SuperGraph-SLP Auto-Vectorization

Vasileios Porpodas

Intel

PACT 2017

Legal Disclaimer & Optimization Notice

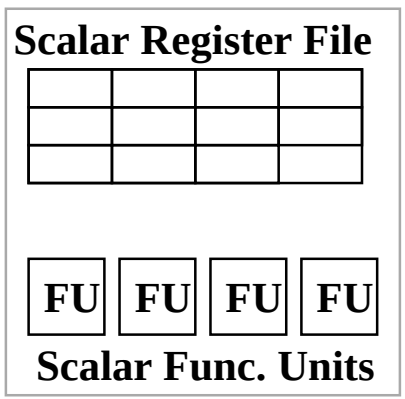
Optimization Notice

Intel's 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

Why SIMD Vectorization?

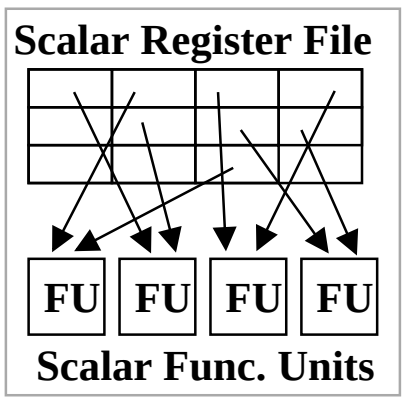
- Scalable parallelism (compared to ILP)



Instruction Level Parallelism(ILP)

Why SIMD Vectorization?

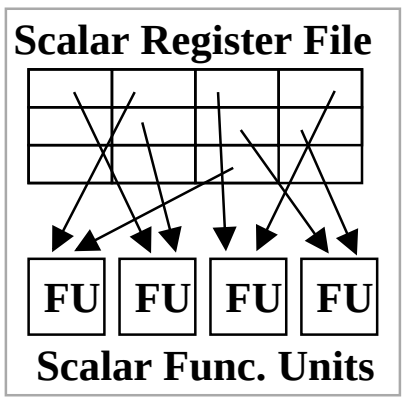
- Scalable parallelism (compared to ILP)



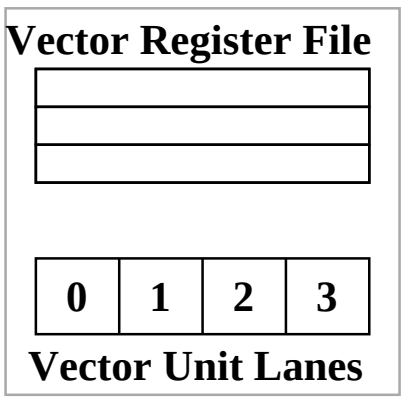
Instruction Level Parallelism(ILP)

Why SIMD Vectorization?

- Scalable parallelism (compared to ILP)



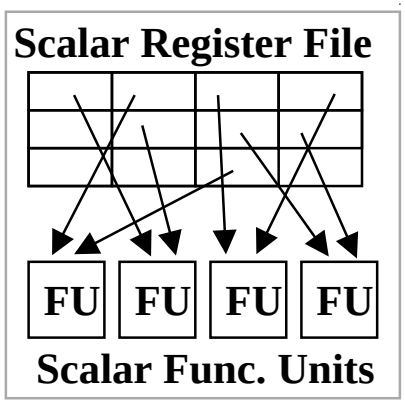
Instruction Level Parallelism(ILP)



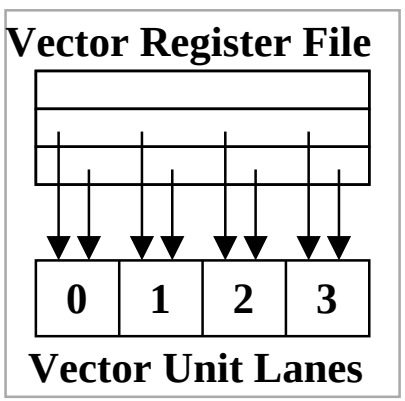
Vector Parallelism

Why SIMD Vectorization?

- Scalable parallelism (compared to ILP)



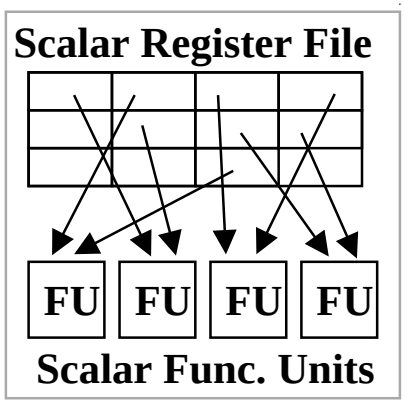
Instruction Level Parallelism(ILP)



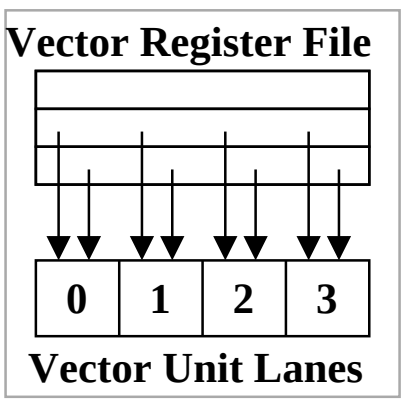
Vector Parallelism

Why SIMD Vectorization?

- Scalable parallelism (compared to ILP)
- High Throughput



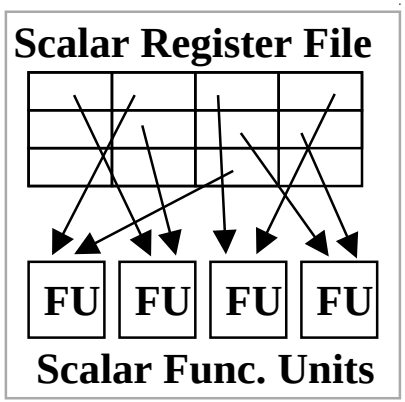
Instruction Level Parallelism(ILP)



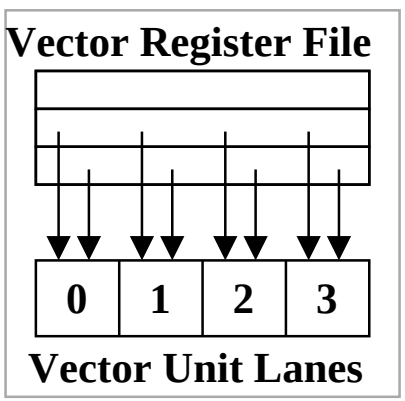
Vector Parallelism

Why SIMD Vectorization?

- Scalable parallelism (compared to ILP)
- High Throughput
- Widely adopted since the 90s



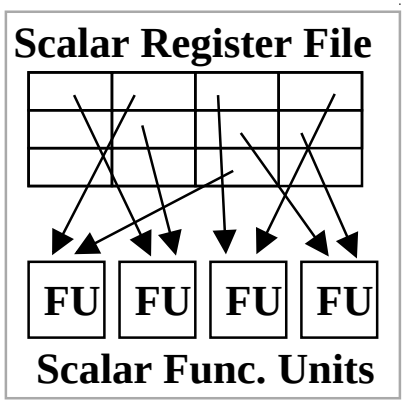
Instruction Level Parallelism(ILP)



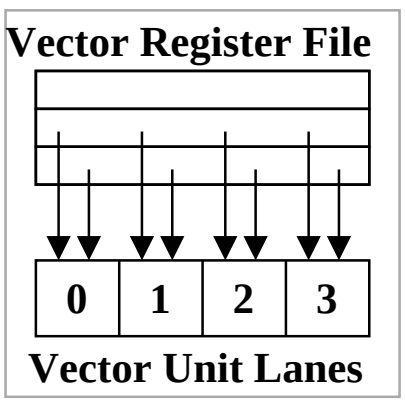
Vector Parallelism

Why SIMD Vectorization?

- Scalable parallelism (compared to ILP)
- High Throughput
- Widely adopted since the 90s
- Vector generation not done in hardware



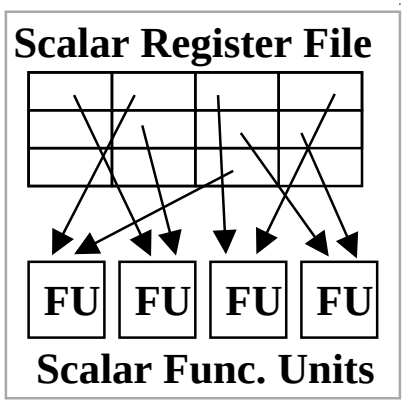
Instruction Level Parallelism(ILP)



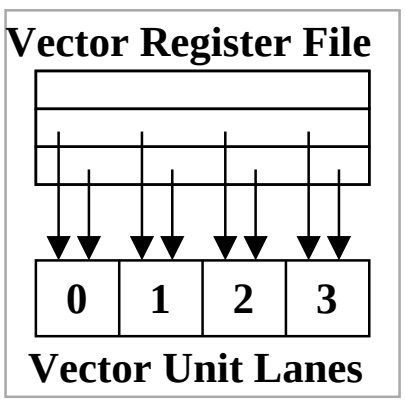
Vector Parallelism

Why SIMD Vectorization?

- Scalable parallelism (compared to ILP)
- High Throughput
- Widely adopted since the 90s
- Vector generation not done in hardware
- Low-level coding or vectorizing compiler



Instruction Level Parallelism(ILP)



Vector Parallelism

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP
- SLP and the loop-vectorizer complement each other:

SLP Straight-Line Code Vectorizer

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

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP
- SLP and the loop-vectorizer complement each other:
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed
- It is missing features present in the Loop vectorizer (e.g., Interleaved Loads, Predication)

SLP Straight-Line Code Vectorizer

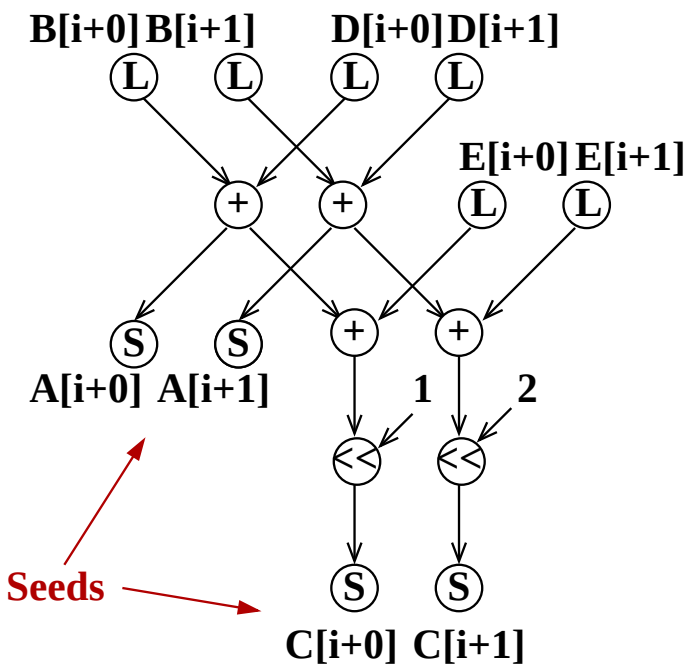
- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP
- SLP and the loop-vectorizer complement each other:
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed
- It is missing features present in the Loop vectorizer (e.g., Interleaved Loads, Predication)
  - Usually run SLP after the Loop Vectorizer

SLP is usually good enough

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```

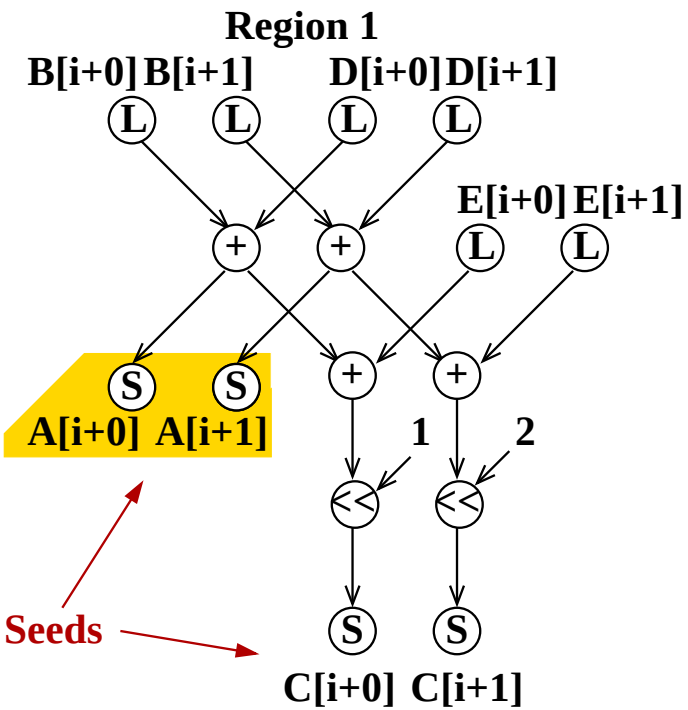
SLP is usually good enough

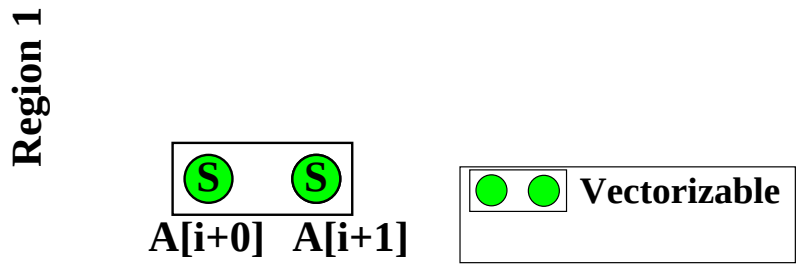
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```



SLP is usually good enough

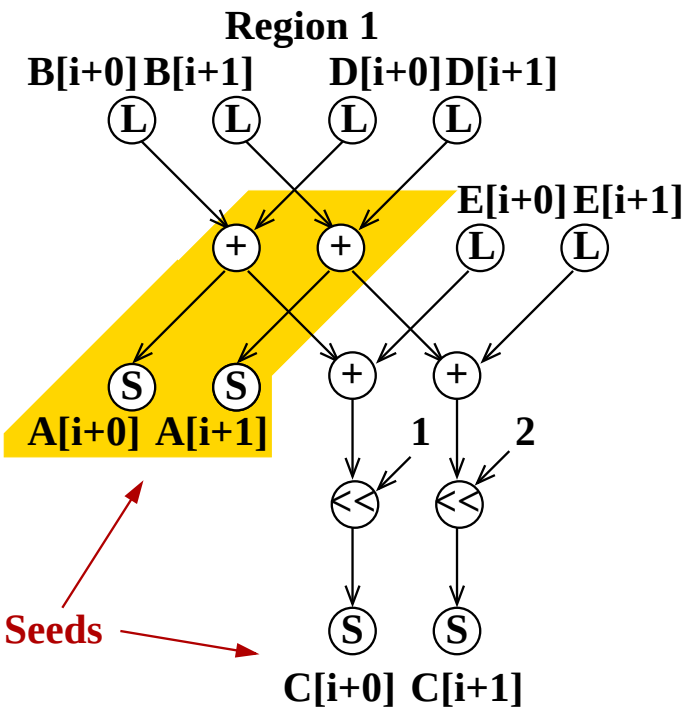
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```

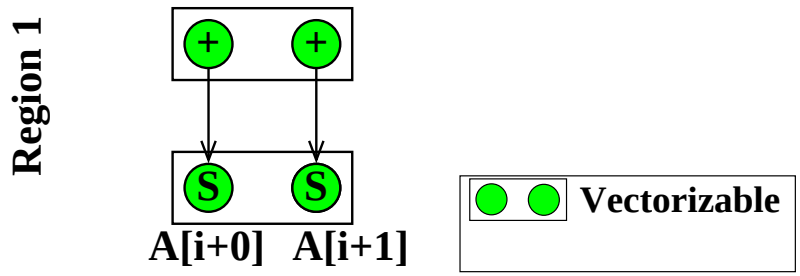




SLP is usually good enough

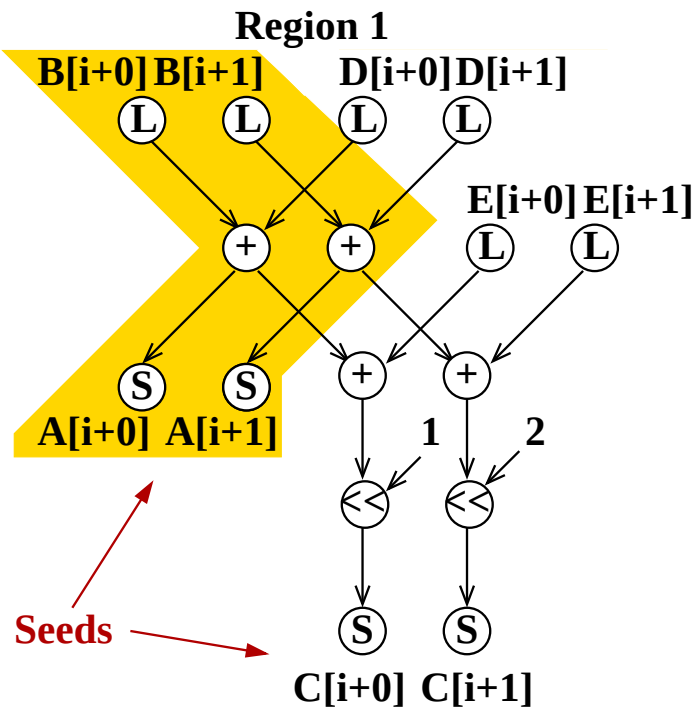
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```

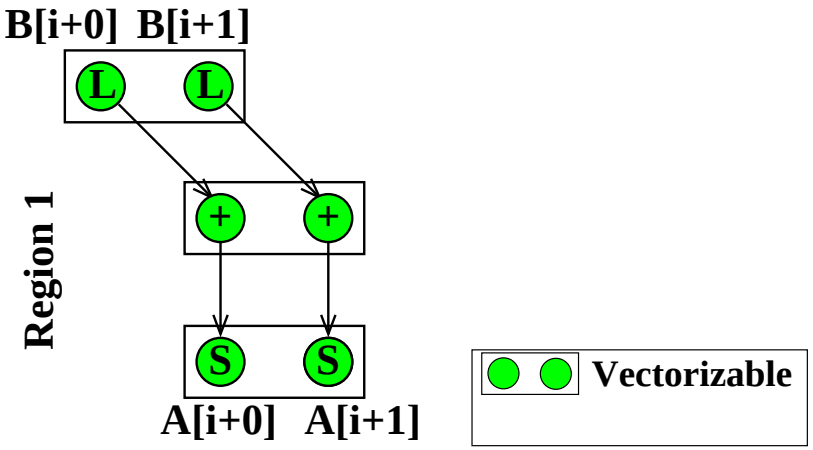




SLP is usually good enough

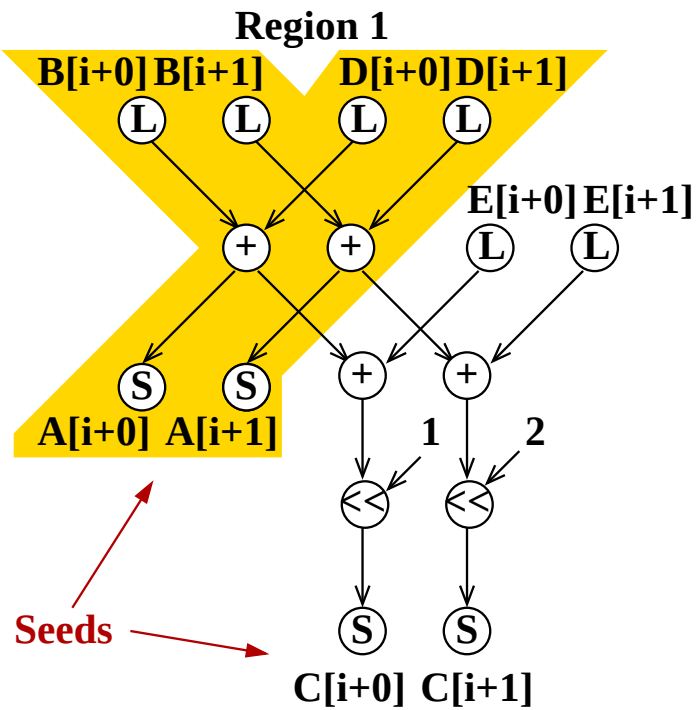
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```

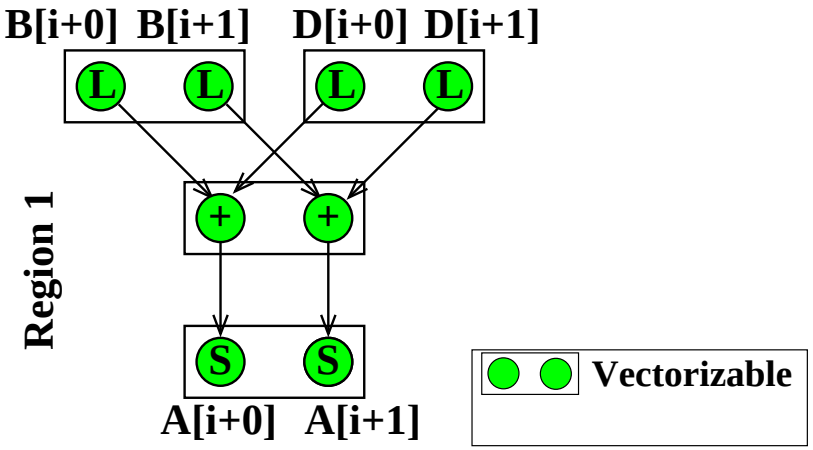




SLP is usually good enough

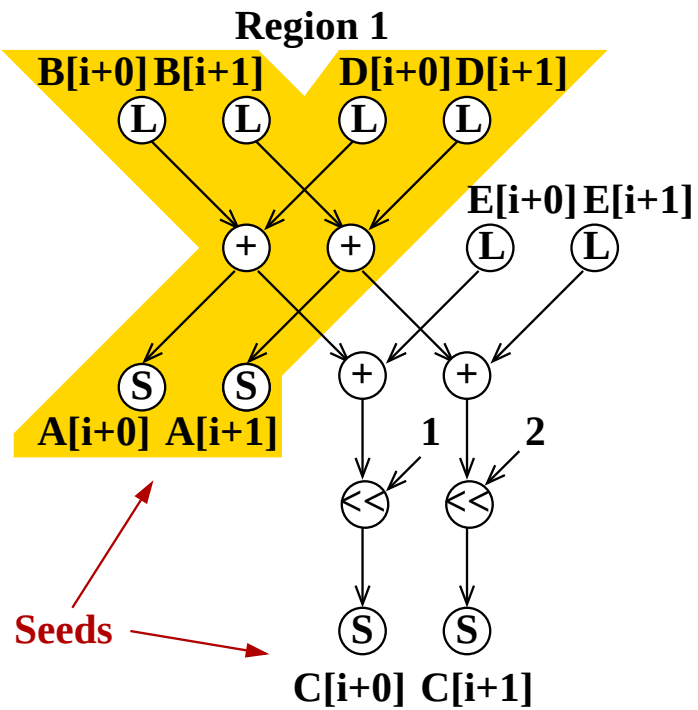
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```

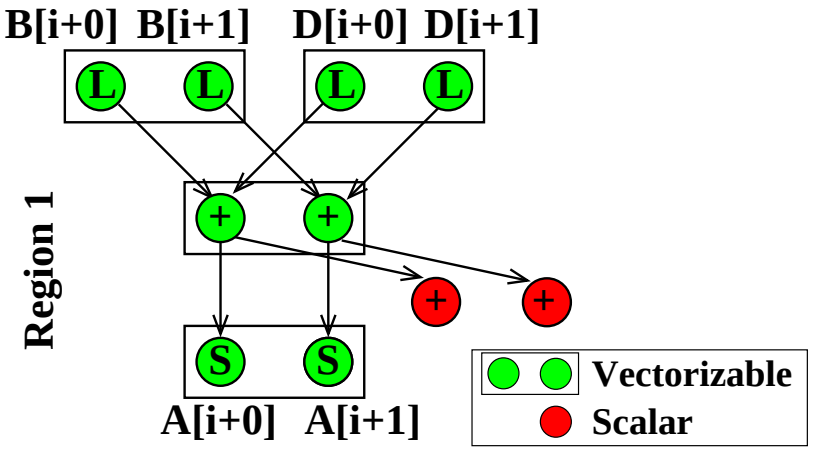




SLP is usually good enough

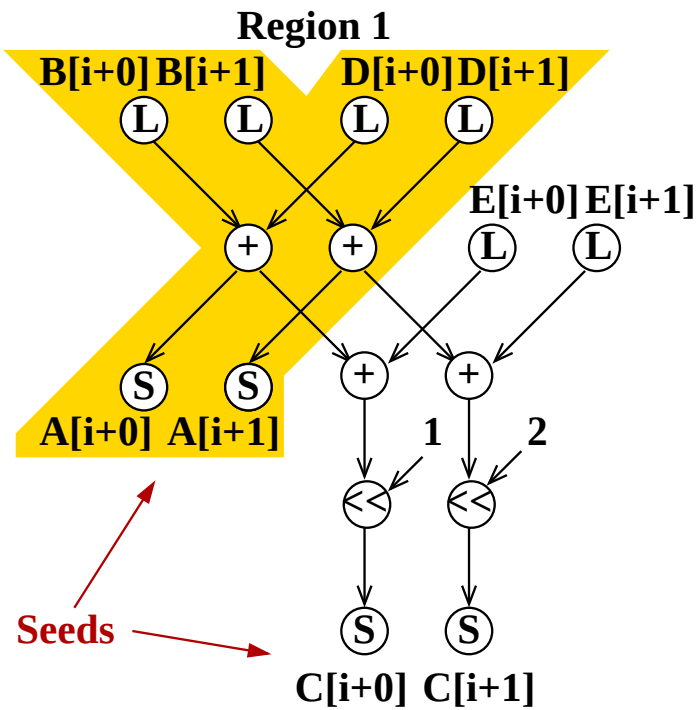
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```

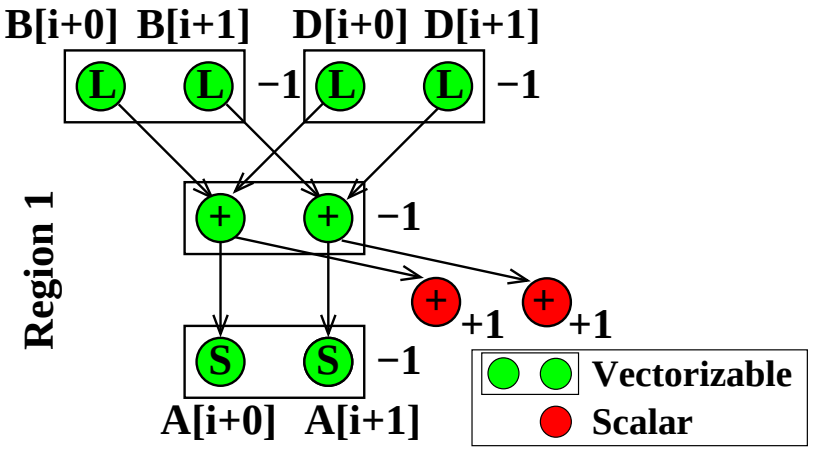




SLP is usually good enough

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```

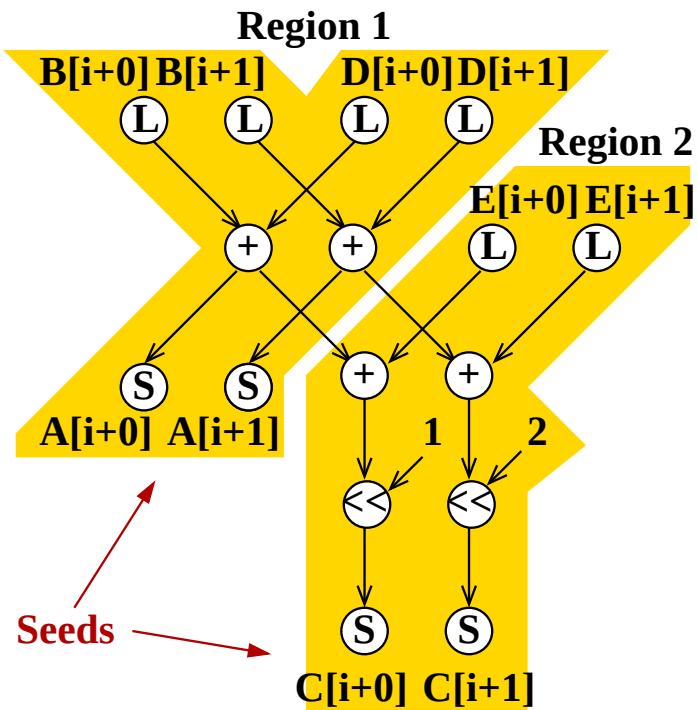


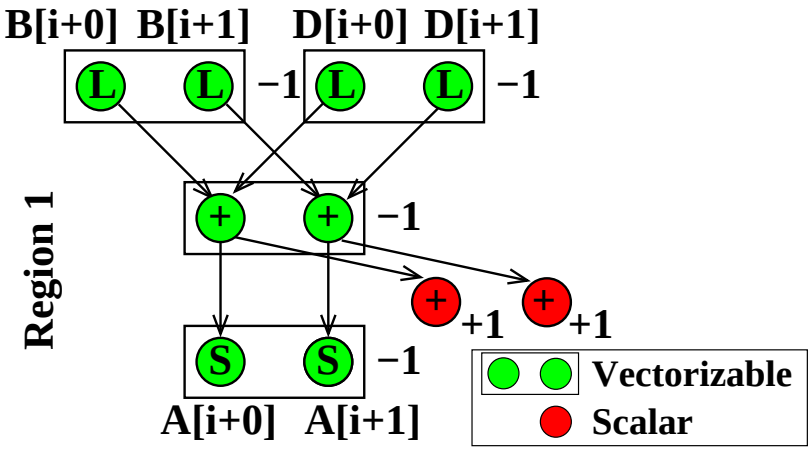


Total Cost: −2 **VECTORIZED !**

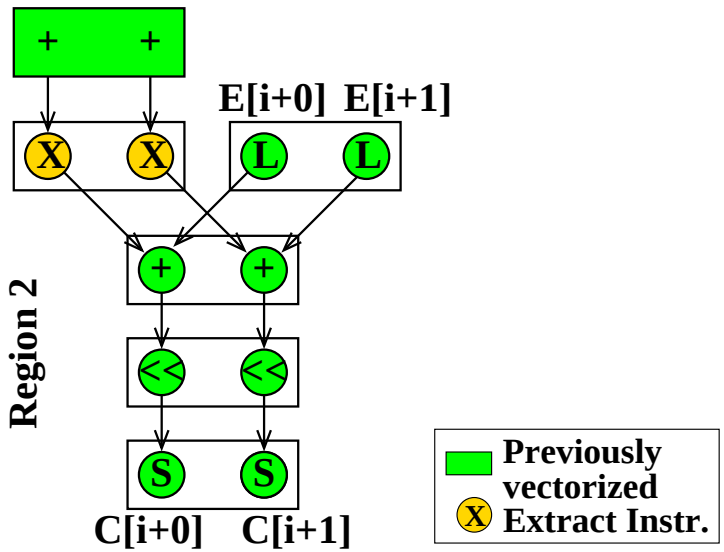
SLP is usually good enough

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```



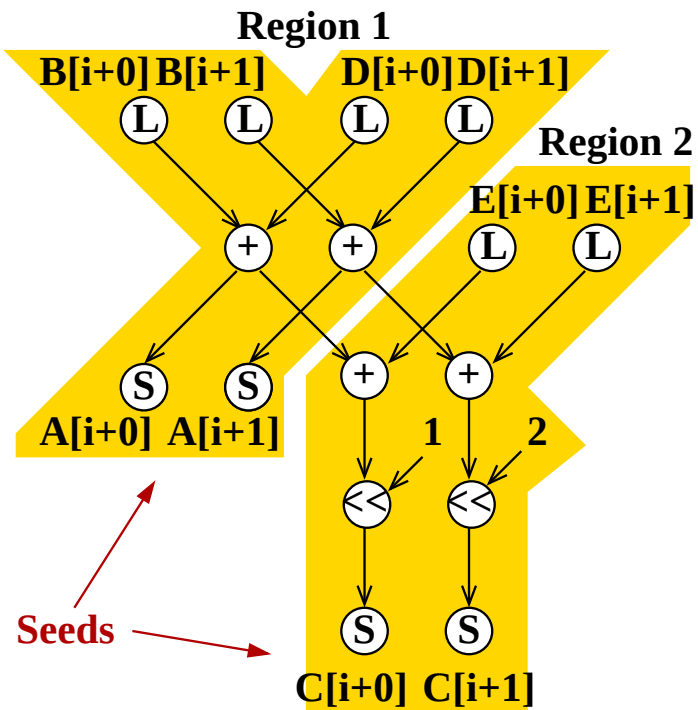


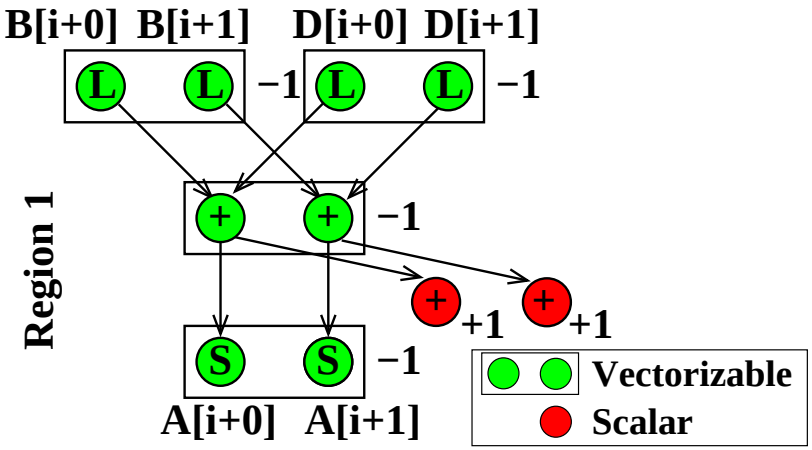
Total Cost: −2 **VECTORIZED !**



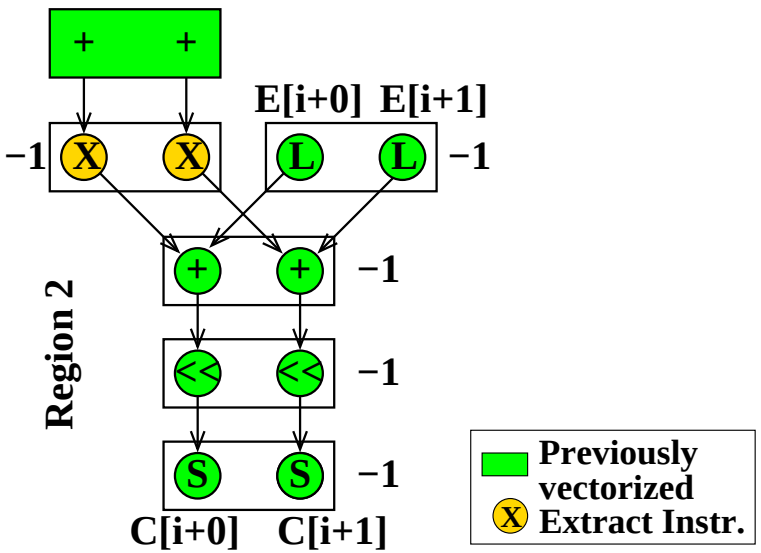
SLP is usually good enough

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```





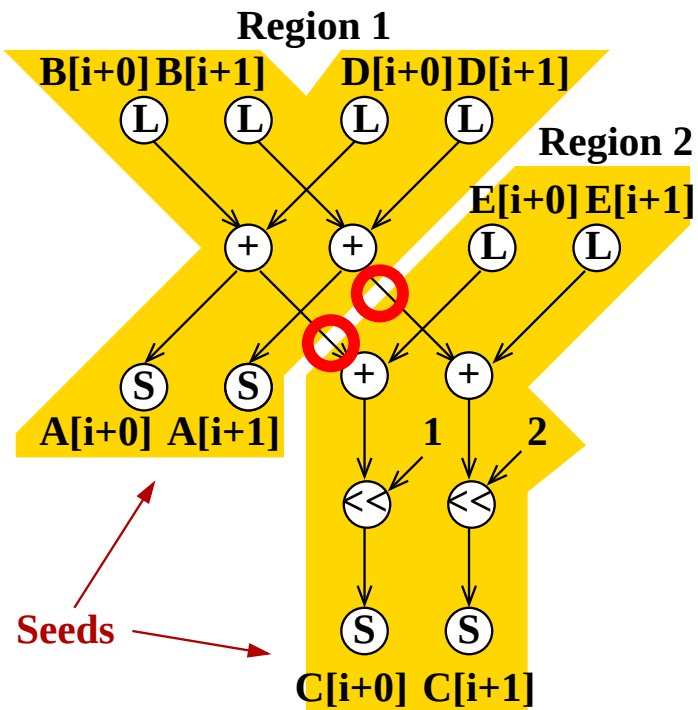
Total Cost: −2 VECTORIZED !

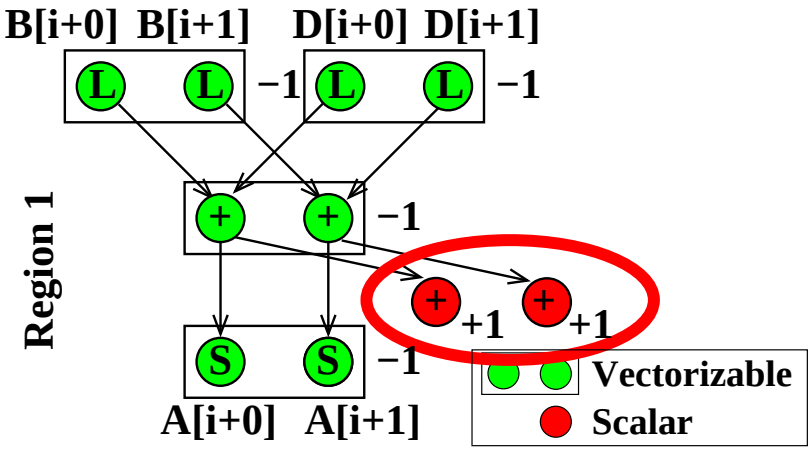


Total Cost: −5 VECTORIZED !

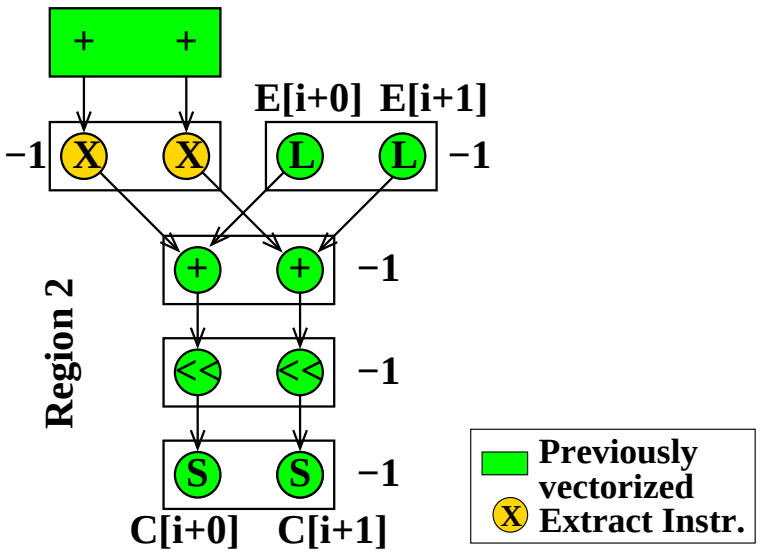
SLP is usually good enough

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```





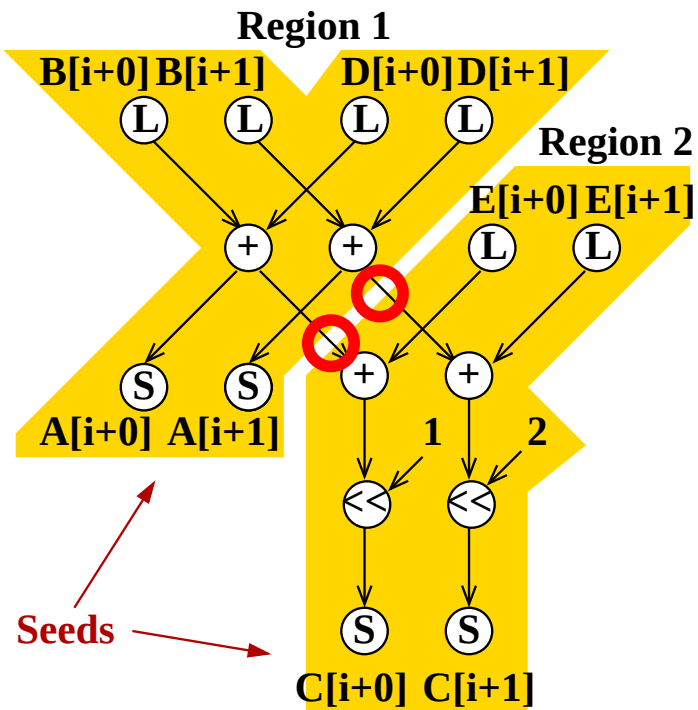
Total Cost: −2 VECTORIZED !

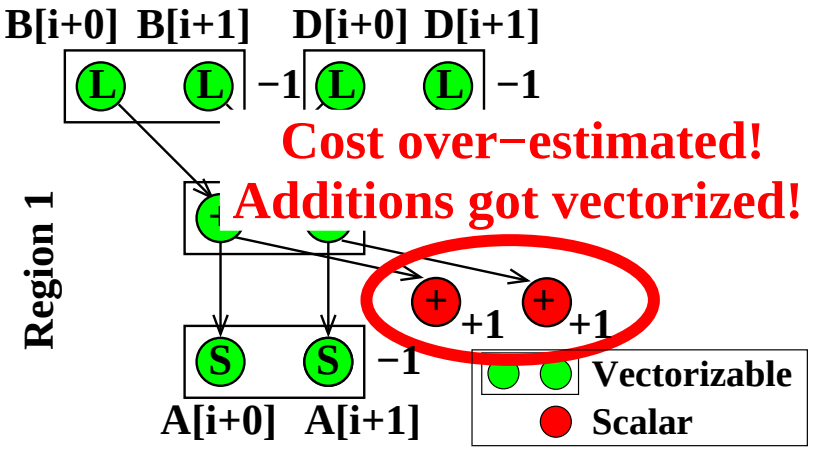


Total Cost: −5 VECTORIZED !

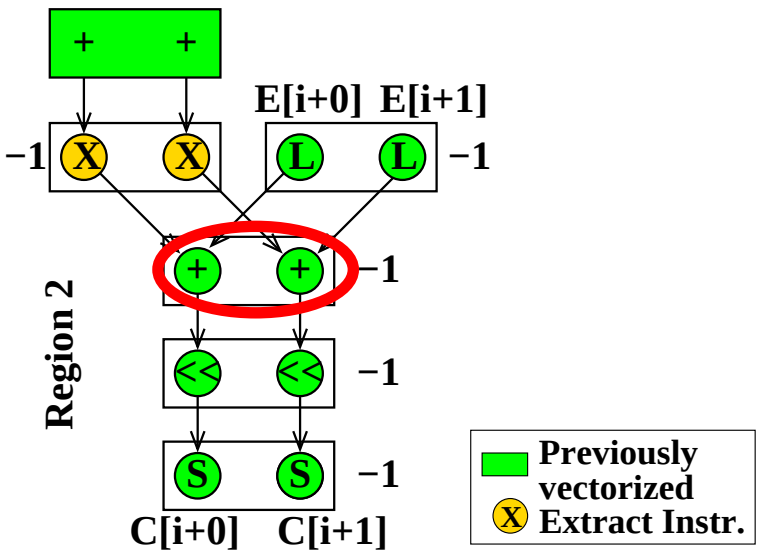
SLP is usually good enough

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = (tmp1 + E[i+0]) << 1;
C[i+1] = (tmp2 + E[i+1]) << 2;
```



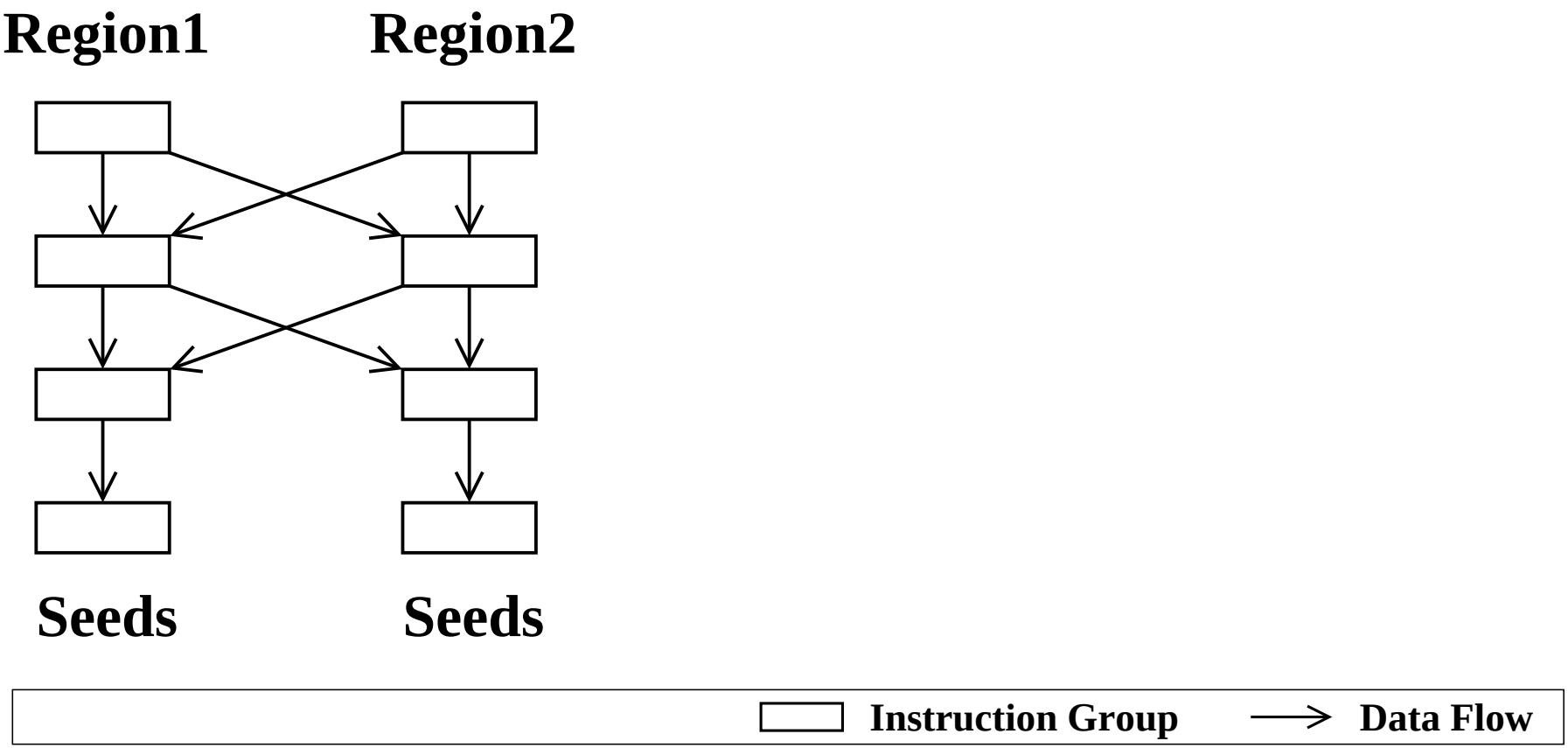


Total Cost: −2 VECTORIZED !

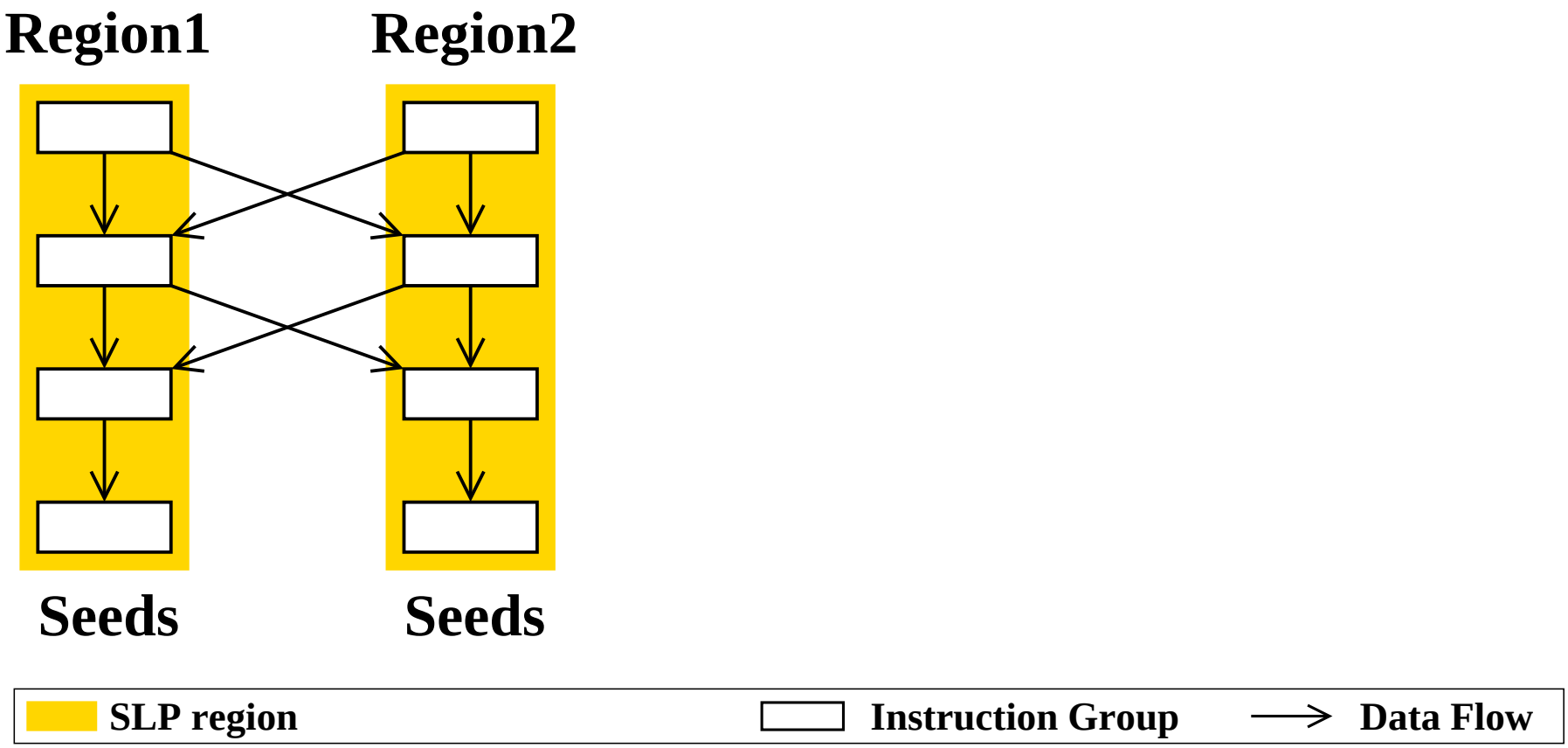


Total Cost: −5 VECTORIZED !

Super-Graph SLP: Larger Unified Region

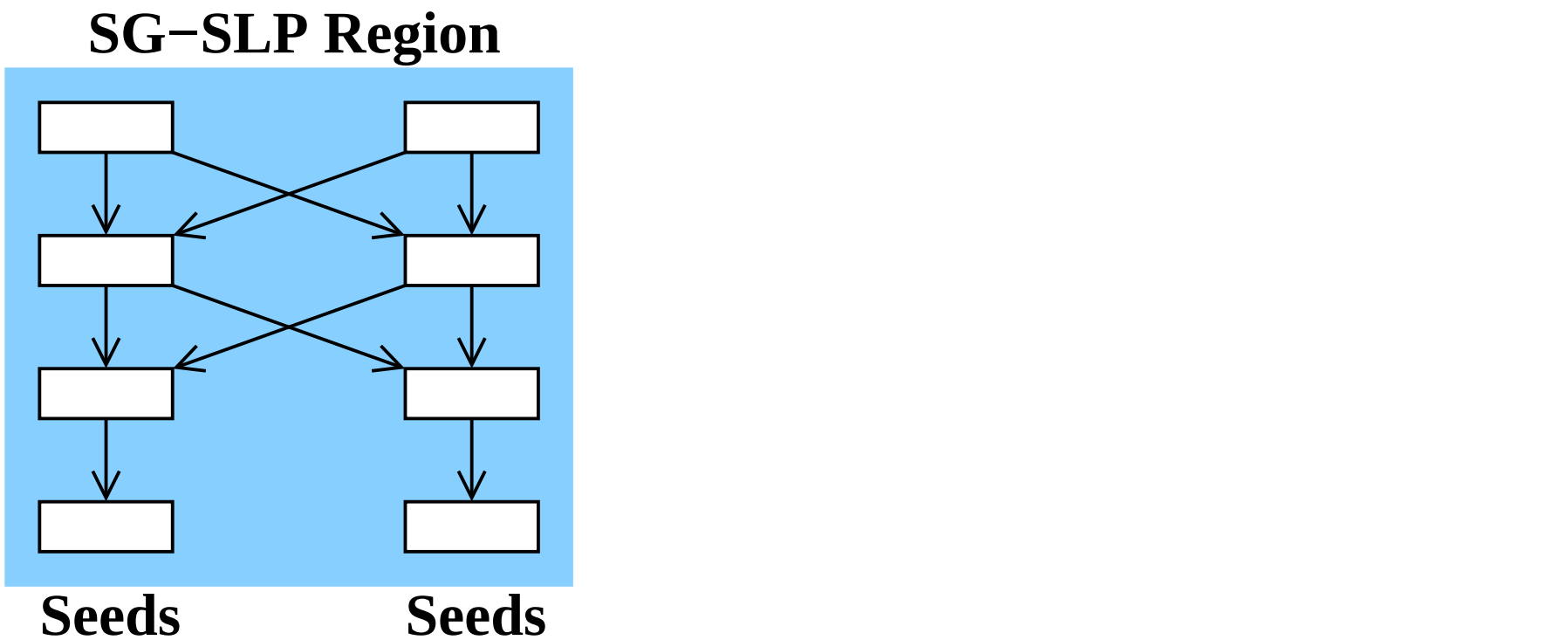


Super-Graph SLP: Larger Unified Region



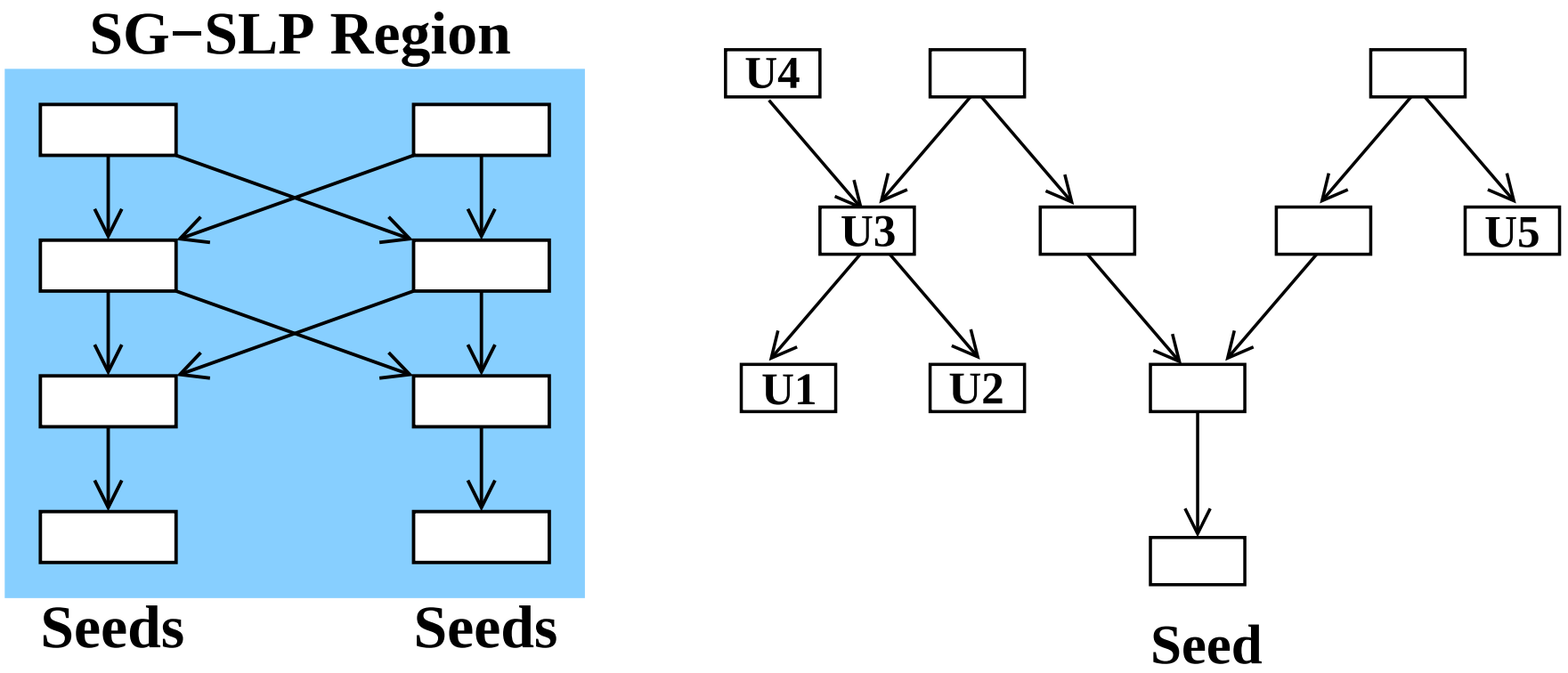
1. Cross-region dependencies

Super-Graph SLP: Larger Unified Region



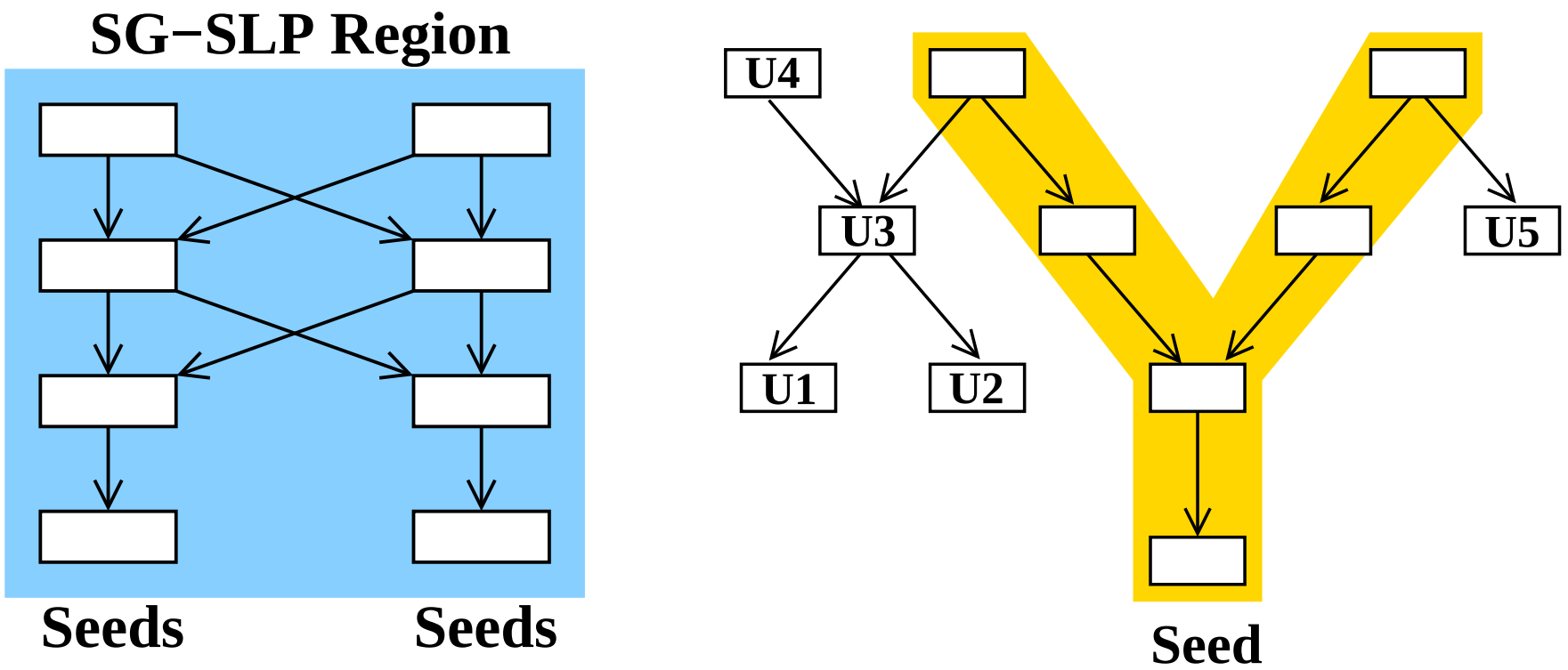
1. Cross-region dependencies

Super-Graph SLP: Larger Unified Region



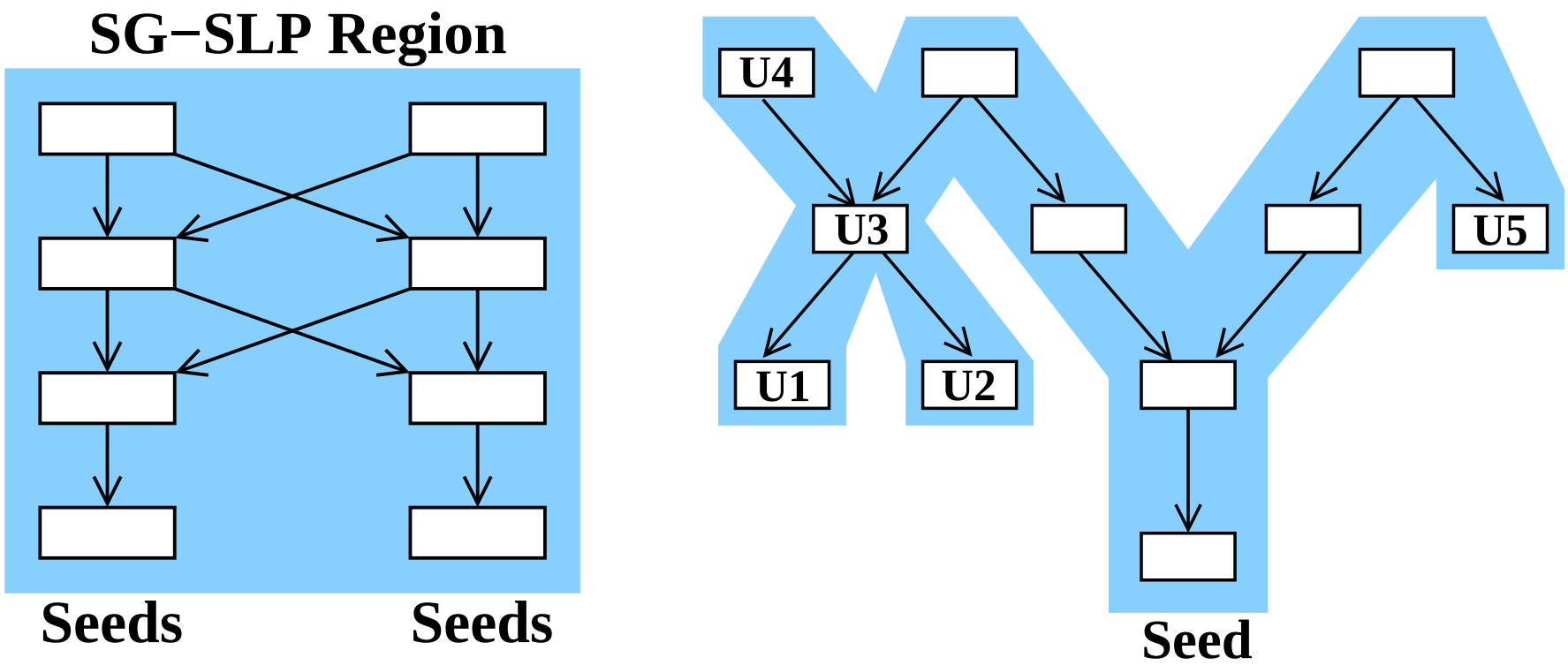
1. Cross-region dependencies

Super-Graph SLP: Larger Unified Region



1. Cross-region dependencies
2. Unreachable instructions by SLP

Super-Graph SLP: Larger Unified Region



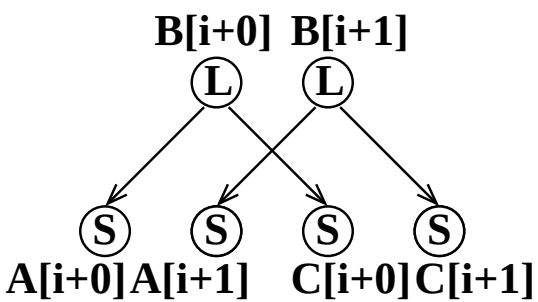
1. Cross-region dependencies
2. Unreachable instructions by SLP

1. SLP Fails due to Cross-Region Dependencies

```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

1. SLP Fails due to Cross-Region Dependencies

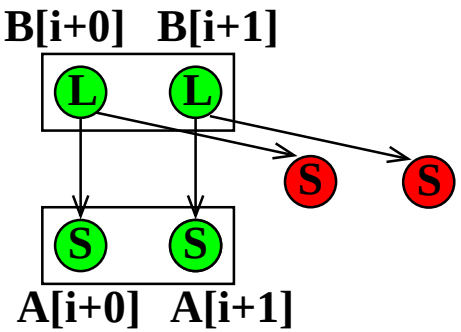
```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

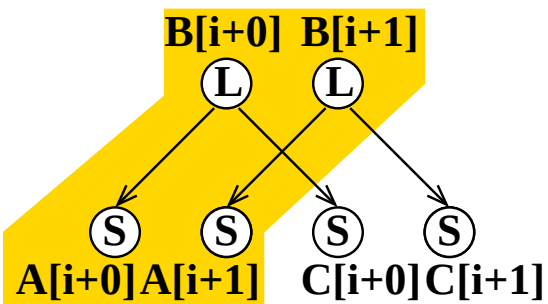


1. SLP Fails due to Cross-Region Dependencies

```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

SLP

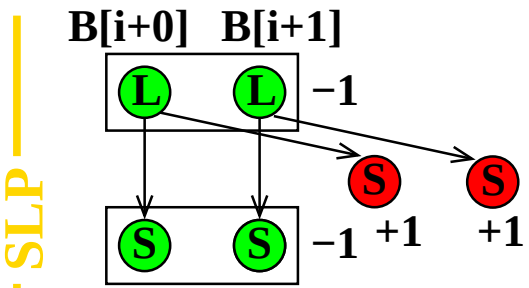




1. SLP Fails due to Cross-Region Dependencies

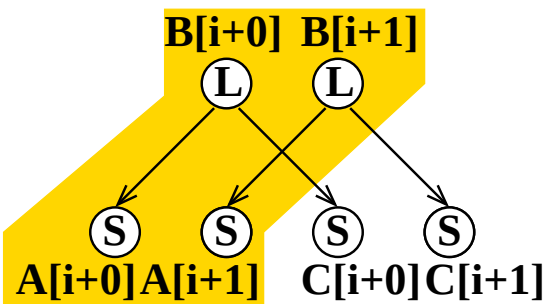
```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

SLP



Total Cost: 0

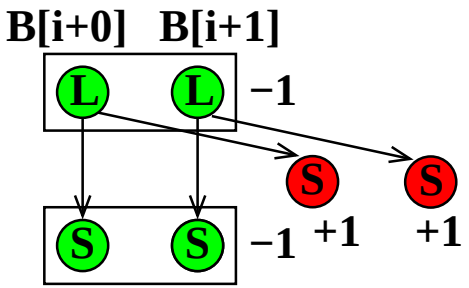
NOT VECTORIZED



1. SLP Fails due to Cross-Region Dependencies

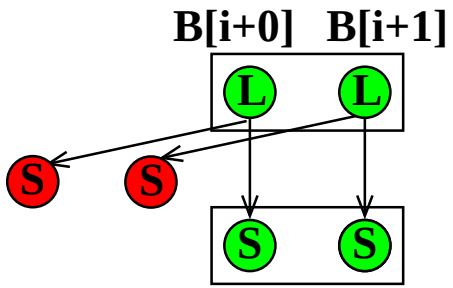
```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

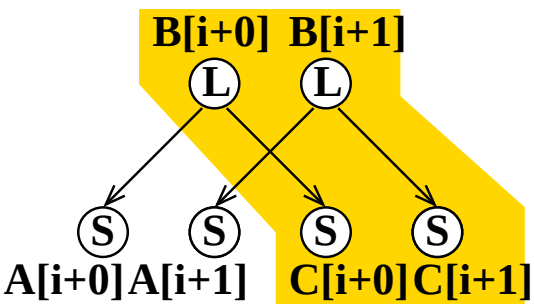
SLP



Total Cost: 0

NOT VECTORIZED

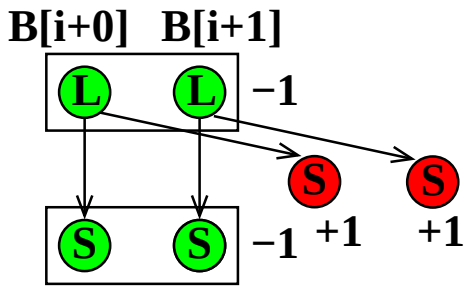




1. SLP Fails due to Cross-Region Dependencies

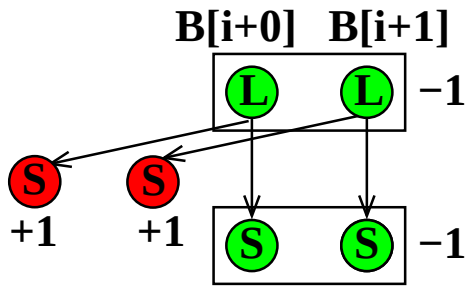
```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

SLP



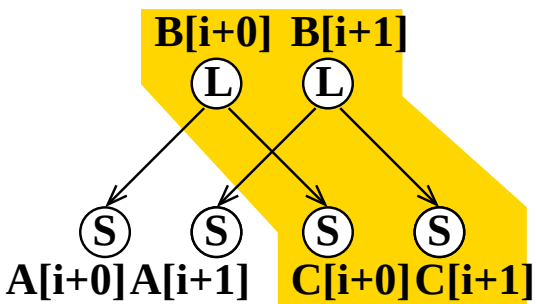
Total Cost: 0

NOT VECTORIZED



Total Cost: 0

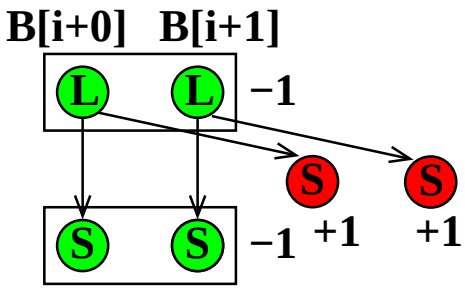
NOT VECTORIZED



1. SLP Fails due to Cross-Region Dependencies

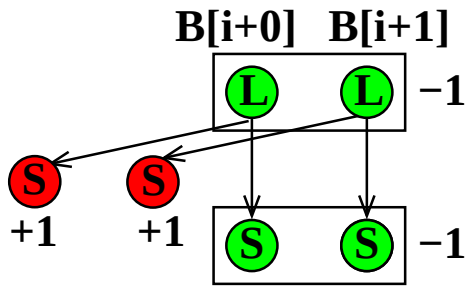
```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

SLP



Total Cost: 0

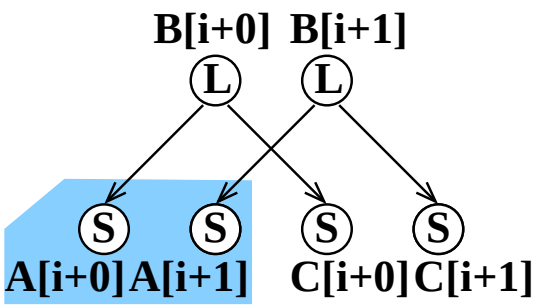
NOT VECTORIZED

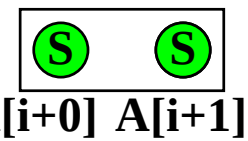


Total Cost: 0

NOT VECTORIZED

SG–SLP

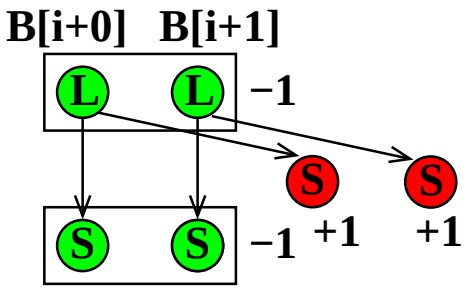




1. SLP Fails due to Cross-Region Dependencies

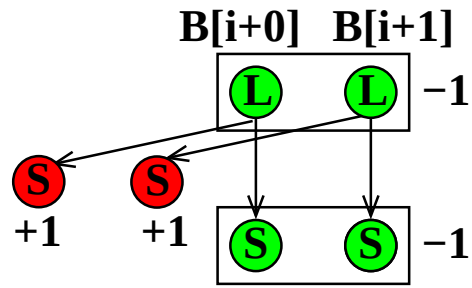
```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

SLP



Total Cost: 0

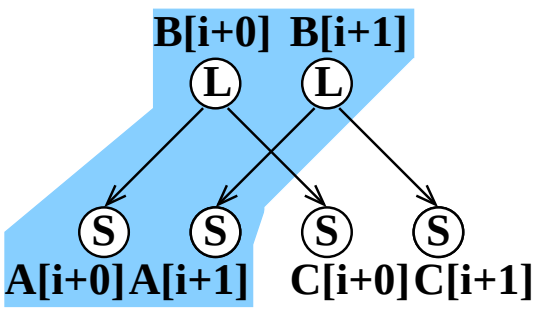
NOT VECTORIZED

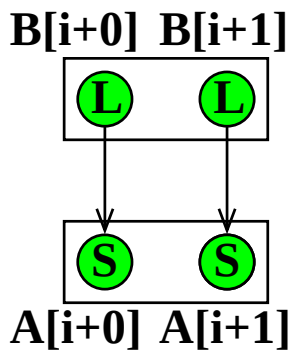


Total Cost: 0

NOT VECTORIZED

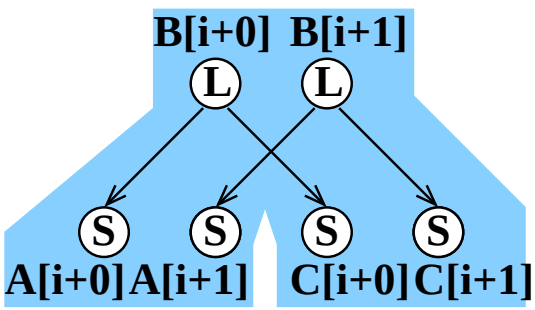
SG–SLP



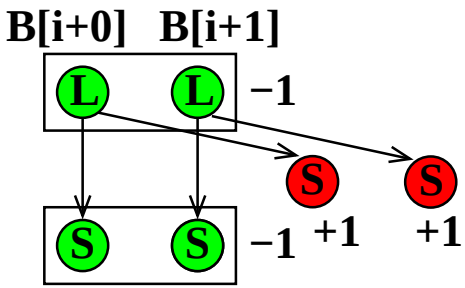


1. SLP Fails due to Cross-Region Dependencies

```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

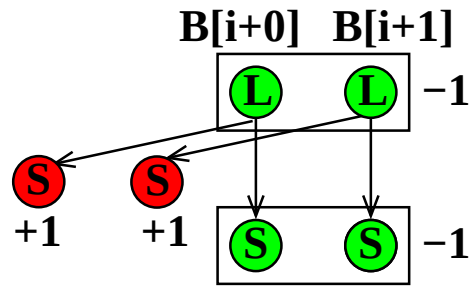


SLP



Total Cost: 0

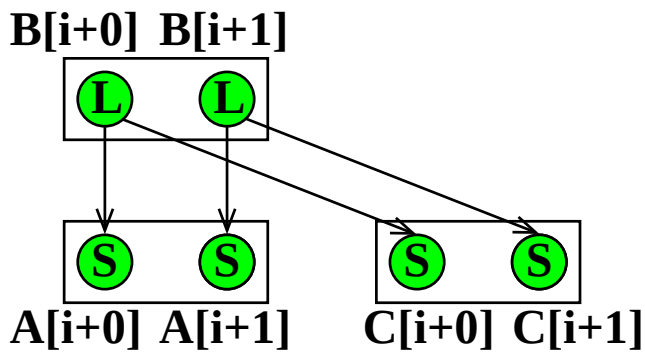
NOT VECTORIZED



Total Cost: 0

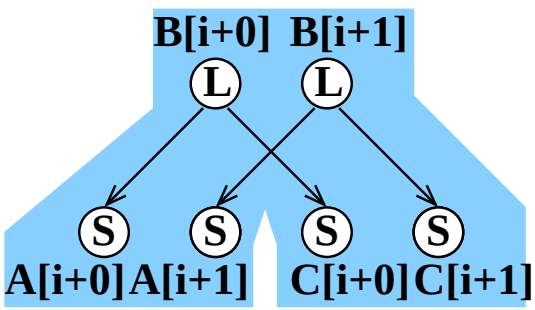
NOT VECTORIZED

SG–SLP

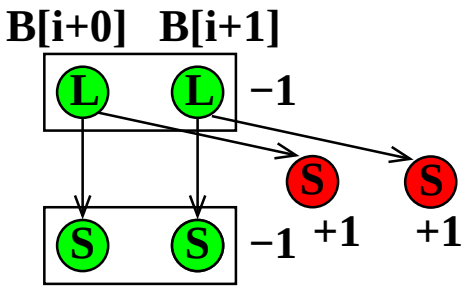


1. SLP Fails due to Cross-Region Dependencies

```
long tmp1, tmp2, A[], B[], C[];
tmp1 = B[i+0];
tmp2 = B[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+0] = tmp1;
C[i+1] = tmp2;
```

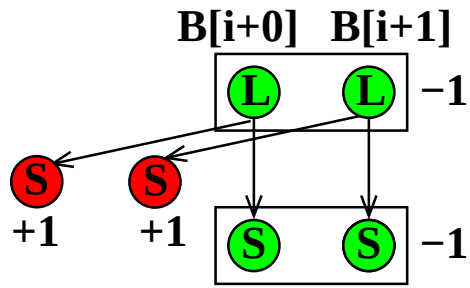


SLP



Total Cost: 0

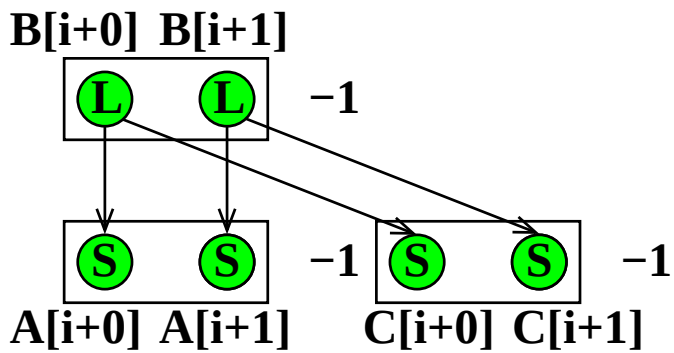
NOT VECTORIZED



Total Cost: 0

NOT VECTORIZED

SG–SLP



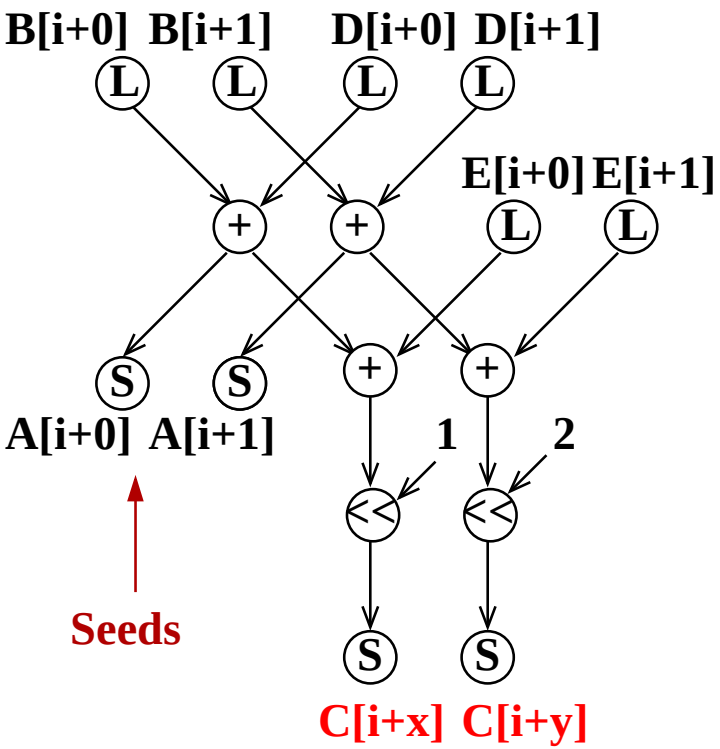
Total Cost: −3 VECTORIZED !

2. SLP Fails due to Unreachable Instructions

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```

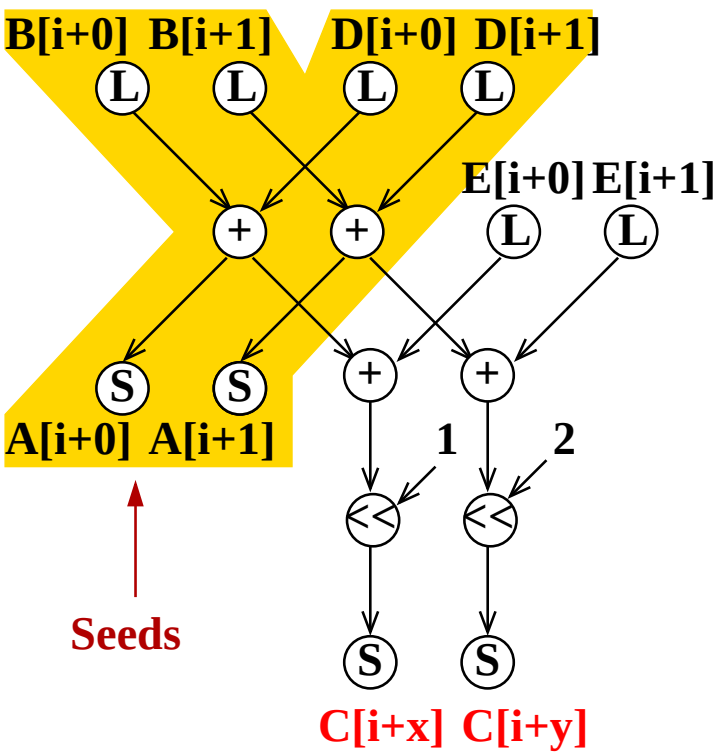
2. SLP Fails due to Unreachable Instructions

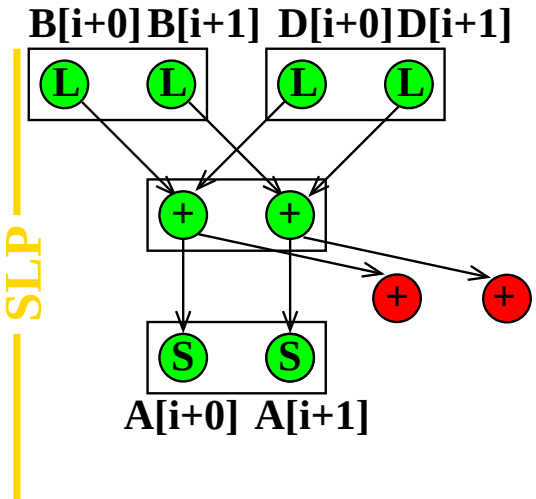
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```



2. SLP Fails due to Unreachable Instructions

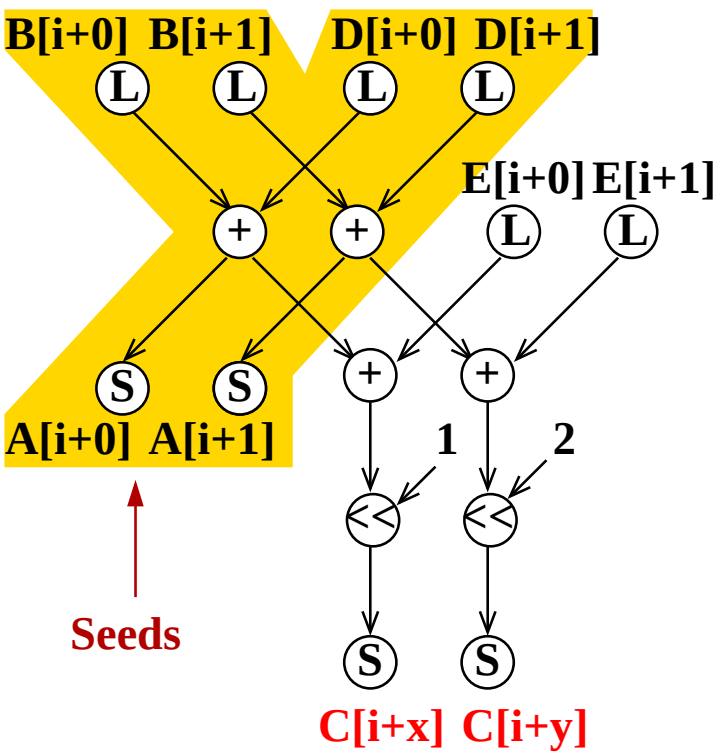
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```

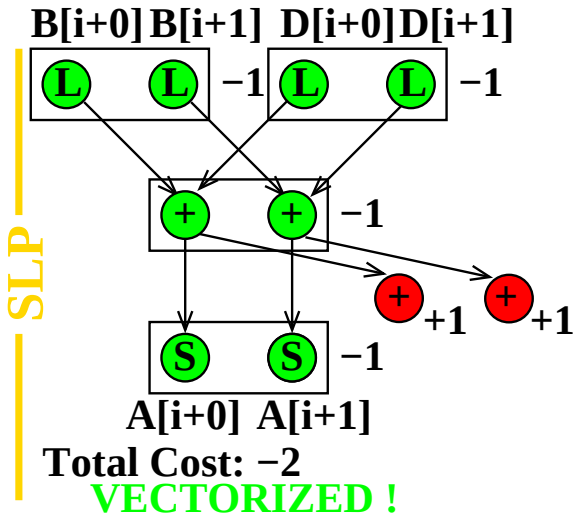




2. SLP Fails due to Unreachable Instructions

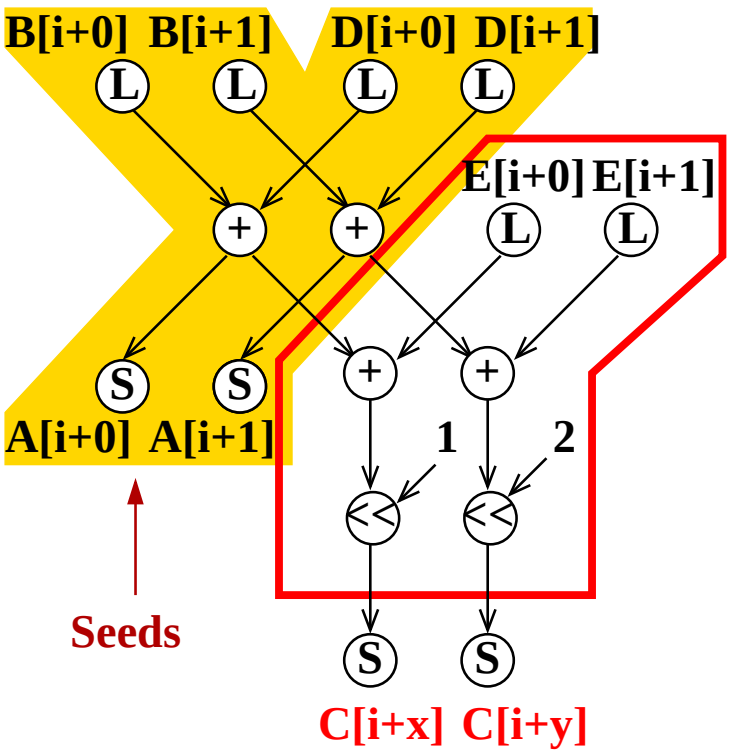
```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```

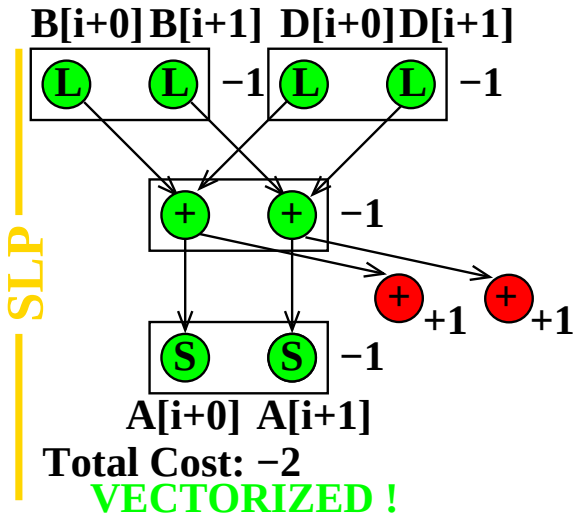


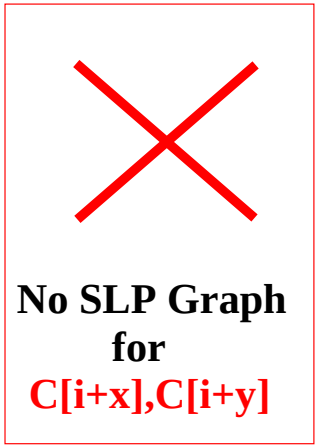


2. SLP Fails due to Unreachable Instructions

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```

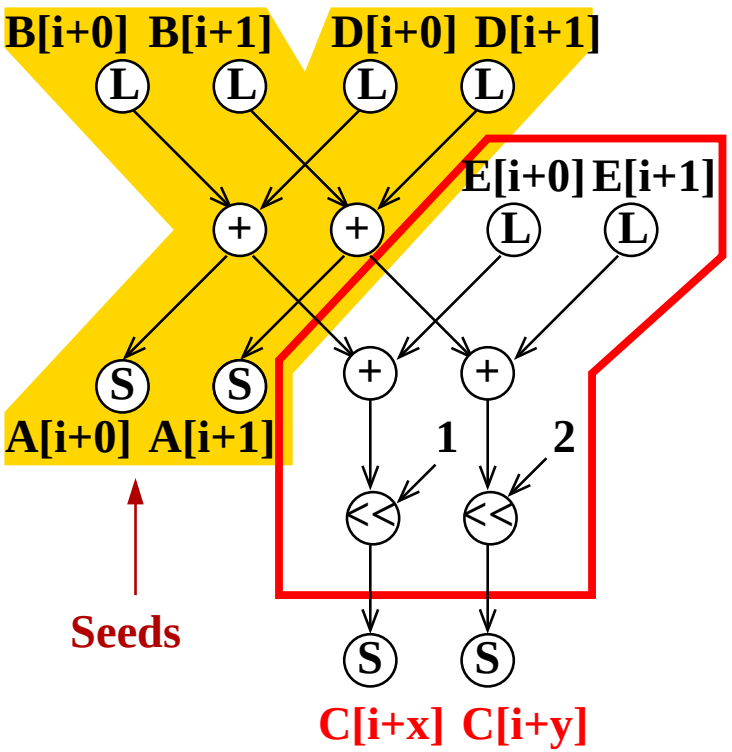


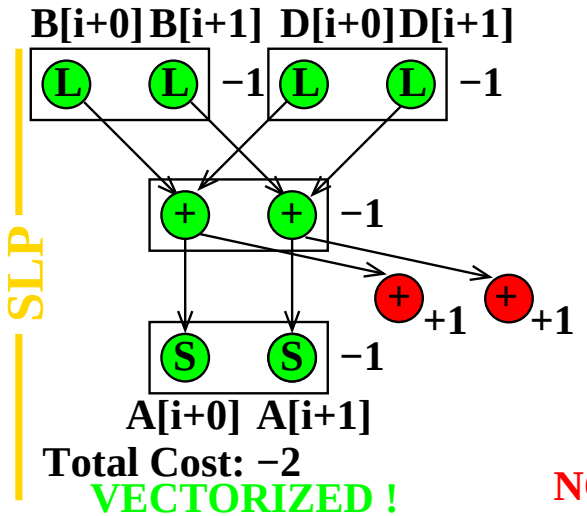


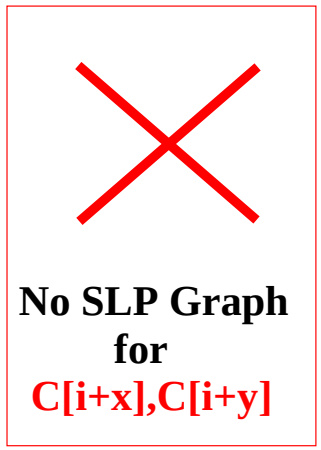


2. SLP Fails due to Unreachable Instructions

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```

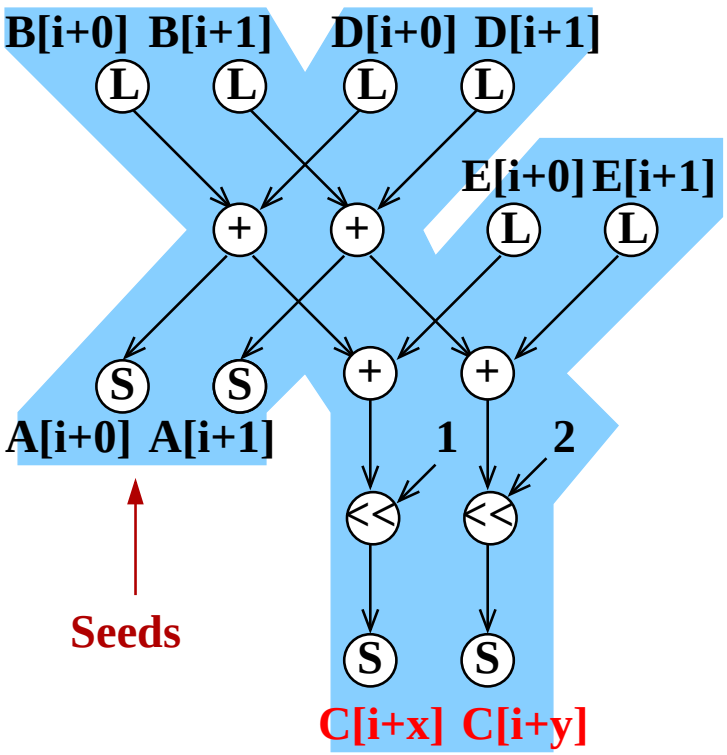




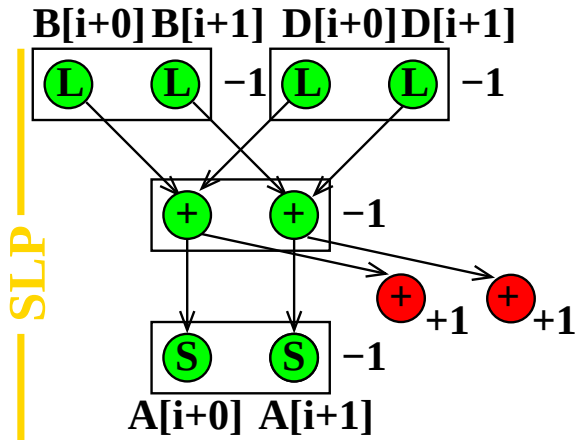


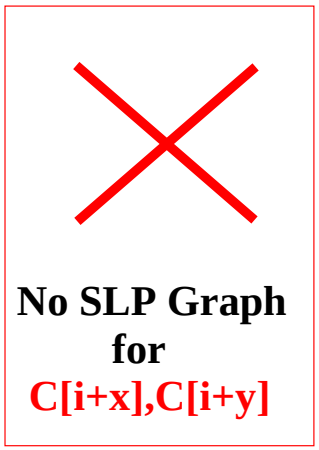
2. SLP Fails due to Unreachable Instructions

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```

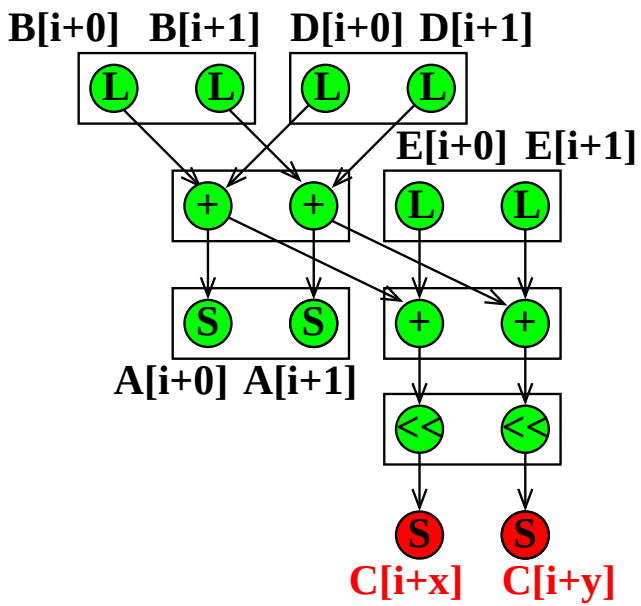


SLP



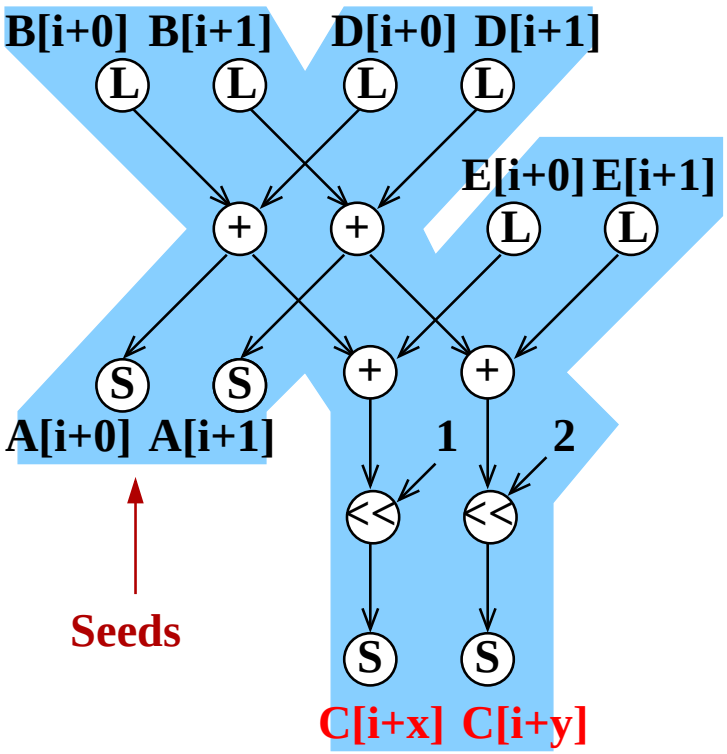


SG−SLP

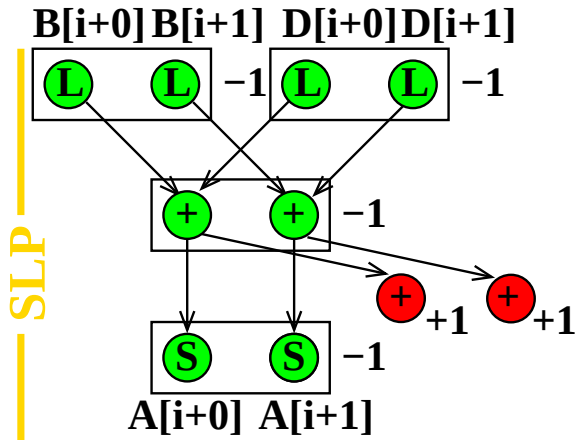


2. SLP Fails due to Unreachable Instructions

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```

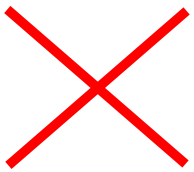


SLP



Total Cost: −2

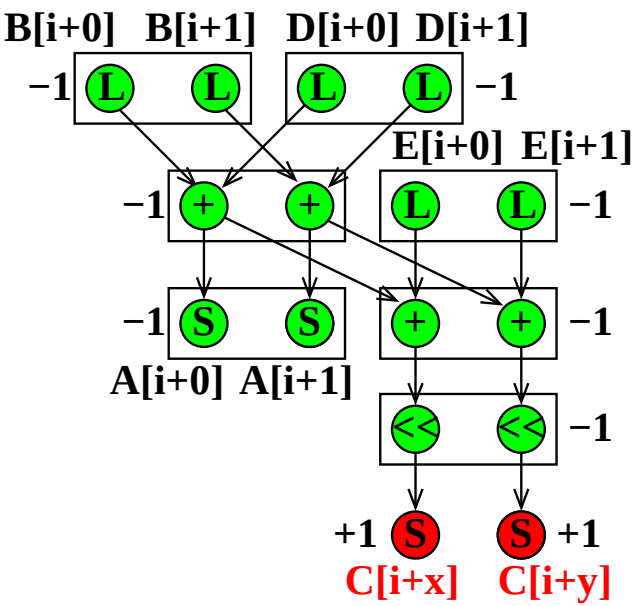
VECTORIZED !



No SLP Graph for C[i+x],C[i+y]

NOT VECTORIZED

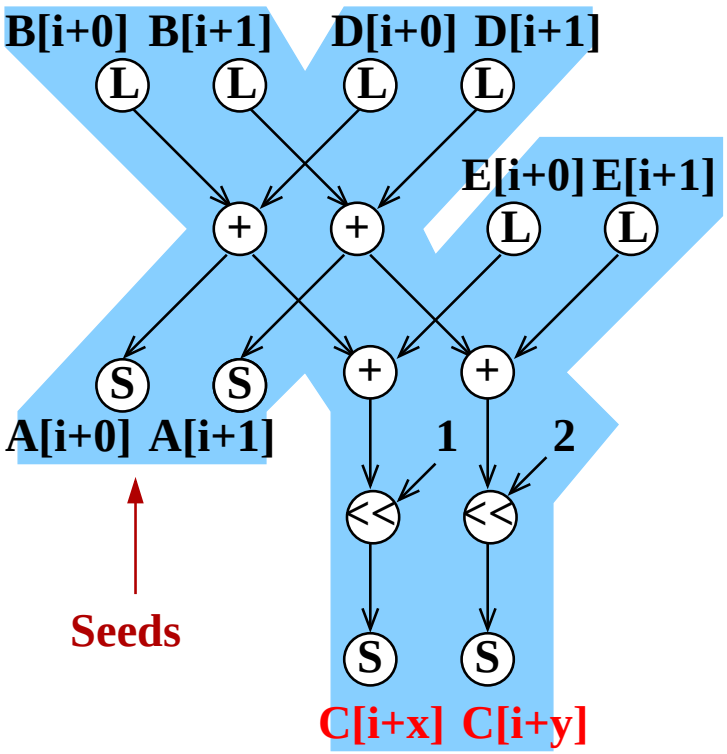
SG−SLP



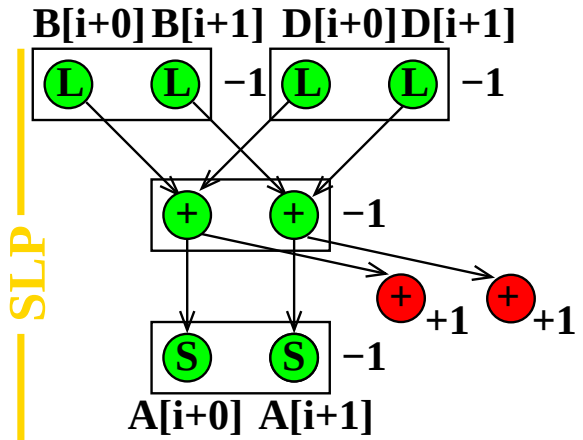
Total Cost: −5 VECTORIZED !

2. SLP Fails due to Unreachable Instructions

```
long tmp1, tmp2, A[], B[], C[], D[], E[];
tmp1 = B[i+0] + D[i+0];
tmp2 = B[i+1] + D[i+1];
A[i+0] = tmp1;
A[i+1] = tmp2;
C[i+x] = (tmp1 + E[i+0]) << 1;
C[i+y] = (tmp2 + E[i+1]) << 2;
```

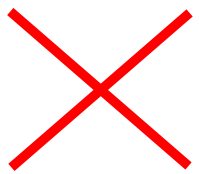


SLP



Total Cost: −2

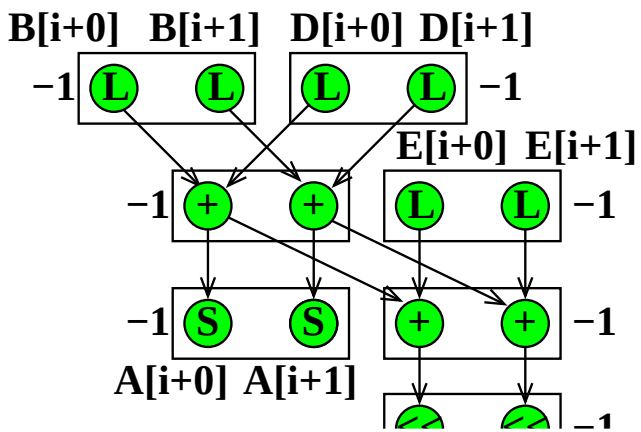
VECTORIZED !



No SLP Graph for C[i+x],C[i+y]

NOT VECTORIZED

SG−SLP

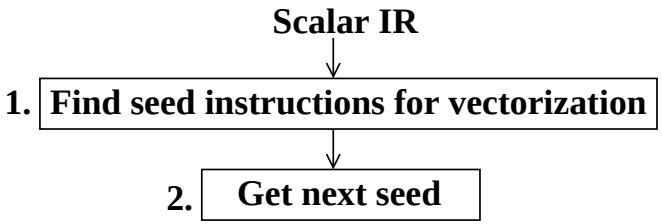


BETTER PERFORMANCE!

C[i+x] C[i+y]

Total Cost: −5 VECTORIZED !

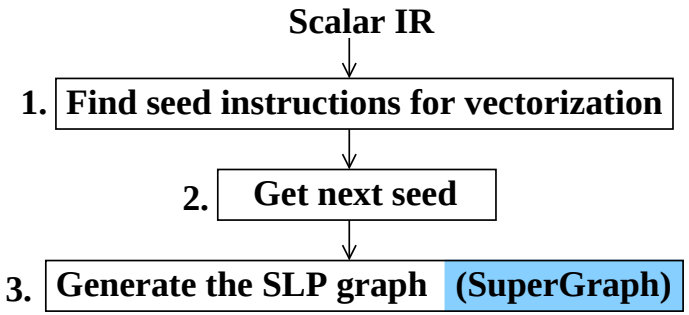
SG-SLP Algorithm



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

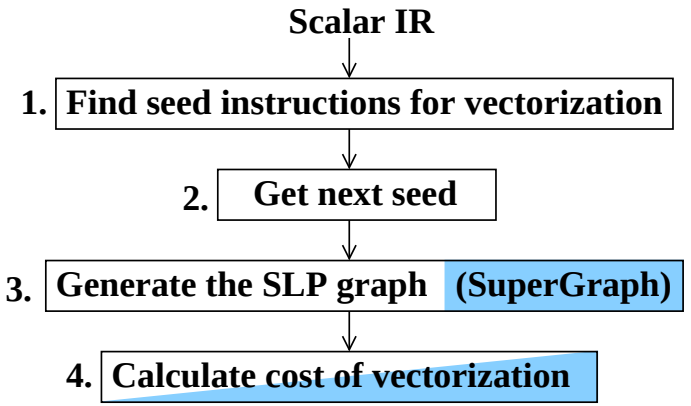
SG-SLP Algorithm

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



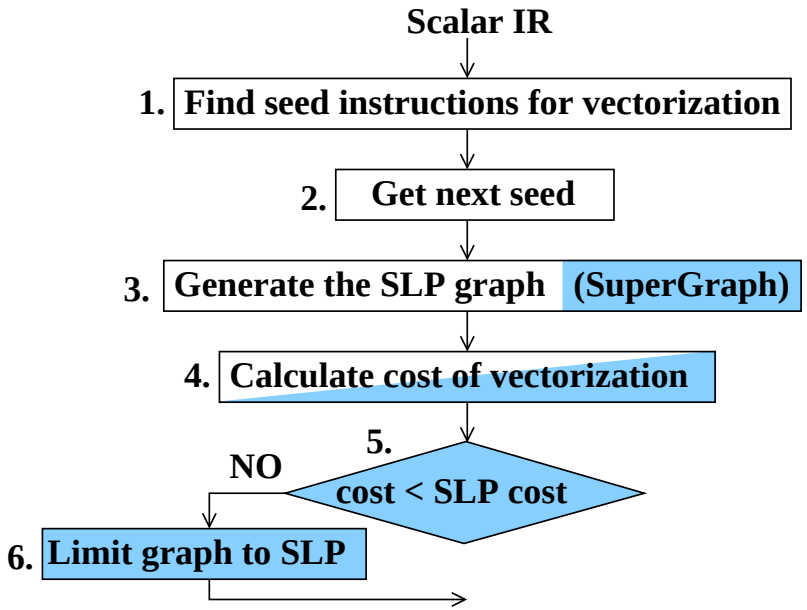
SG-SLP Algorithm

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



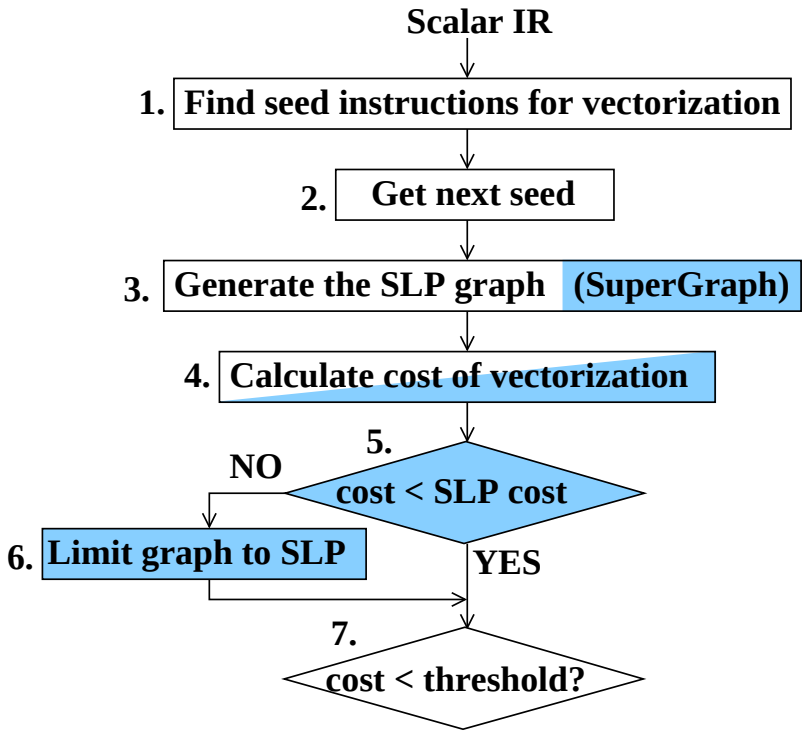
SG-SLP Algorithm

- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check SG-SLP vs SLP profitability



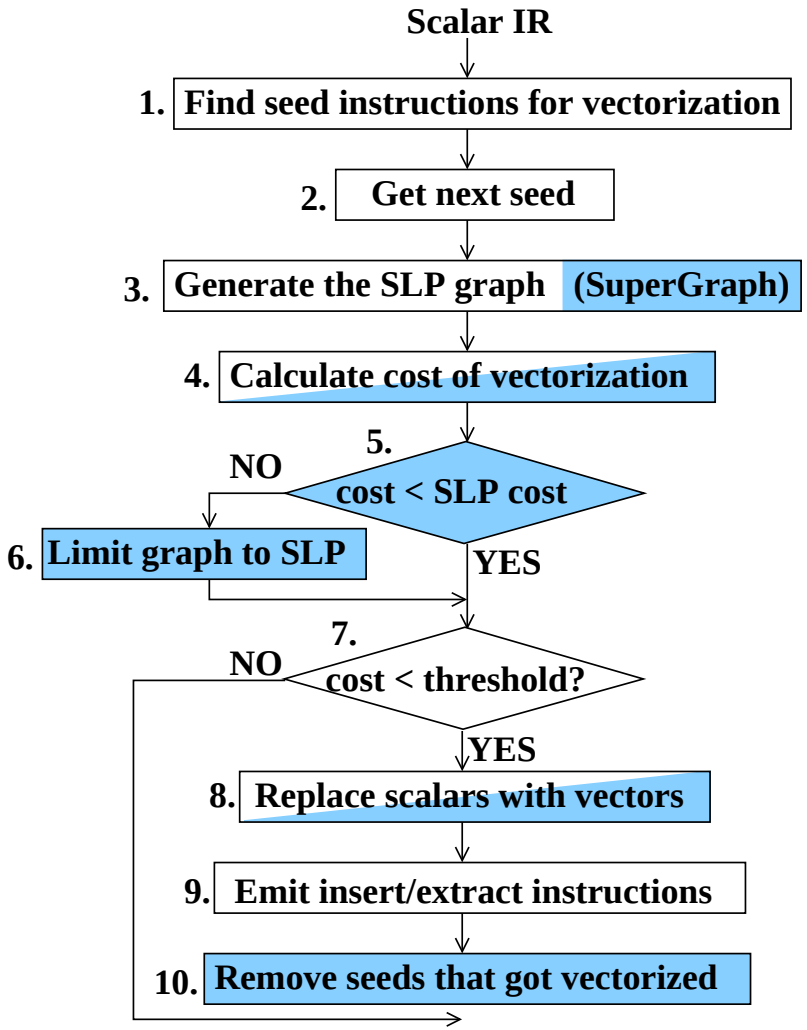
SG-SLP Algorithm

- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check SG-SLP vs SLP profitability
- Check overall profitability



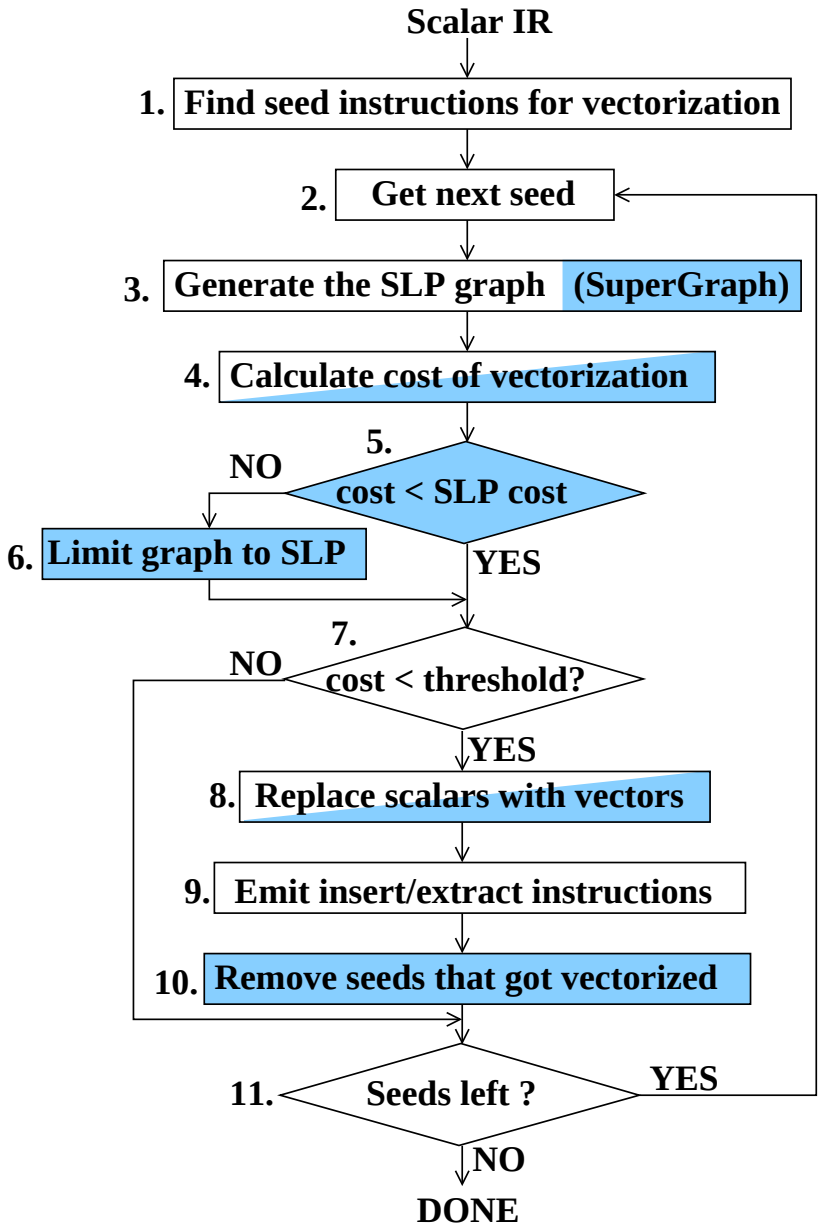
SG-SLP Algorithm

- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check SG-SLP vs SLP profitability
- Check overall profitability
- Generate vector code



SG-SLP Algorithm

- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check SG-SLP vs SLP profitability
- Check overall profitability
- Generate vector code
- Repeat



Experimental Setup

- Implemented SG-SLP in LLVM 3.6

Experimental Setup

- Implemented SG-SLP in LLVM 3.6
- Target: Intel Core i5-6600K

Experimental Setup

- Implemented SG-SLP in LLVM 3.6
- Target: Intel Core i5-6600K
- Compiler flags: -O3 -allow-partial-unroll -march=skylake -mtune=skylake -mavx2

Experimental Setup

- Implemented SG-SLP in LLVM 3.6
- Target: Intel Core i5-6600K
- Compiler flags: -O3 -allow-partial-unroll -march=skylake -mtune=skylake -mavx2
- Kernels from SPEC CPU2006
- We evaluated the following cases:

Experimental Setup

- Implemented SG-SLP in LLVM 3.6
- Target: Intel Core i5-6600K
- Compiler flags: -O3 -allow-partial-unroll -march=skylake -mtune=skylake -mavx2
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  1. All loop, SLP and SG-SLP vectorizers disabled (O3)

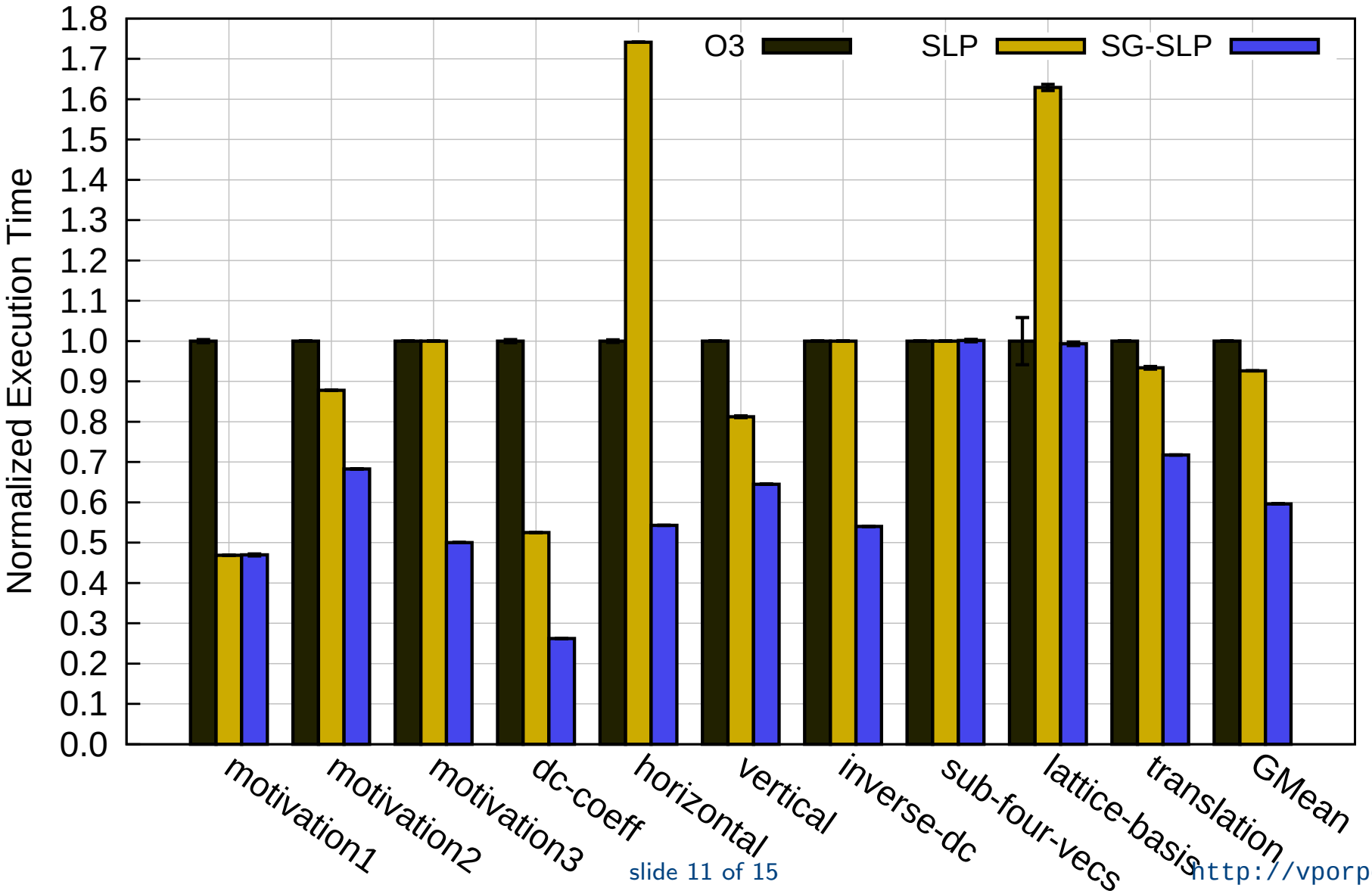
Experimental Setup

- Implemented SG-SLP in LLVM 3.6
- Target: Intel Core i5-6600K
- Compiler flags: -O3 -allow-partial-unroll -march=skylake -mtune=skylake -mavx2
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  1. All loop, SLP and SG-SLP vectorizers disabled (O3)
  2. O3 + SLP enabled (SLP)

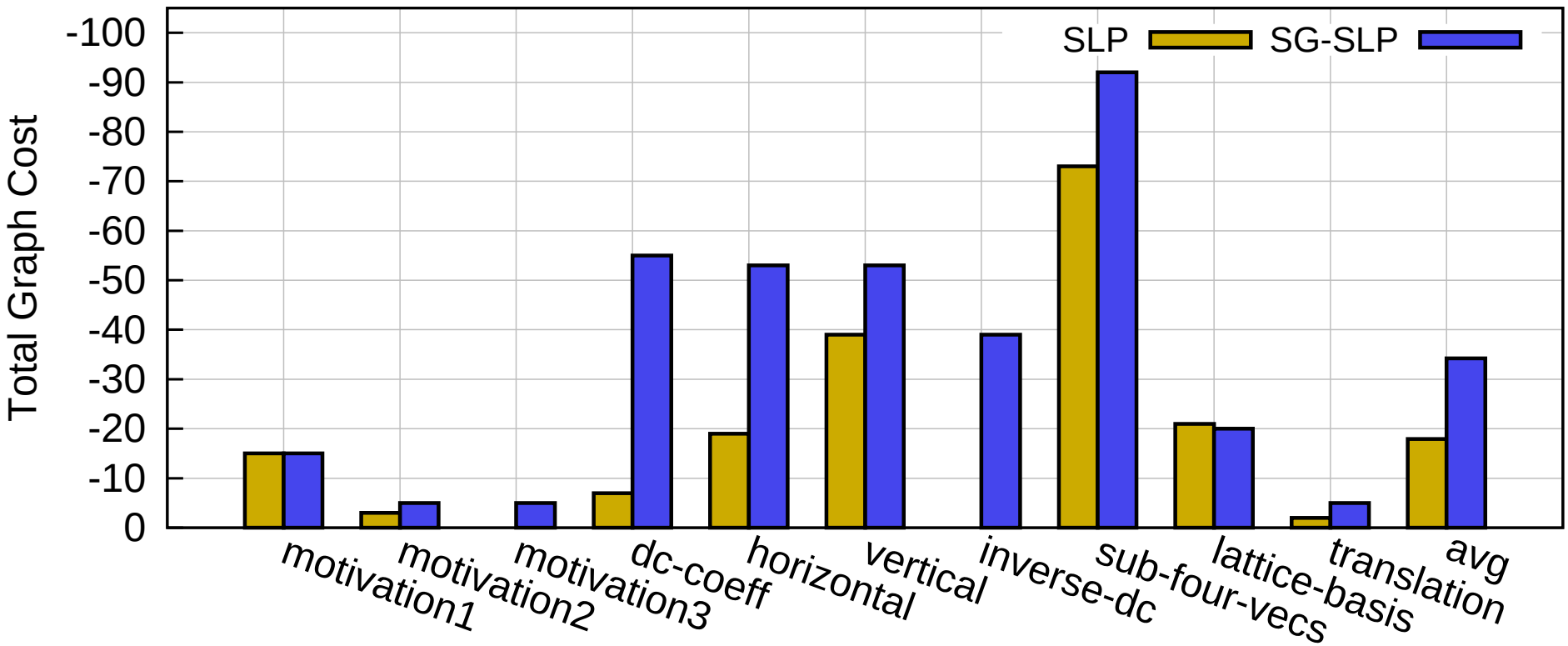
Experimental Setup

- Implemented SG-SLP in LLVM 3.6
- Target: Intel Core i5-6600K
- Compiler flags: -O3 -allow-partial-unroll -march=skylake -mtune=skylake -mavx2
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  1. All loop, SLP and SG-SLP vectorizers disabled (O3)
  2. O3 + SLP enabled (SLP)
  3. O3 + SG-SLP enabled (SG-SLP)

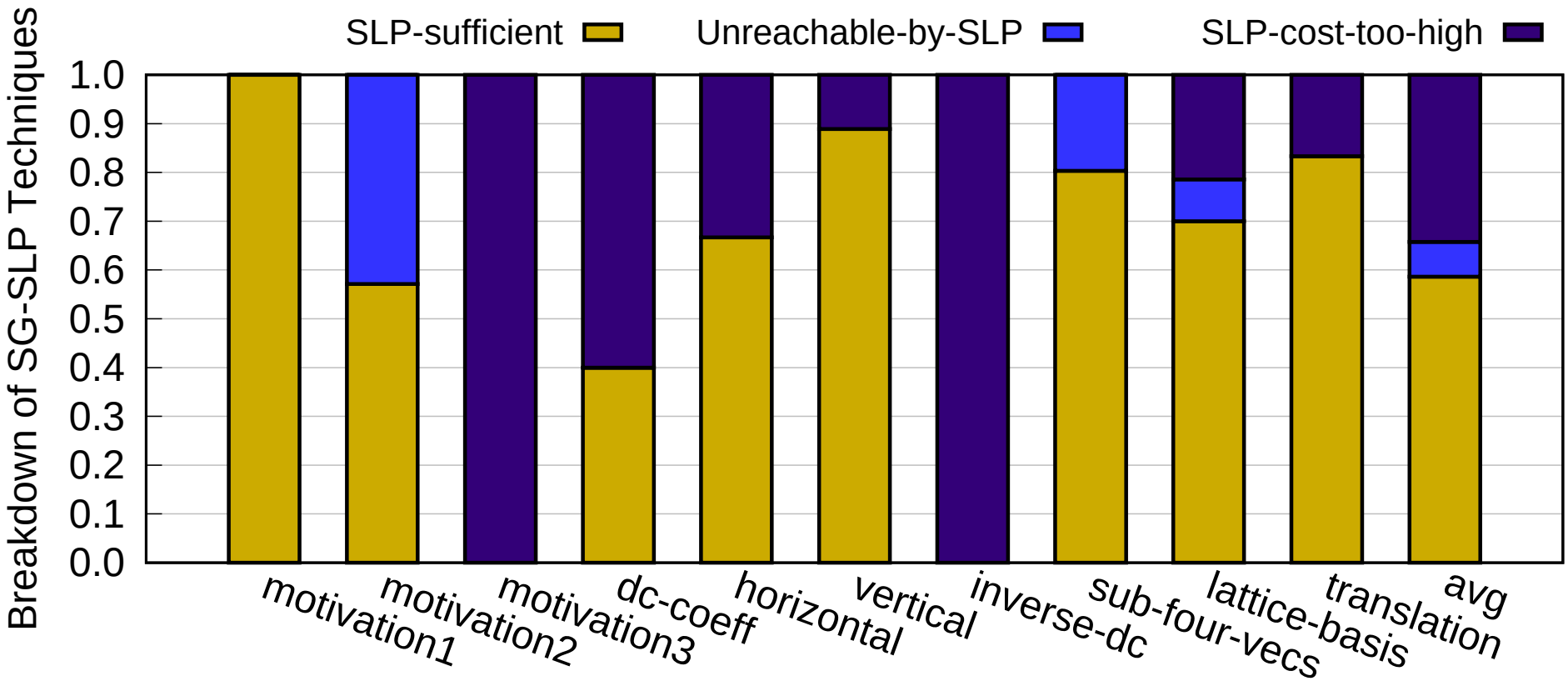
Performance (normalized to O3)



Static Cost (the higher the better)

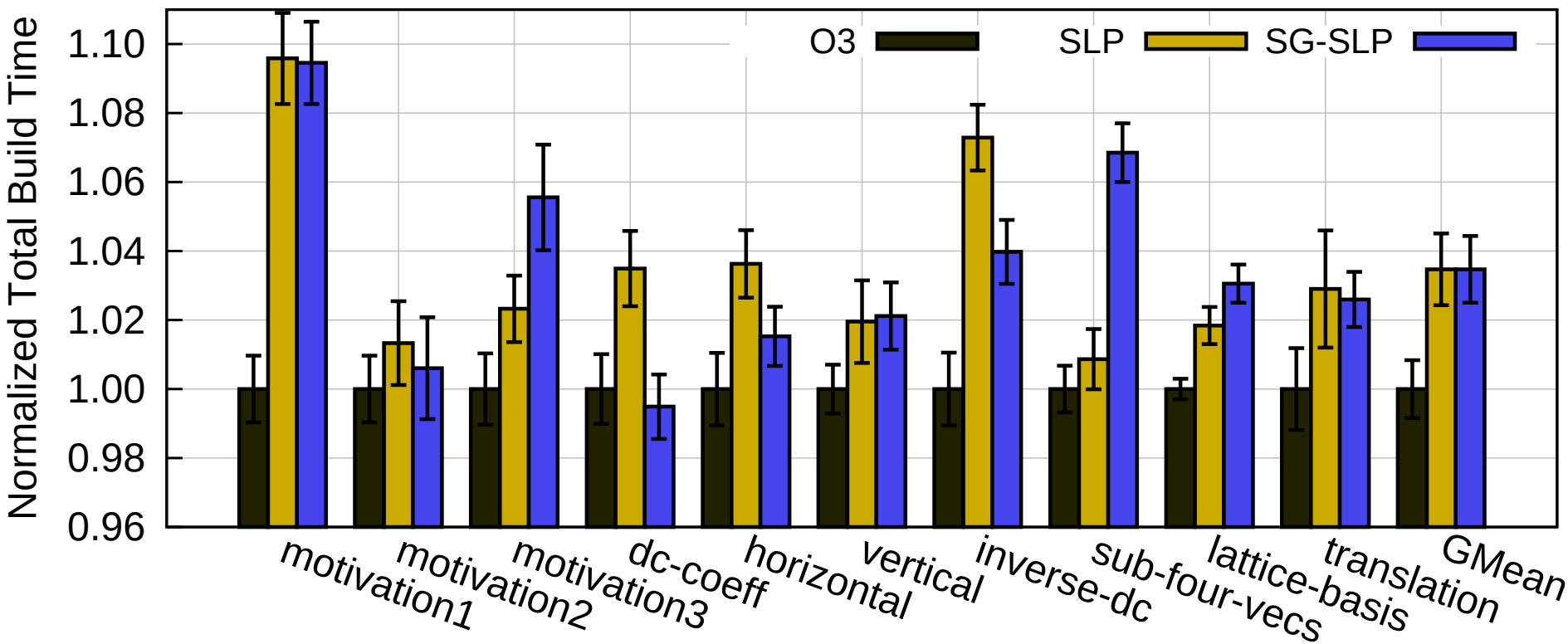


Breakdown of SG-SLP Improvements



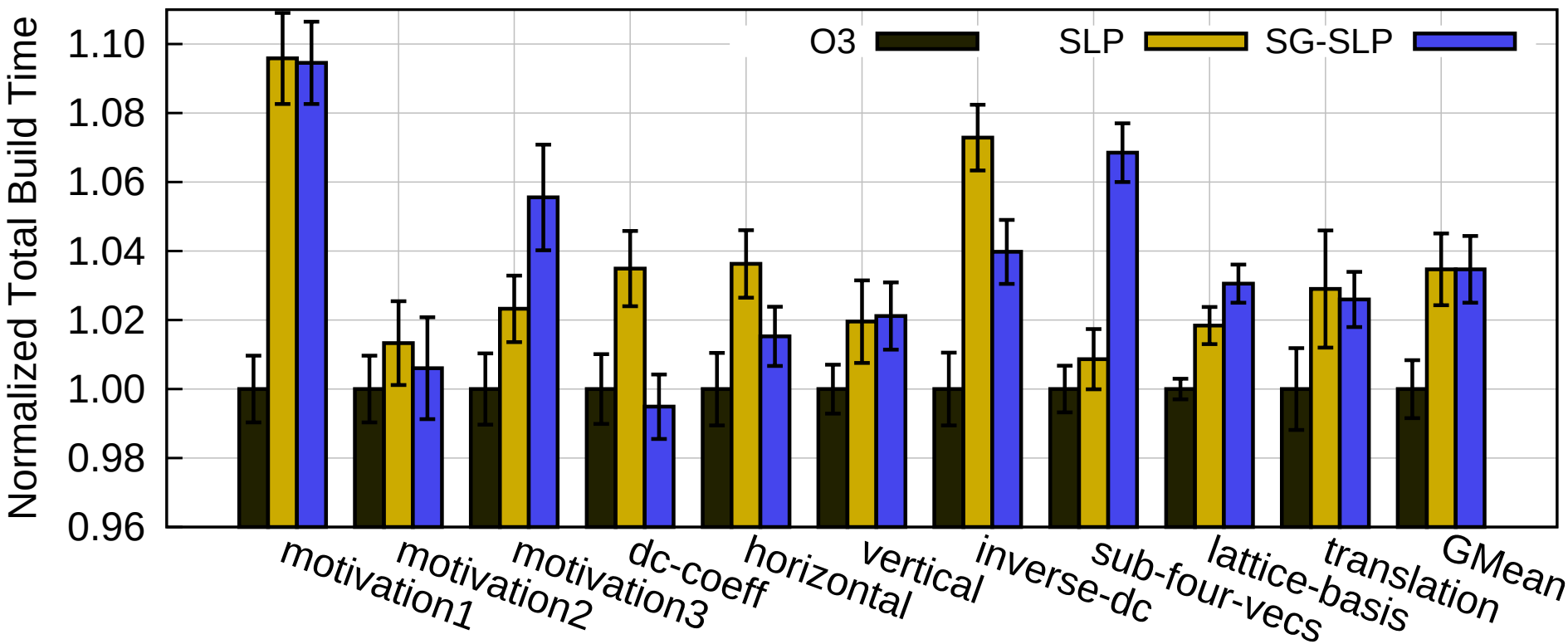
Total Compilation Time

- No significant difference in compilation time



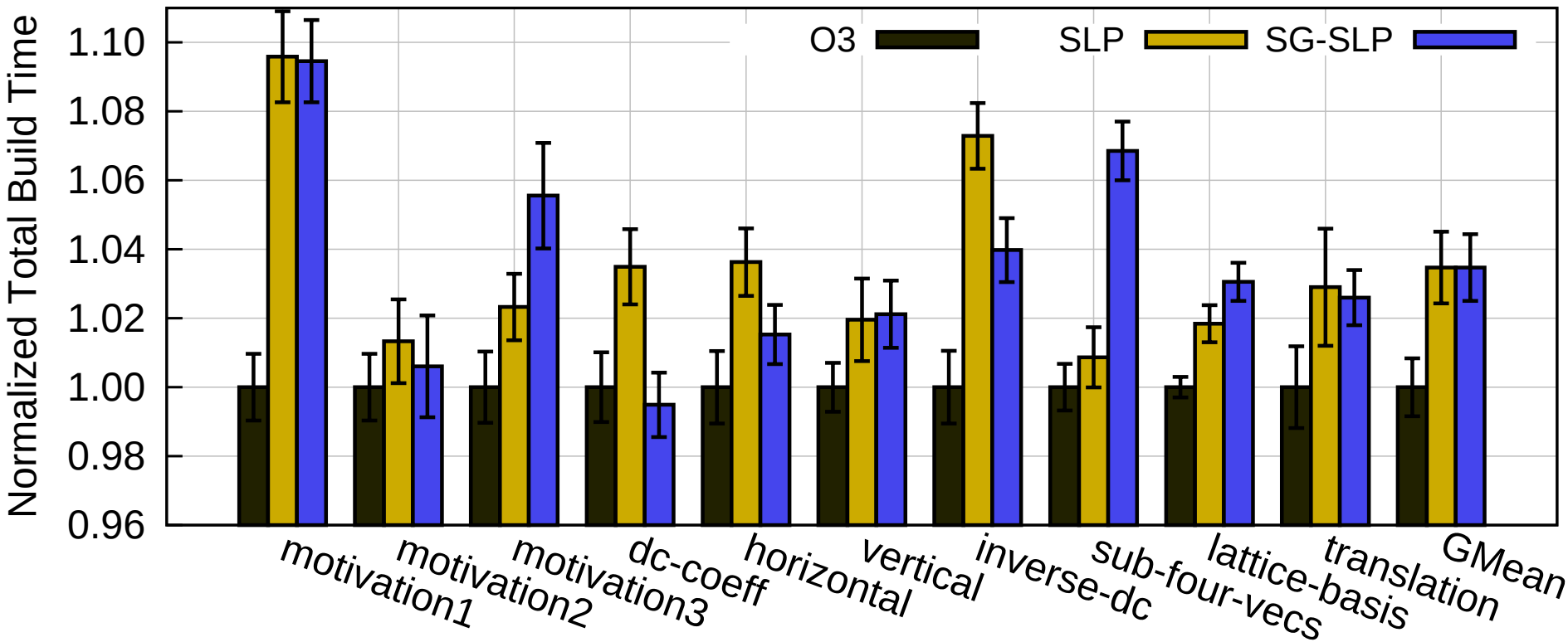
Total Compilation Time

- No significant difference in compilation time
- SG-SLP fails early on non-vectorizable code



Total Compilation Time

- No significant difference in compilation time
- SG-SLP fails early on non-vectorizable code
- Vectorized code usually decreases code size



Conclusion

- SG-SLP forms a larger unified region

Conclusion

- SG-SLP forms a larger unified region
- Overcomes SLP limitations caused by:

Conclusion

- SG-SLP forms a larger unified region
- Overcomes SLP limitations caused by:
  - Cost over estimation due to cross-region dependencies
  - Unreachable Instructions by SLP

Conclusion

- SG-SLP forms a larger unified region
- Overcomes SLP limitations caused by:
  - Cost over estimation due to cross-region dependencies
  - Unreachable Instructions by SLP
- Improves performance and vectorization coverage

Conclusion

- SG-SLP forms a larger unified region
- Overcomes SLP limitations caused by:
  - Cost over estimation due to cross-region dependencies
  - Unreachable Instructions by SLP
- Improves performance and vectorization coverage
- No significant impact on compilation time