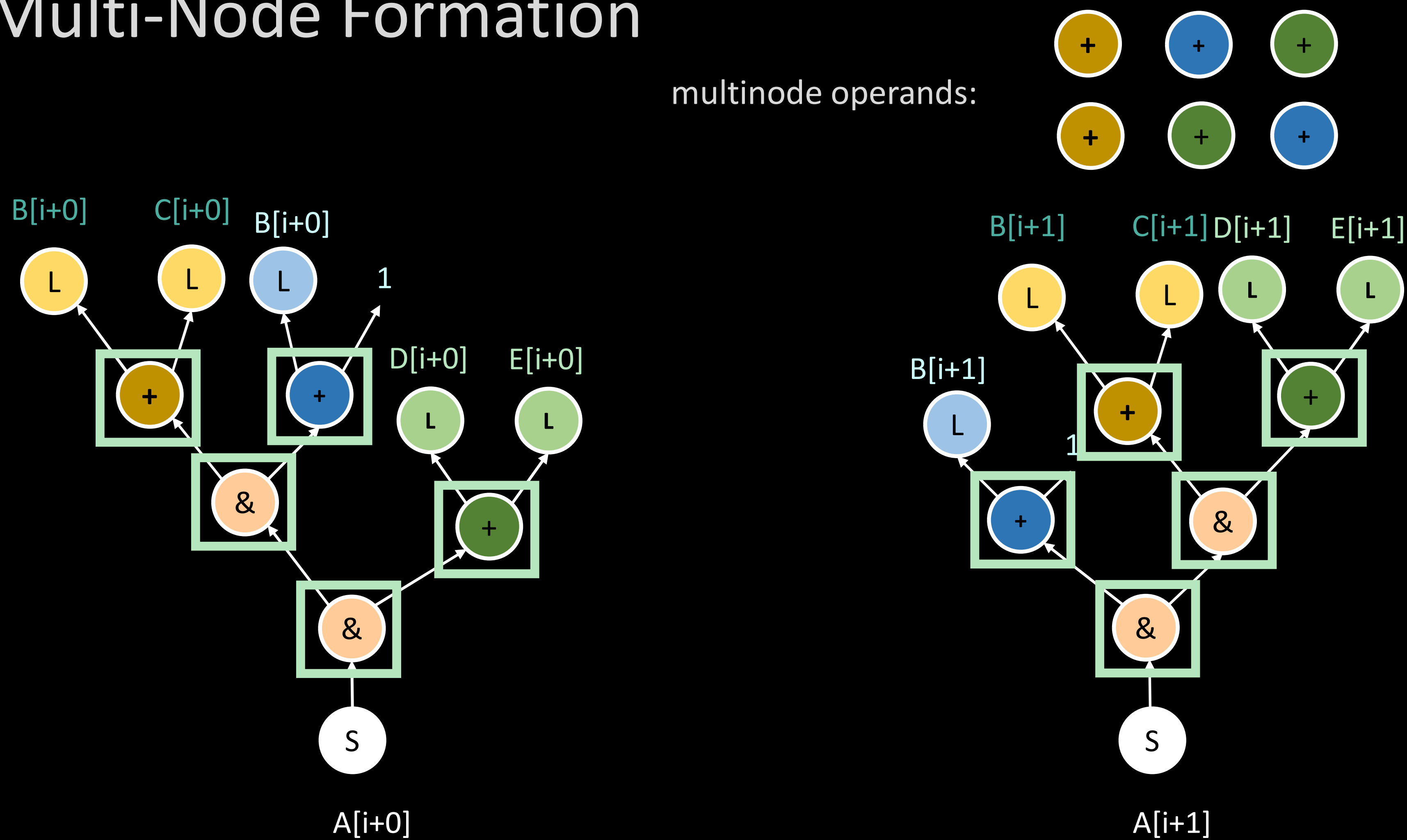
```
build_graph(values):
  if values are non-vectorizable: return
  if values are commutative:
    foreach operands of values:
      if opcode matches
        and not escape multi-node:
         build_graph(operands)
      else:
         add operands to multimode-operands
     if values are root of multi-node:
        // reorder and build_graph recursively
  else:
     // build_graph recursively
```

Core Idea:

1. Given a commutative operation at the root;
2. Find connected component with same opcode in the children;
3. Group the operands of the members of connected component together.

Multi-Node Formation

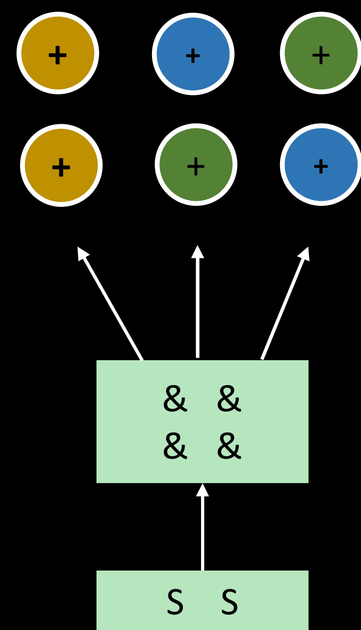
multinode operands:



Multi-Node Formation - Result

multinode operands:

Every operand can be reordered inside its lane

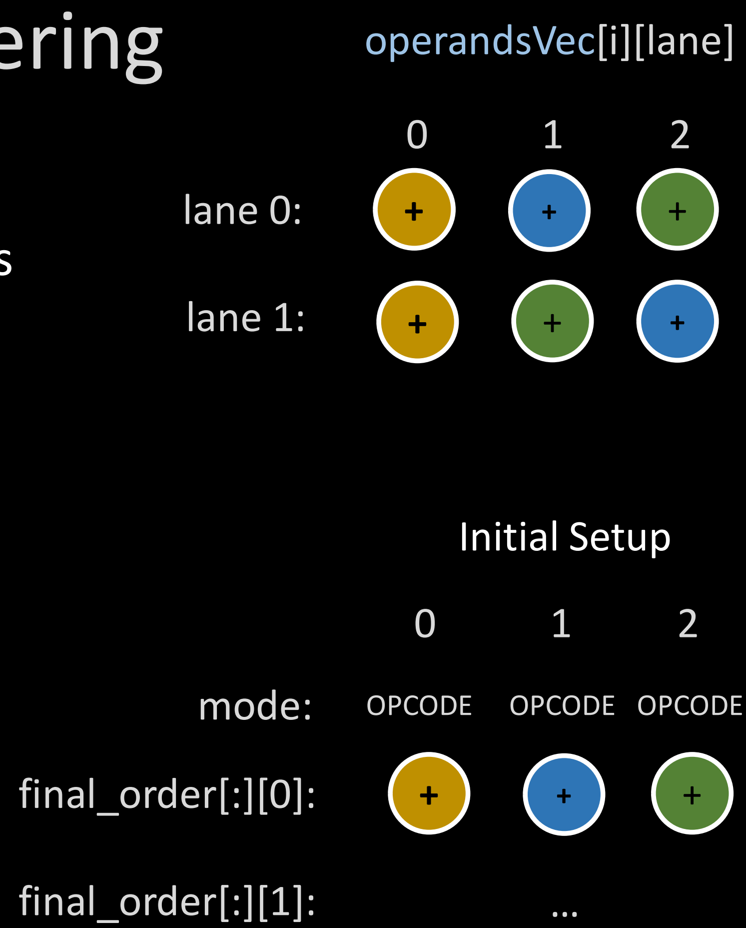


A chain of & operations

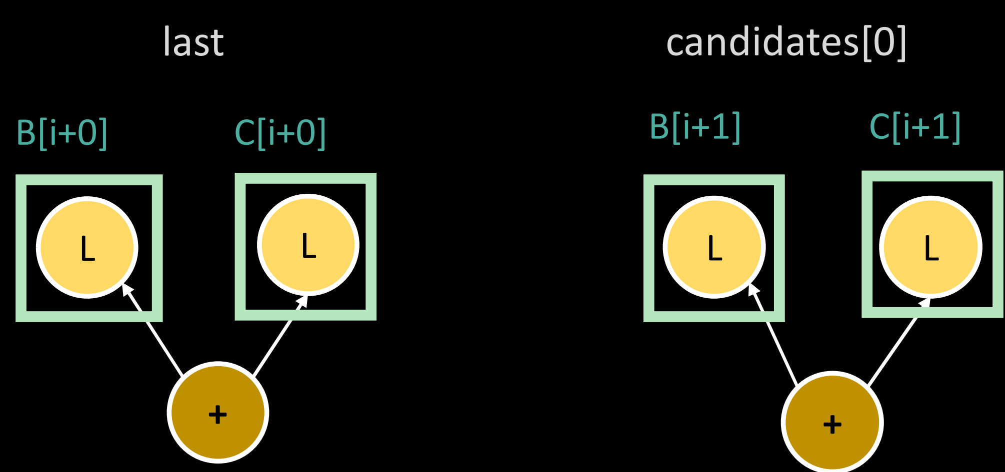
A[i : i+1]

Top-Level Operand Reordering

```
reorder_operands(operandsVec):
  setup mode and final_order for operands
based on first lane
  foreach lane:
    foreach i in {0..numOperands}
      if mode[i] is FAILED: continue
        last = final_order[i][lane - 1]
        get_best(mode[i], last,
operandsVec[:][lane])
        update final_order
  return final_order
```

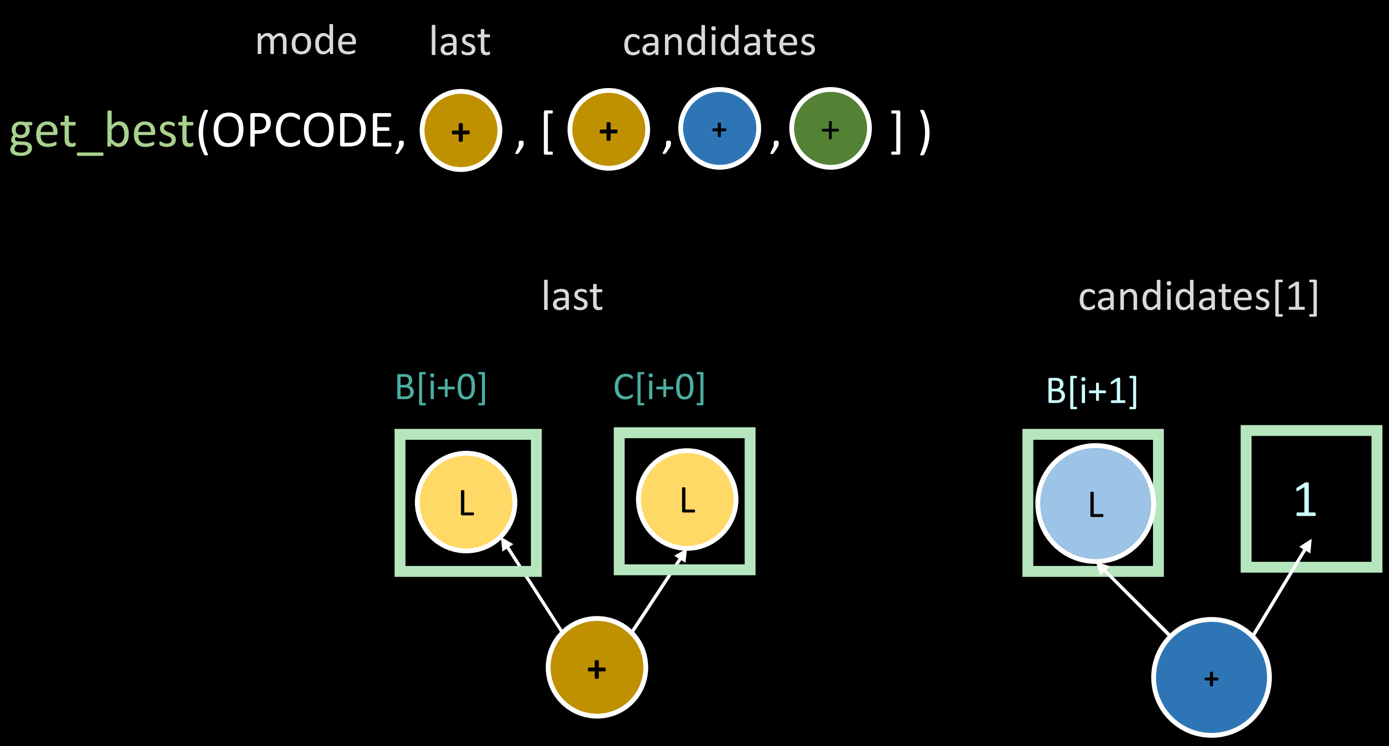


Top-Level Operand Reordering – get_best



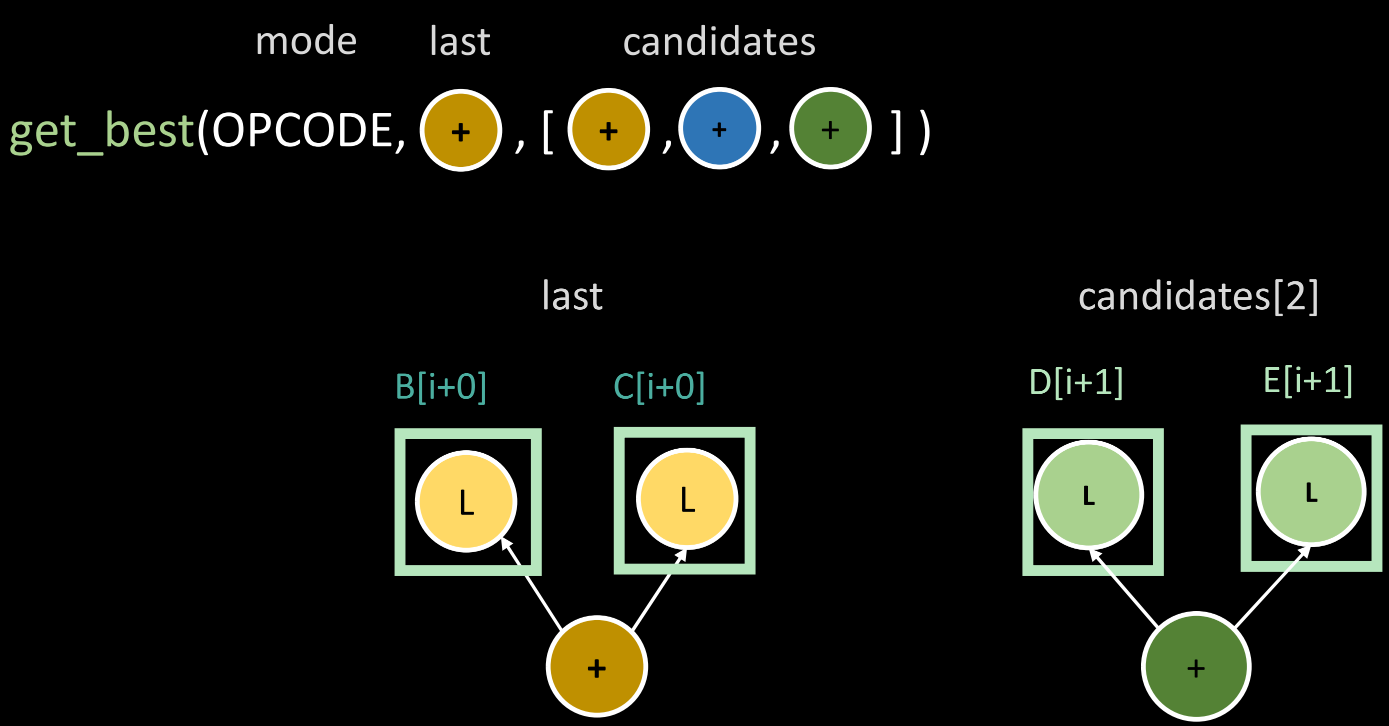
LAScore: 1+0+0+1 = 2

Top-Level Operand Reordering – get_best



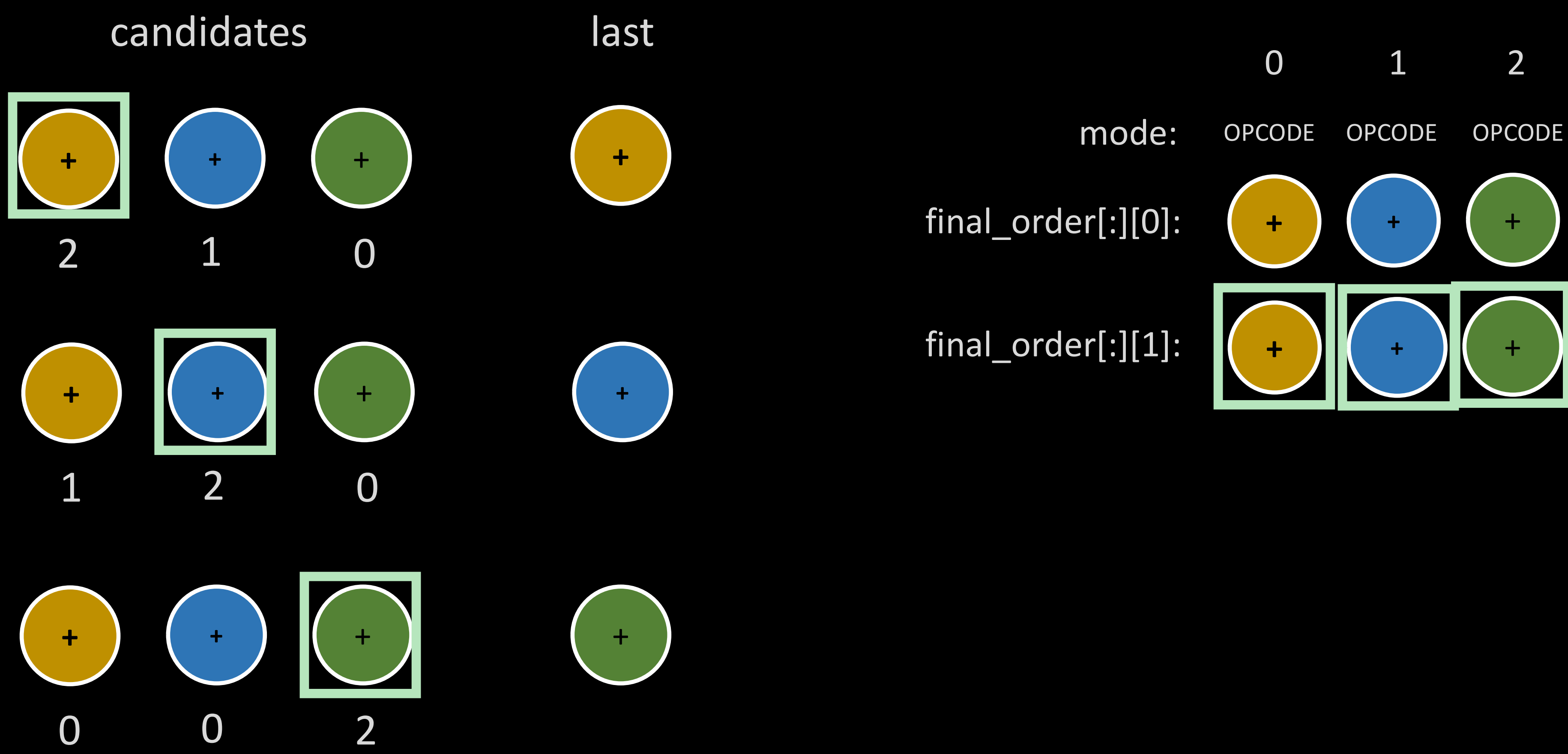
LAScore: 1+0+0+0 = 1

Top-Level Operand Reordering – get_best



LAScore: 0+0+0+0 = 0

Top-Level Operand Reordering – Result



Experimental Setup

Evaluated the following cases:

- Baseline: All Loop, SLP and LSLP vectorizers disabled (O3)
- O3 + SLP enabled but No Operand Reordering(SLP-NR)
- O3 + SLP enabled (SLP)
- O3 + LSLP enabled (LSLP)

Evaluation – Speedup

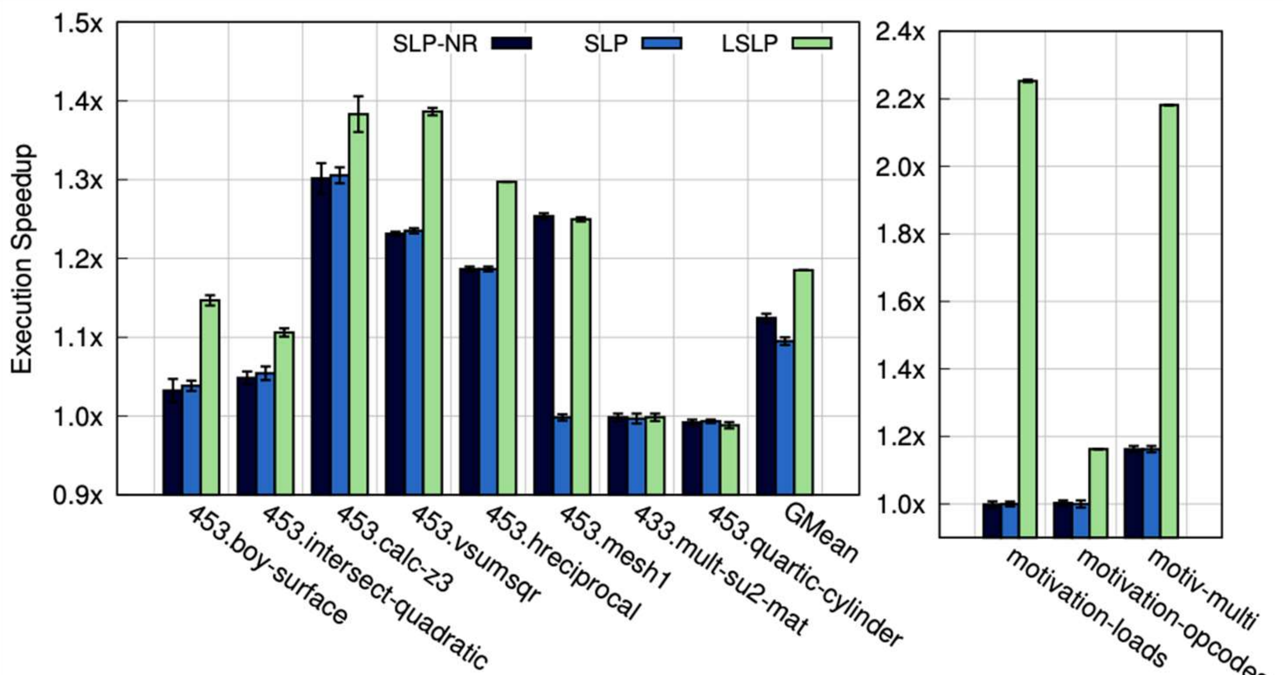


Figure 9. Speedup of LSLP, SLP and SLP-NR over O3.

Evaluation – Total Static Cost Seen by Each Scheme

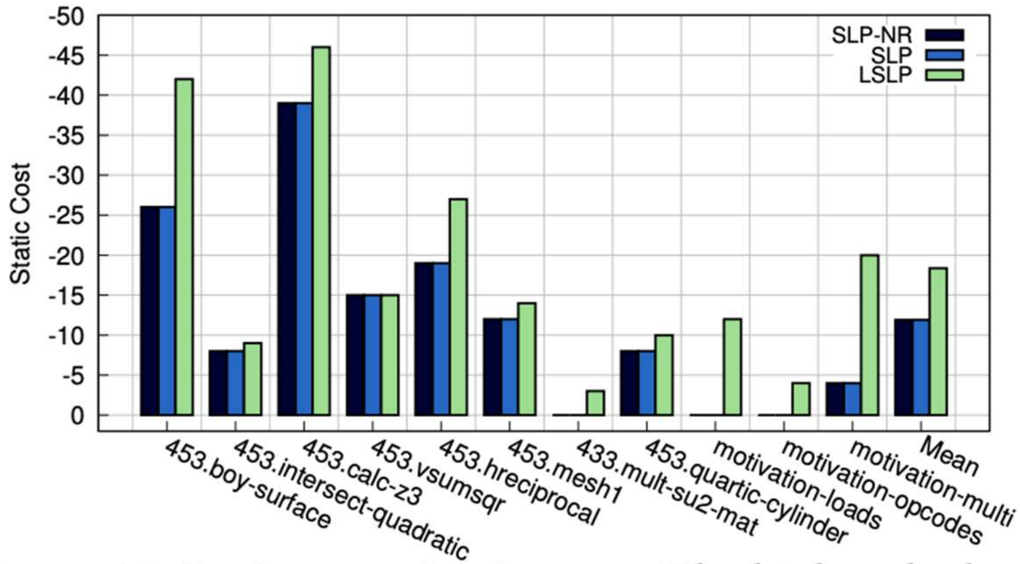


Figure 10. Static vectorization cost. The higher the better.

LSLP can locally improve individual vectorization regions compared to both SLP and SLP-NR

Evaluation – Compilation Time

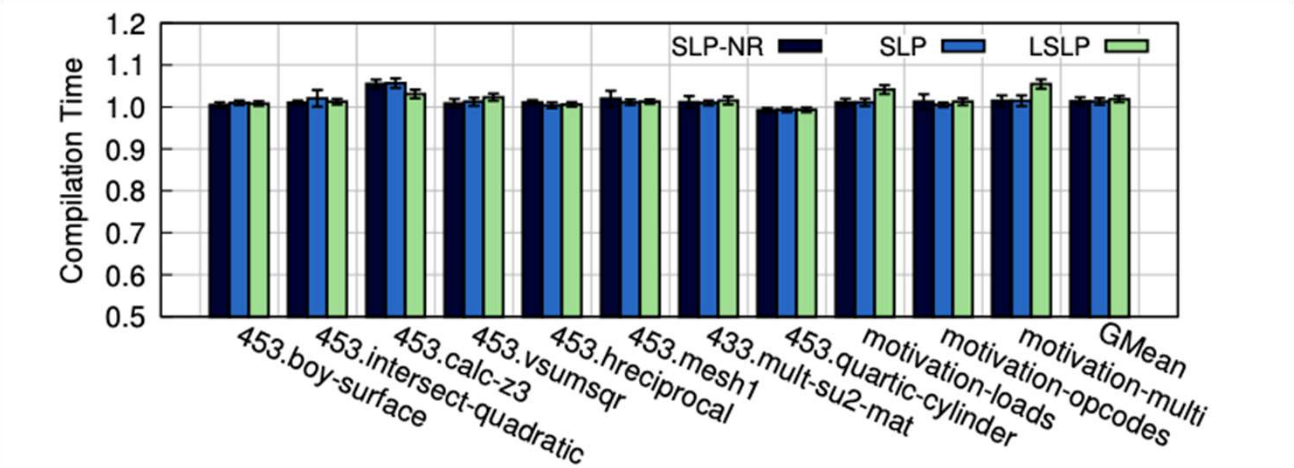


Figure 14. Compilation time, normalized to O3 (LA=8).

On average, LSLP increases the compilation over SLP slightly, by less than 1%

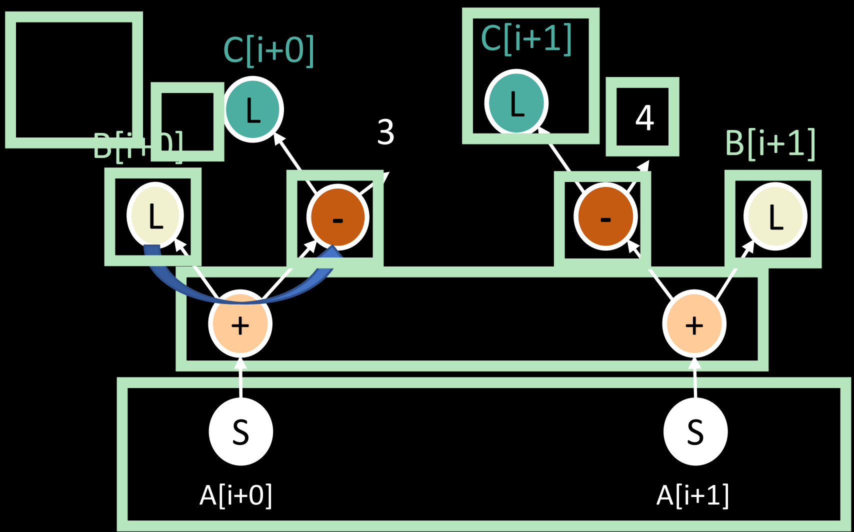
Discussion

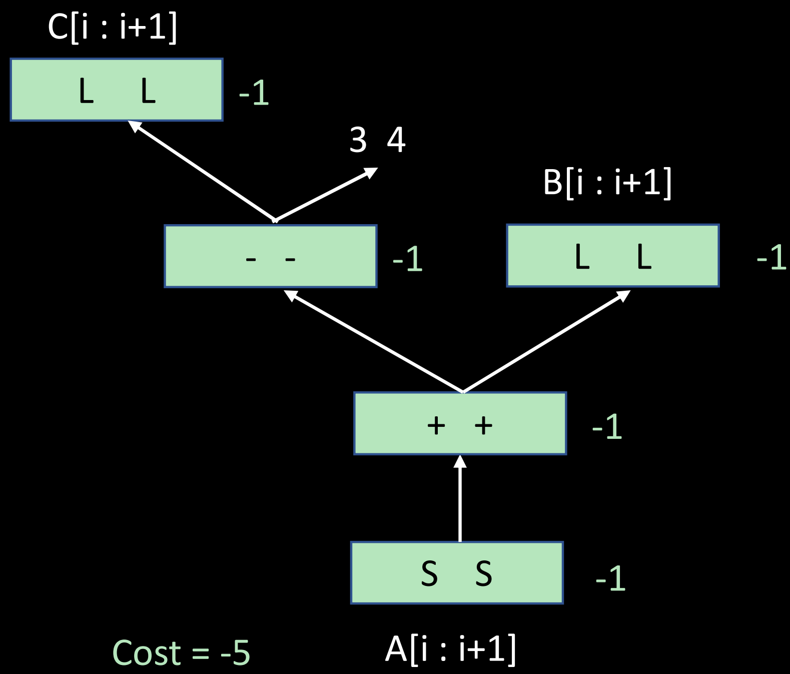
- LSLP introduces an effective scheme for dealing with commutative operations with little compilation time overhead.
- There are certain cases that LSLP can fail, which can be further improved.

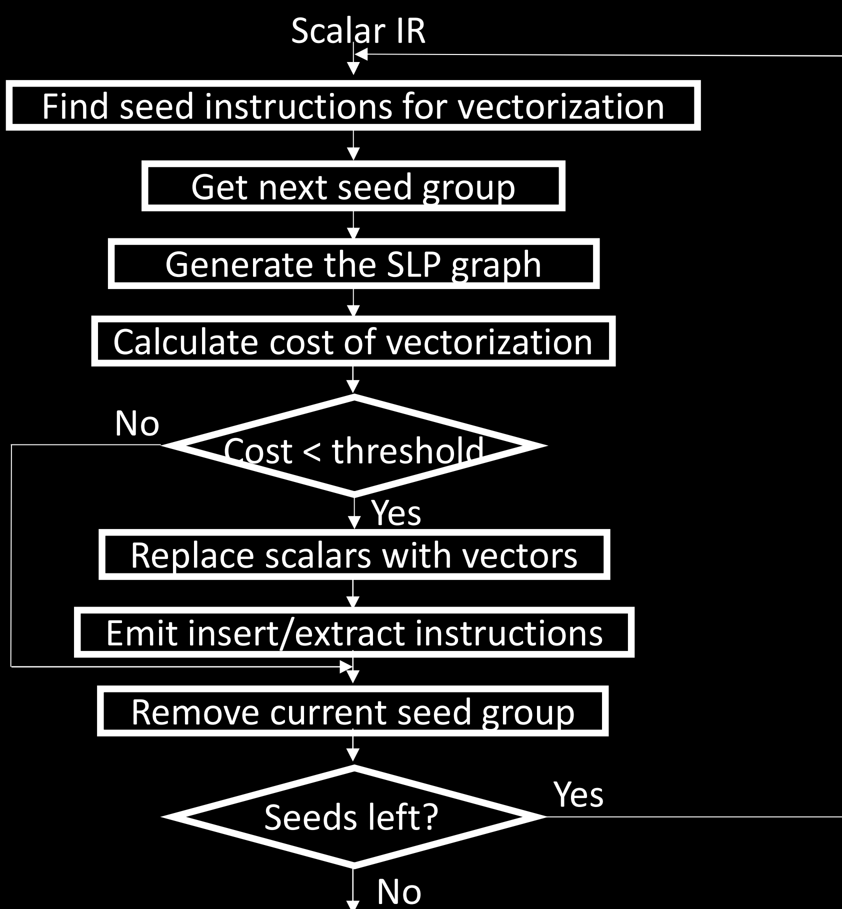
Q&A

How SLP works

```
long A[], B[], C[];
A[i+0] = B[i+0]         +  (C[i+0] – 3);
A[i+1] = (C[i+1] – 4)  +  B[i+1];
```








```
long A[], B[], C[];
A[i+0] = B[i+0]         +  (C[i+0] – 3);
A[i+1] = (C[i+1] – 4)  +  B[i+1];
```