Throttling Automatic Vectorization: When Less is More

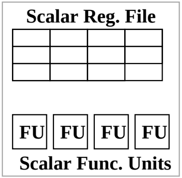
Vasileios Porpodas and Timothy M. Jones

University of Cambridge

PACT 2015

Why SIMD Vectorization?

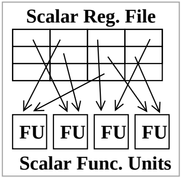
- Scalable parallelism



a. ILP

Why SIMD Vectorization?

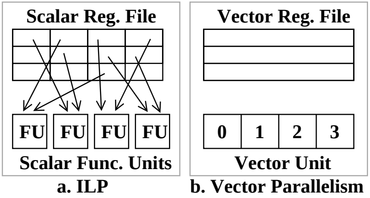
- Scalable parallelism



a. ILP

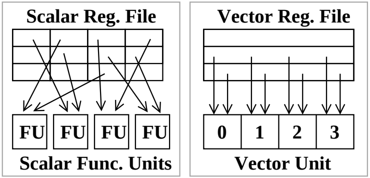
Why SIMD Vectorization?

- Scalable parallelism



Why SIMD Vectorization?

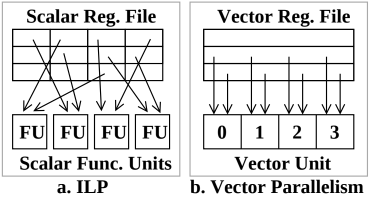
- Scalable parallelism



a. ILP   b. Vector Parallelism

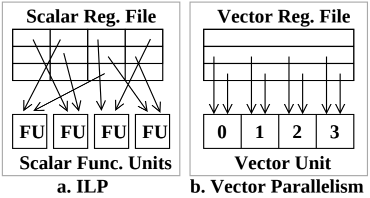
Why SIMD Vectorization?

- Scalable parallelism
- High Performance



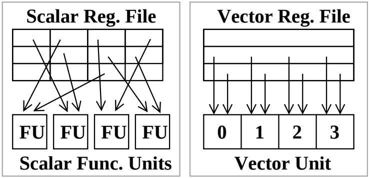
Why SIMD Vectorization?

- Scalable parallelism
- High Performance
- Energy efficiency



Why SIMD Vectorization?

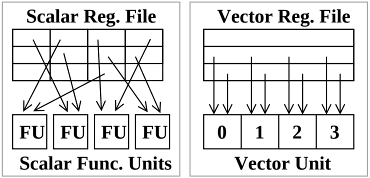
- Scalable parallelism
- High Performance
- Energy efficiency
- Supported since mid 90's
- Frequent updates of vector ISAs



a. ILP   b. Vector Parallelism

Why SIMD Vectorization?

- Scalable parallelism
- High Performance
- Energy efficiency
- Supported since mid 90's
- Frequent updates of vector ISAs
- Vector generation not done in hardware
- Low-level programming or capable compiler



a. ILP   b. Vector Parallelism

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- State-of-the-art straight-line code vectorizer

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- State-of-the-art straight-line code vectorizer
- Implemented in most compilers (including GCC and LLVM)

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- State-of-the-art straight-line code vectorizer
- Implemented in most compilers (including GCC and LLVM)
- In theory it should be a superset of loop-vectorizer

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- State-of-the-art straight-line code vectorizer
- Implemented in most compilers (including GCC and LLVM)
- In theory it should be a superset of loop-vectorizer
  - 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]
- State-of-the-art straight-line code vectorizer
- Implemented in most compilers (including GCC and LLVM)
- In theory it should be a superset of loop-vectorizer
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed
- In practice it is missing features present in the Loop vectorizer (Interleaved Loads, Predication)

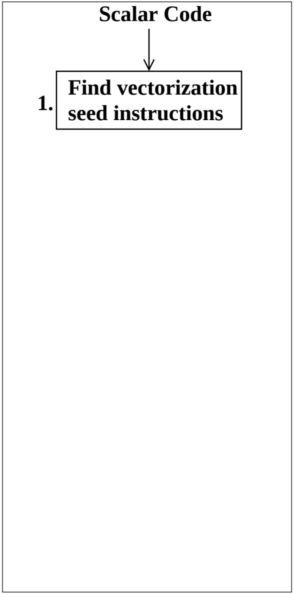
SLP Vectorization Algorithm

- Input is scalar IR

Scalar Code

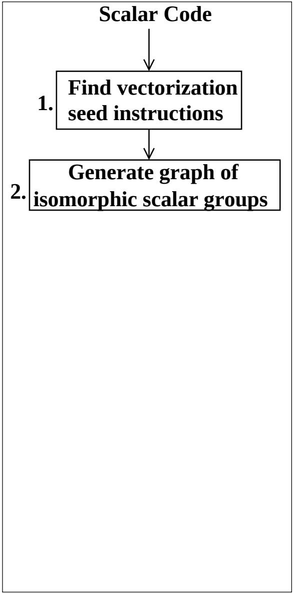
SLP Vectorization Algorithm

- Input is scalar IR
- Seed instructions are:
  1. Consecutive Stores
  2. Reductions



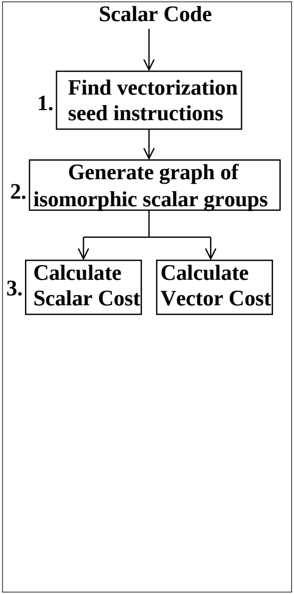
SLP Vectorization Algorithm

- Input is scalar IR
- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains vectorizable isomorphic instructions



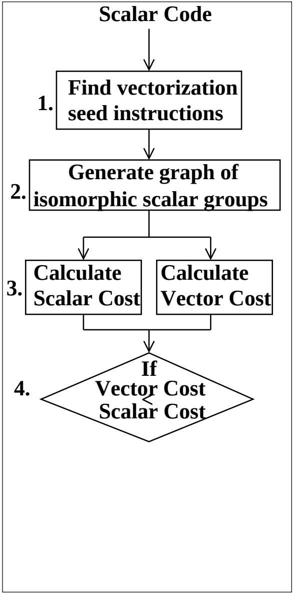
SLP Vectorization Algorithm

- Input is scalar IR
- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains vectorizable isomorphic instructions
- Cost: weighted instr. count



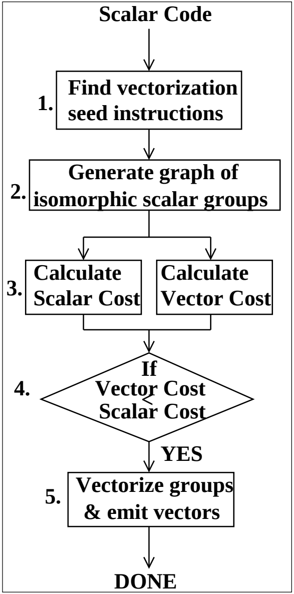
SLP Vectorization Algorithm

- Input is scalar IR
- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains vectorizable isomorphic instructions
- Cost: weighted instr. count
- Check vectorization profitability



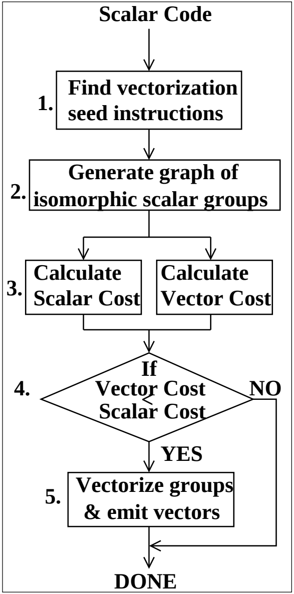
SLP Vectorization Algorithm

- Input is scalar IR
- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains vectorizable isomorphic instructions
- Cost: weighted instr. count
- Check vectorization profitability
- Emit vectors only if profitable



SLP Vectorization Algorithm

- Input is scalar IR
- Seed instructions are:
  1. Consecutive Stores
  2. Reductions
- Graph contains vectorizable isomorphic instructions
- Cost: weighted instr. count
- Check vectorization profitability
- Emit vectors only if profitable



When SLP is not profitable

- Costs outweigh the benefits: E.g. too many gather/scatter instructions

Original	Vectorized
ADD1	
ADD2	
ADD3	
ADD4	

When SLP is not profitable

- Costs outweigh the benefits: E.g. too many gather/scatter instructions

Original	Vectorized
ADD1	
ADD2	
ADD3	
ADD4	
	ADD1 | ADD2 | ADD3 | ADD4

When SLP is not profitable

- Costs outweigh the benefits: E.g. too many gather/scatter instructions

Original	Vectorized
ADD1	Insert1
ADD2	Insert2
ADD3	Insert3
ADD4	Insert4
	ADD1 | ADD2 | ADD3 | ADD4
	Extract1
	Extract2
	Extract3
	Extract4

SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```

SLP not profitable for whole graph

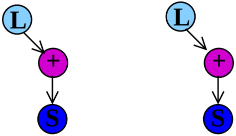
```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```

SLP not profitable for whole graph

```
A[i]   =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```

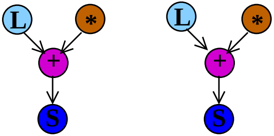
SLP not profitable for whole graph

```
A[i]  =B[i]    + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1]  + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



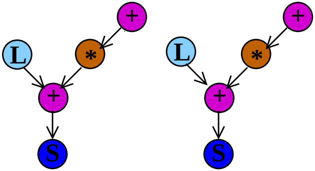
SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



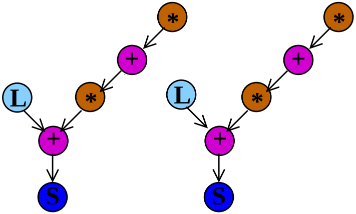
SLP not profitable for whole graph

```
A[i]  =B[i]    + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1]  + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



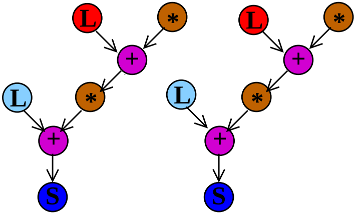
SLP not profitable for whole graph

```
A[i]  =B[i]    + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1]  + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



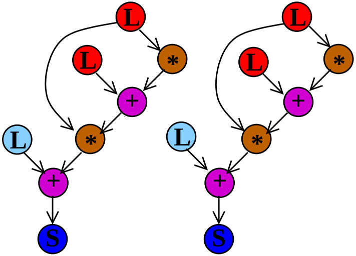
SLP not profitable for whole graph

```
A[i]  =B[i]    + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1]  + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



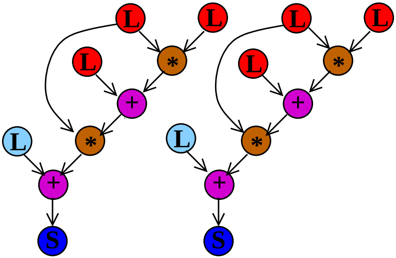
SLP not profitable for whole graph

```
A[i]  =B[i]    + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1]  + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



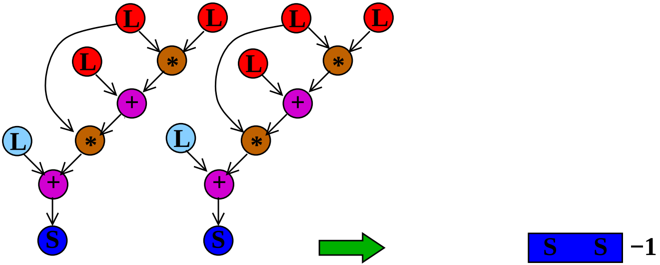
SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



SLP not profitable for whole graph

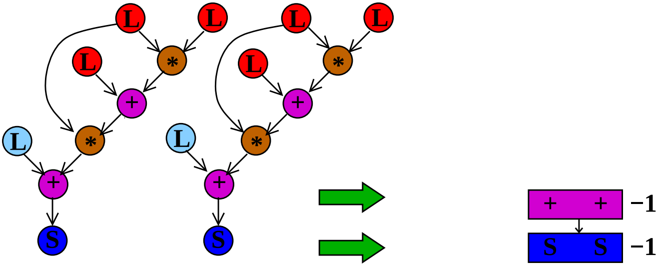
```
A[i]  =B[i]    + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1]  + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



Total Cost: −1

SLP not profitable for whole graph

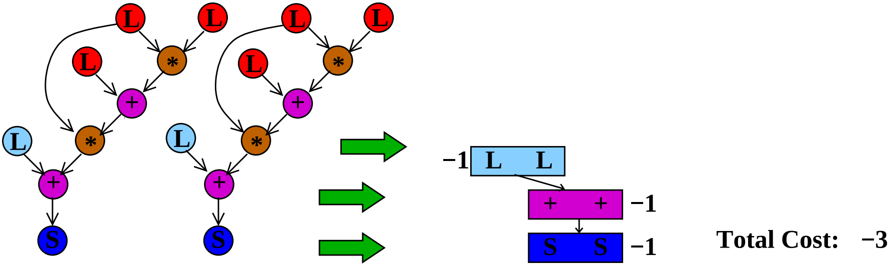
```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



Total Cost: −2

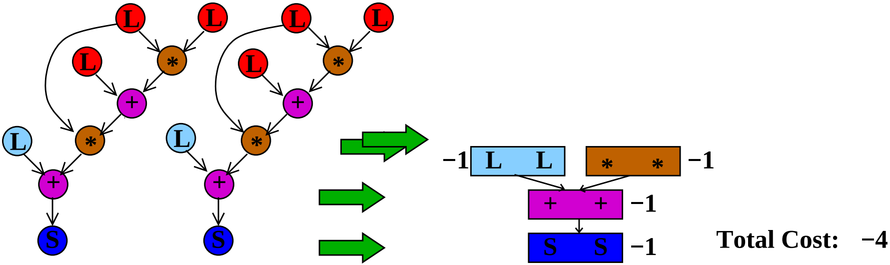
SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



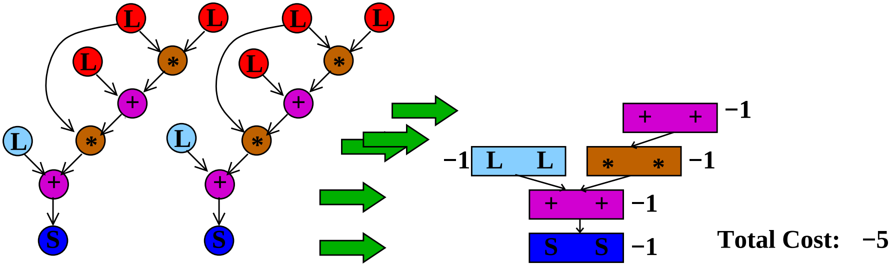
SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



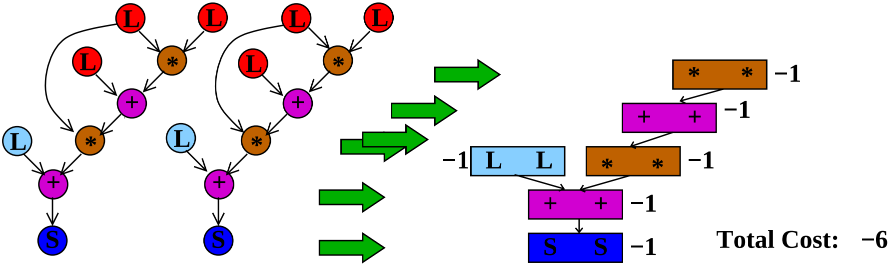
SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



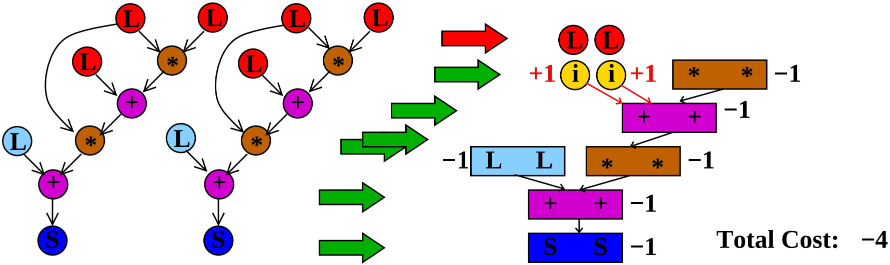
SLP not profitable for whole graph

```
A[i]  =B[i]    + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1]  + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



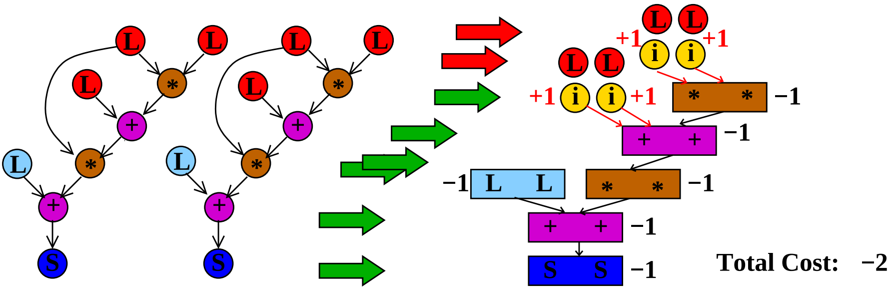
SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



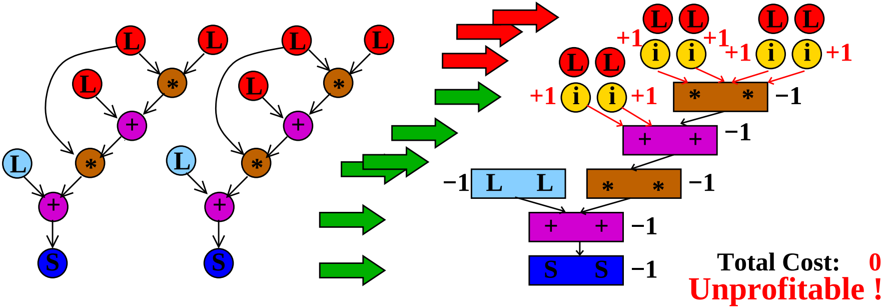
SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



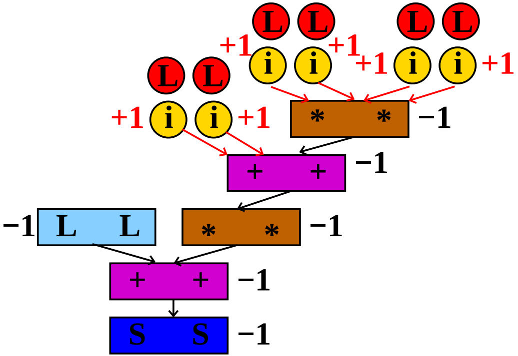
SLP not profitable for whole graph

```
A[i]  =B[i]   + (C[2*i]*(D[2*i]+(E[2*i]*C[2*i])))
A[i+1]=B[i+1] + (C[3*i]*(D[3*i]+(E[3*i]*C[3*i])))
```



TSLP removes unprofitable region

SLP

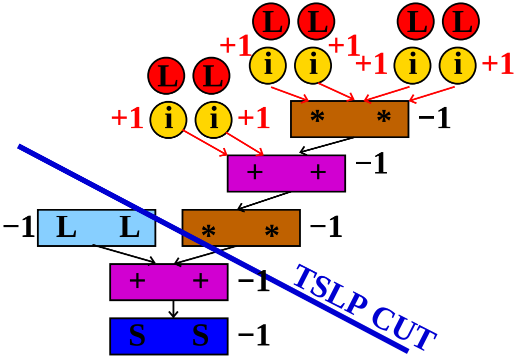


Total Cost: 0

Unprofitable!

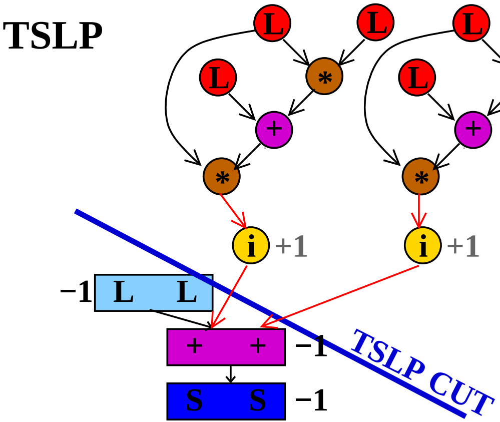
TSLP removes unprofitable region

TSLP



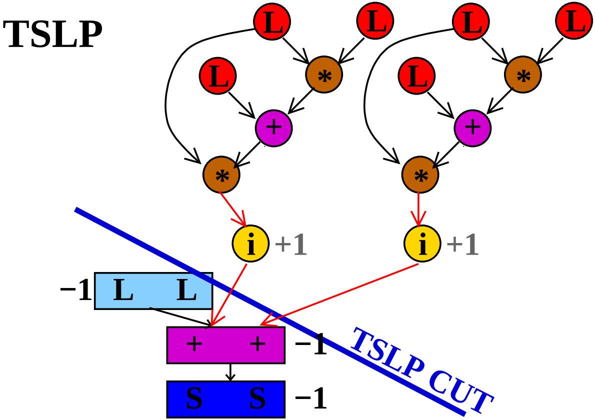
Total Cost:

TSLP removes unprofitable region



Total Cost:

TSLP removes unprofitable region



Total Cost: −1

Profitable !

TSLP Algorithm

- Extension to SLP

Scalar IR

↓

1. **Find seed instructions for vectorization**

TSLP Algorithm

- Extension to SLP

Scalar IR

↓

1. **Find seed instructions for vectorization**

↓

2. **Generate the SLP graph**

TSLP Algorithm

- Extension to SLP
- Try out many cuts

Scalar IR

↓

1. **Find seed instructions for vectorization**

↓

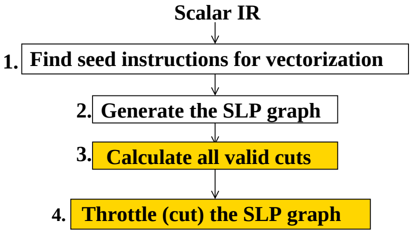
2. **Generate the SLP graph**

↓

3. **Calculate all valid cuts**

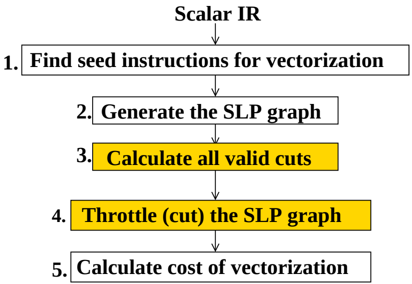
TSLP Algorithm

- Extension to SLP
- Try out many cuts



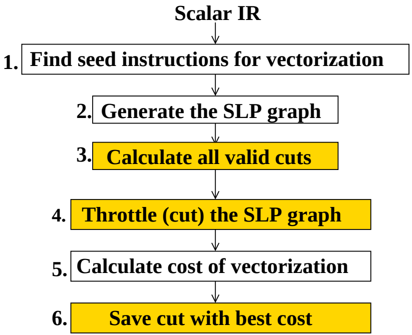
TSLP Algorithm

- Extension to SLP
- Try out many cuts



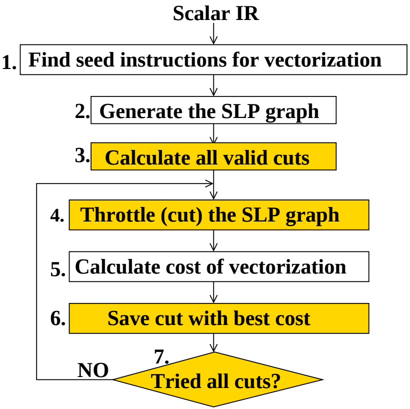
TSLP Algorithm

- Extension to SLP
- Try out many cuts
- Keep best cut



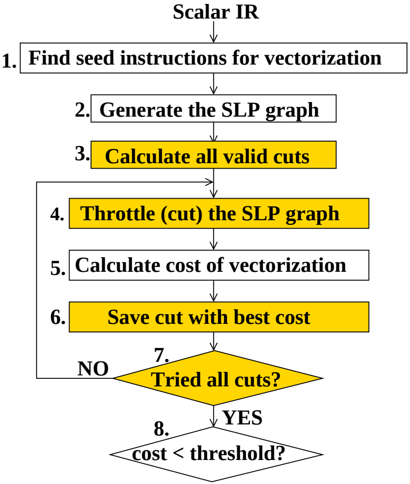
TSLP Algorithm

- Extension to SLP
- Try out many cuts
- Keep best cut



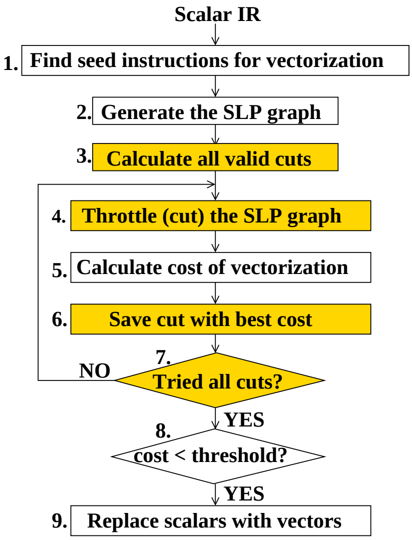
TSLP Algorithm

- Extension to SLP
- Try out many cuts
- Keep best cut
- Vanilla SLP



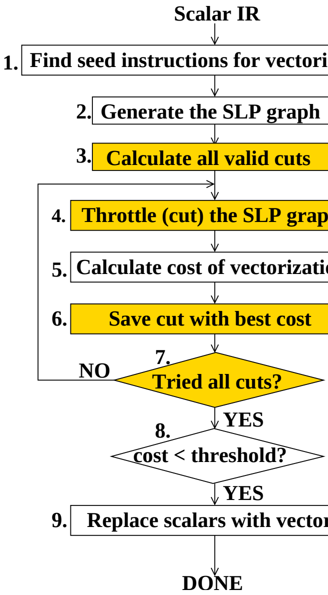
TSLP Algorithm

- Extension to SLP
- Try out many cuts
- Keep best cut
- Vanilla SLP



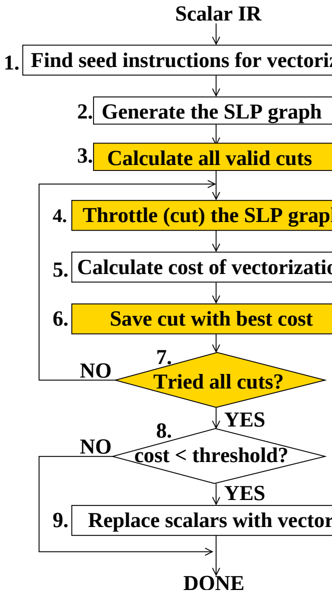
TSLP Algorithm

- Extension to SLP
- Try out many cuts
- Keep best cut
- Vanilla SLP

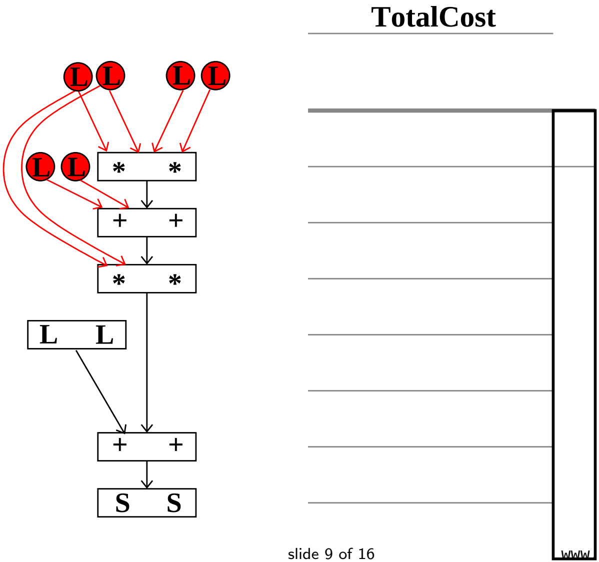


TSLP Algorithm

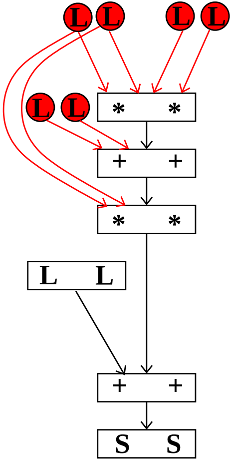
- Extension to SLP
- Try out many cuts
- Keep best cut
- Vanilla SLP

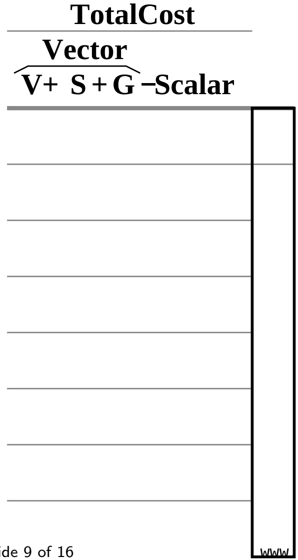


Cost calculation example

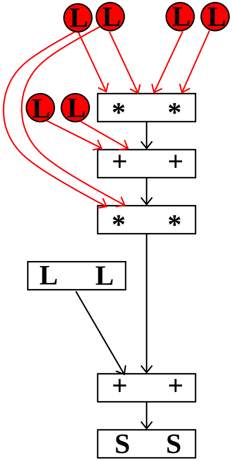


Cost calculation example





Cost calculation example

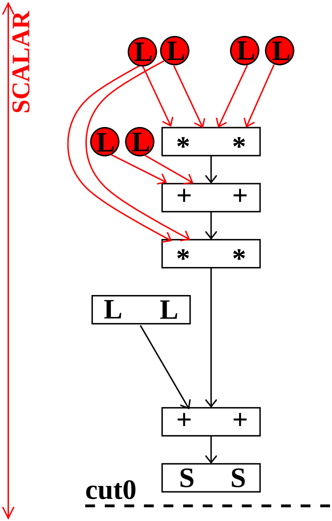


TotalCost

$\overbrace{\mathbf{V+\ S+G}}^{\mathbf{Vector}}\mathbf{-Scalar}$

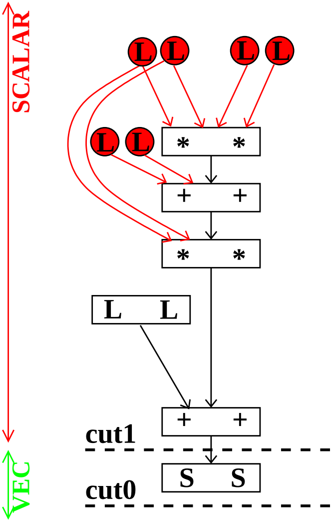
− 18
− 18
− 18
− 18
− 18
− 18
− 18
− 18

Cost calculation example



TotalCost					
Vector: V+	S +	G	−Scalar		
			− 18		
			− 18		
			− 18		
			− 18		
			− 18		
			− 18		
			− 18		
0 +	18+	0	− 18	=	0

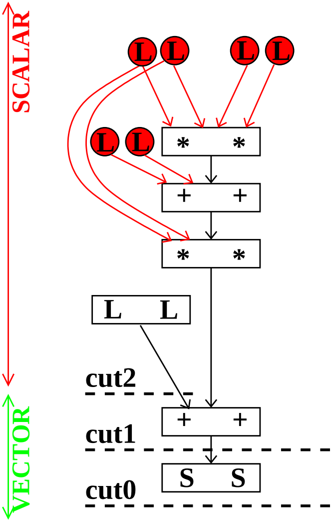
Cost calculation example



TotalCost

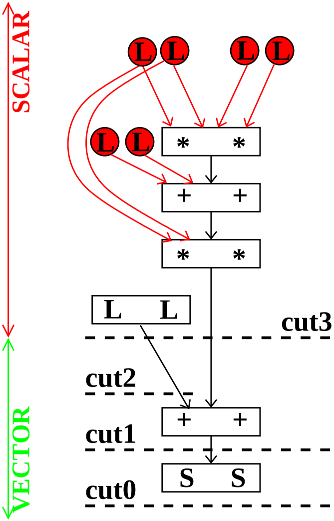
Vector: V +	S +	G	− Scalar		
			− 18		
			− 18		
			− 18		
			− 18		
			− 18		
			− 18		
1 +	16 +	2	− 18	=	+1
0 +	18 +	0	− 18	=	0

Cost calculation example



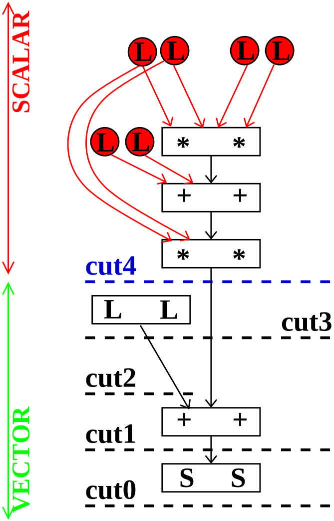
TotalCost		
$\overbrace{V+S+G}^{\text{Vector}}-\text{Scalar}$		
-18		
-18		
-18		
-18		
-18		
$5+8+8-18$	=	+3
$1+16+2-18$	=	+1
$0+18+0-18$	=	0

Cost calculation example



TotalCost						
Vector: V+	S+	G	− Scalar			
			−	18		
			−	18		
			−	18		
			−	18		
2+	14+	4	−	18	=	+2
5+	8+	8	−	18	=	+3
1+	16+	2	−	18	=	+1
0+	18+	0	−	18	=	0

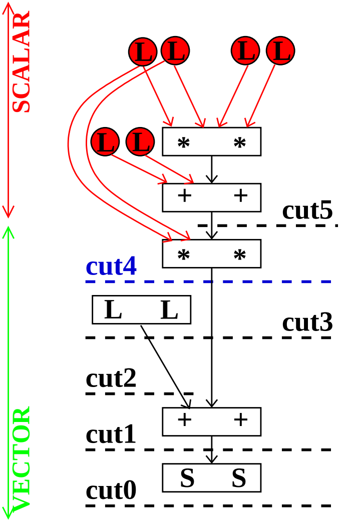
Cost calculation example



TotalCost

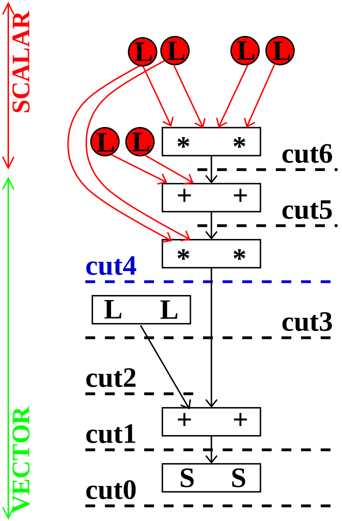
Vector: V +	S +	G	− Scalar		TSLP
			− 18		
			− 18		
			− 18		
3 +	12 +	2	− 18	=	−1
2 +	14 +	4	− 18	=	+2
5 +	8 +	8	− 18	=	+3
1 +	16 +	2	− 18	=	+1
0 +	18 +	0	− 18	=	0

Cost calculation example



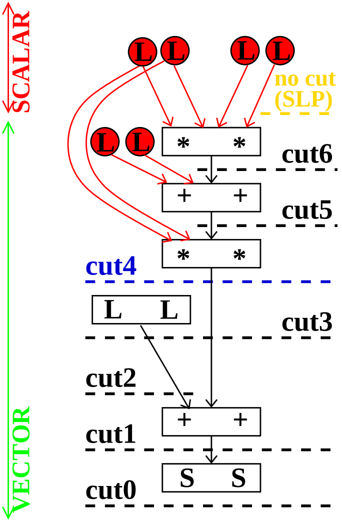
Vector: V + S + G	− Scalar		TotalCost
	− 18		
	− 18		
4 + 10 + 4	− 18	=	0
3 + 12 + 2	− 18	=	−1 (TSLP)
2 + 14 + 4	− 18	=	+2
5 + 8 + 8	− 18	=	+3
1 + 16 + 2	− 18	=	+1
0 + 18 + 0	− 18	=	0

Cost calculation example



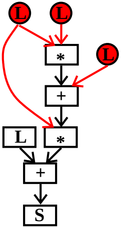
TotalCost: Vector ($\overbrace{V + S + G}$) − Scalar		TSLP
− 18		
5 + 8 + 6 − 18	=	+1
4 + 10 + 4 − 18	=	0
3 + 12 + 2 − 18	=	−1
2 + 14 + 4 − 18	=	+2
5 + 8 + 8 − 18	=	+3
1 + 16 + 2 − 18	=	+1
0 + 18 + 0 − 18	=	0

Cost calculation example



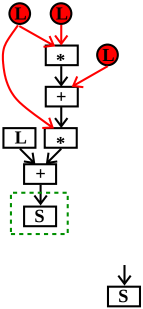
Vector: V + S + G − Scalar	=	TotalCost	
6 + 6 + 6 − 18	=	0	SLP
5 + 8 + 6 − 18	=	+1	
4 + 10 + 4 − 18	=	0	
3 + 12 + 2 − 18	=	−1	TSLP
2 + 14 + 4 − 18	=	+2	
5 + 8 + 8 − 18	=	+3	
1 + 16 + 2 − 18	=	+1	
0 + 18 + 0 − 18	=	0	

Subgraph (Cuts) Generation Algorithm



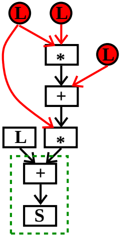
- Only connected subgraphs that include the root

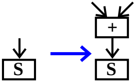
Subgraph (Cuts) Generation Algorithm



- Only connected subgraphs that include the root

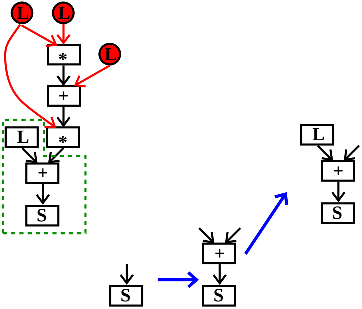
Subgraph (Cuts) Generation Algorithm





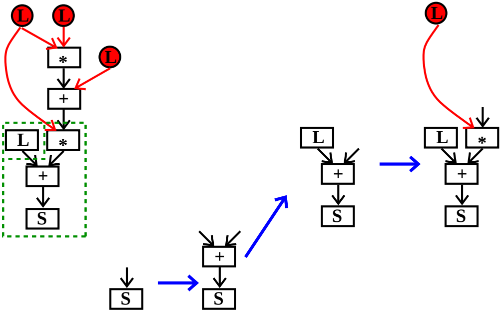
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



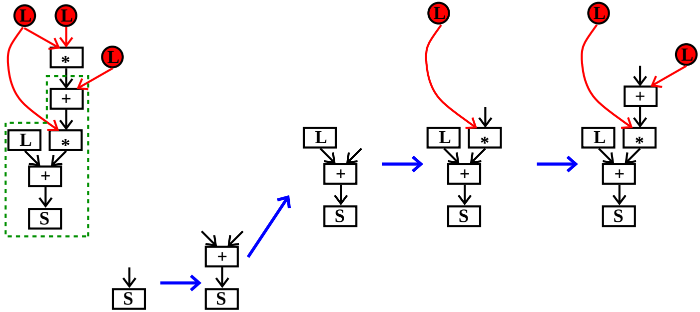
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



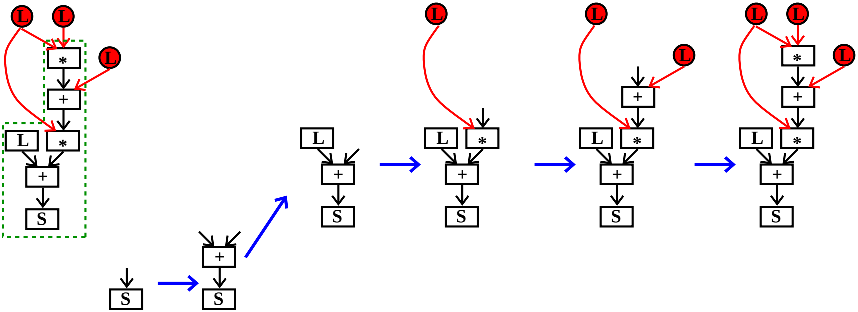
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



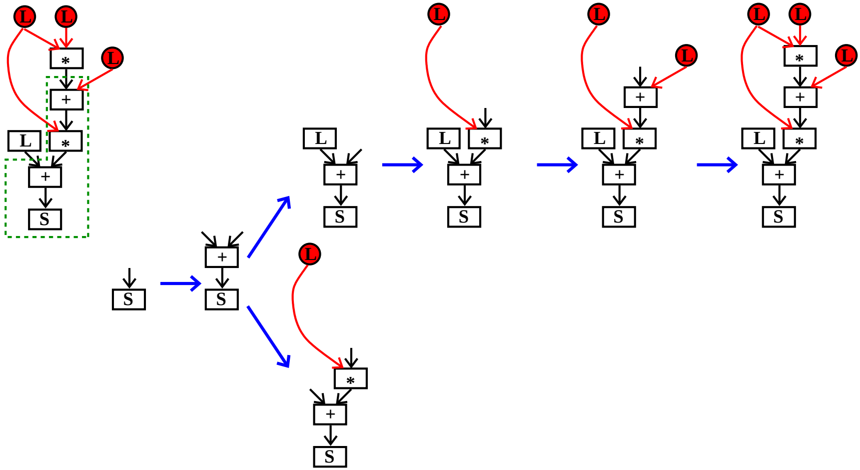
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



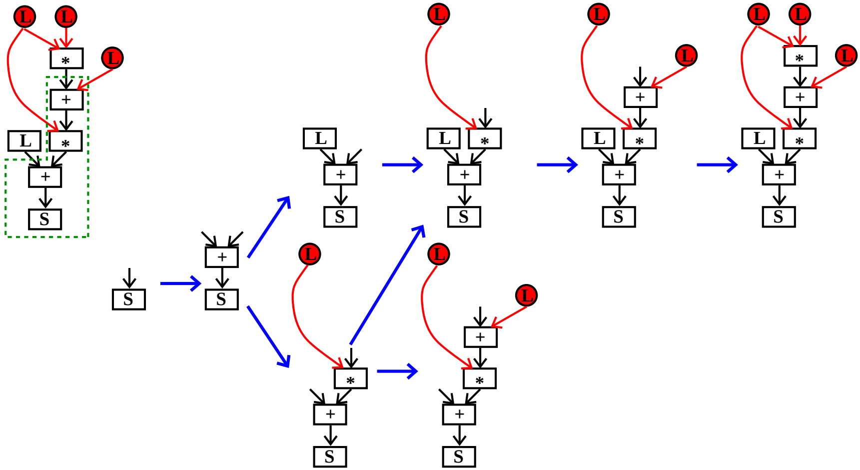
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



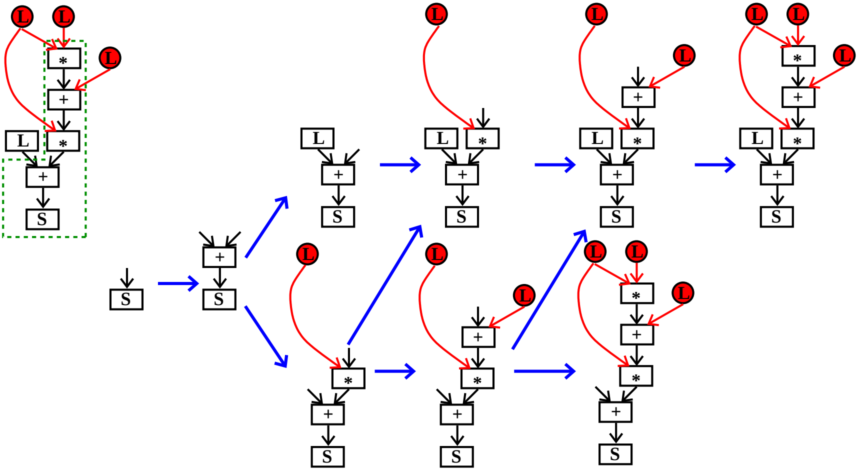
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



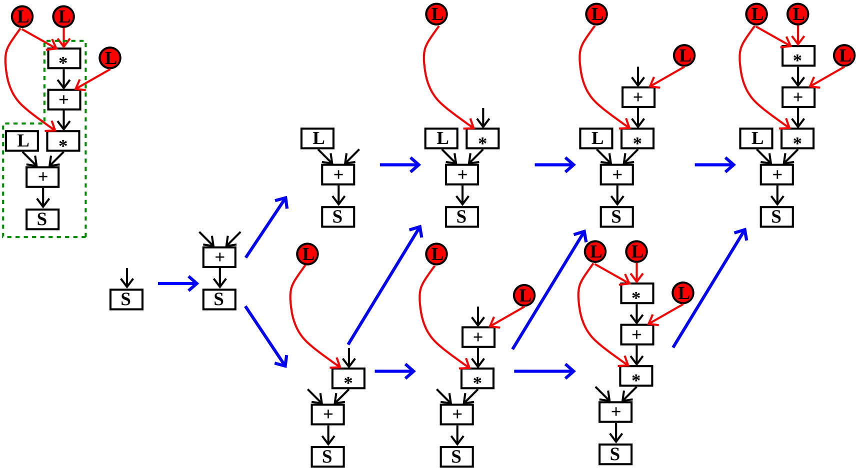
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



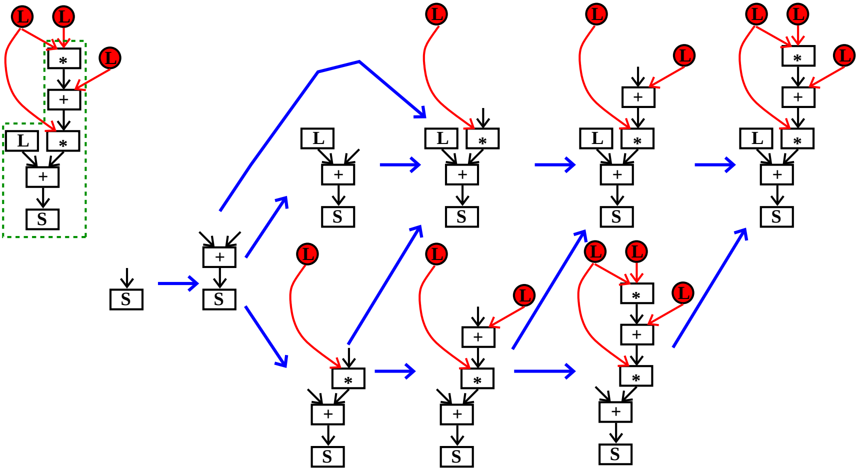
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



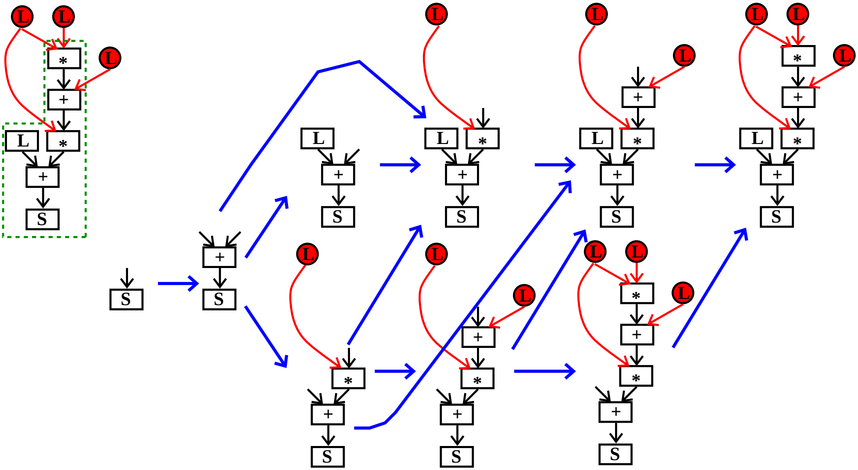
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



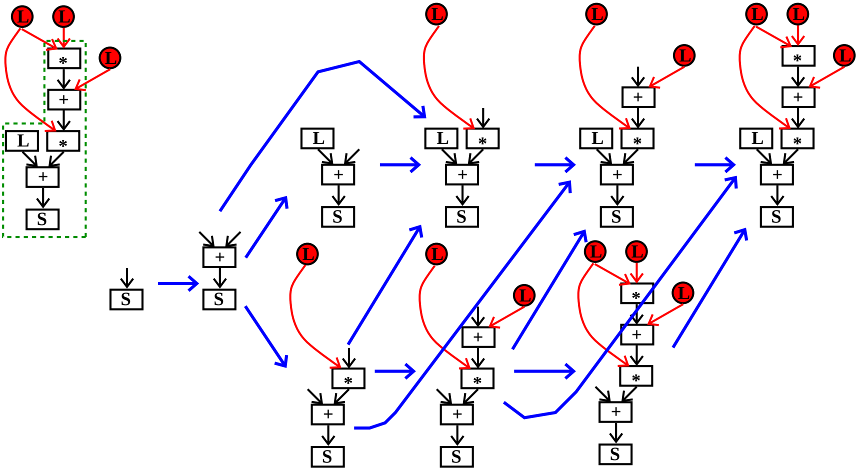
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



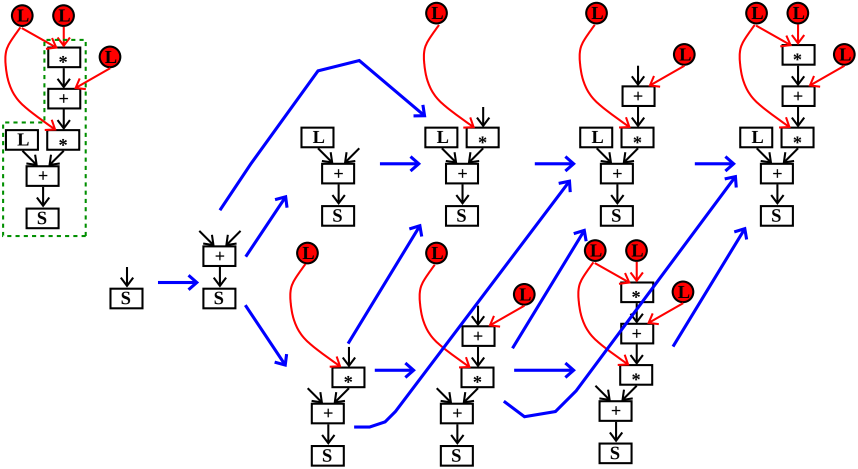
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm



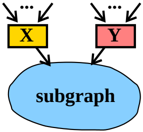
- Only connected subgraphs that include the root

Subgraph (Cuts) Generation Algorithm

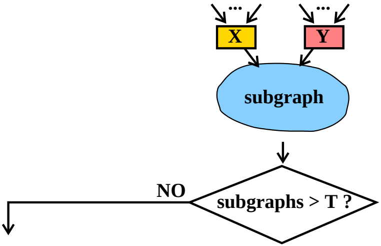


- Only connected subgraphs that include the root
- Worst time complexity $O(2^B x N)$ (N=Nodes, B=Neighbors)

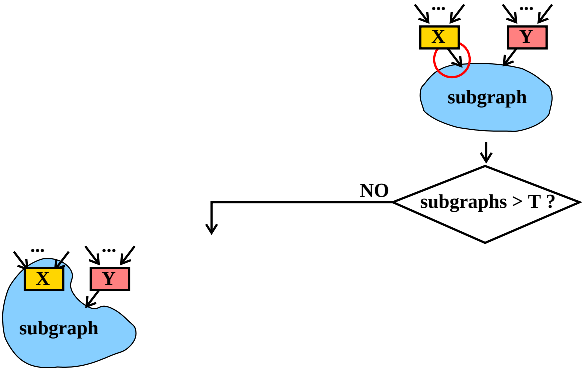
Fast Subgraph (Cuts) Generation Algorithm



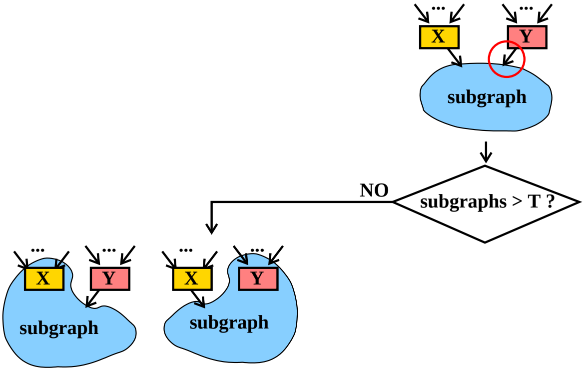
Fast Subgraph (Cuts) Generation Algorithm



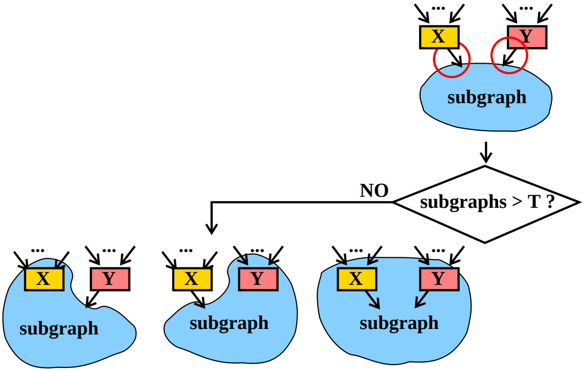
Fast Subgraph (Cuts) Generation Algorithm



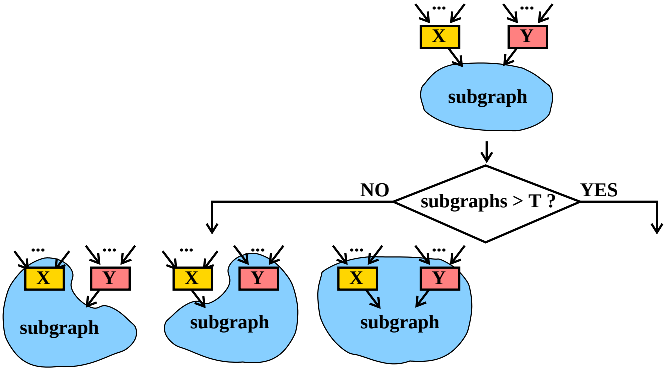
Fast Subgraph (Cuts) Generation Algorithm



Fast Subgraph (Cuts) Generation Algorithm

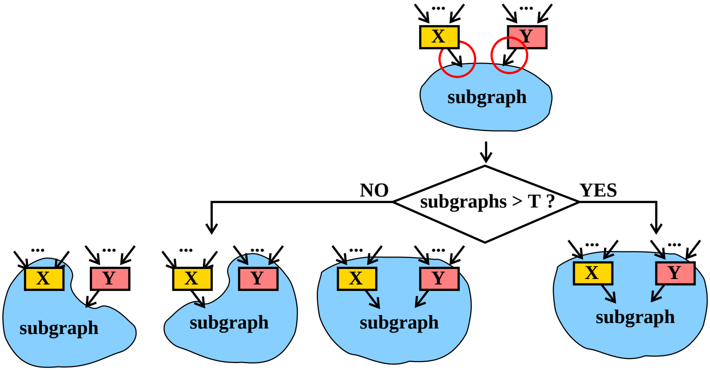


Fast Subgraph (Cuts) Generation Algorithm



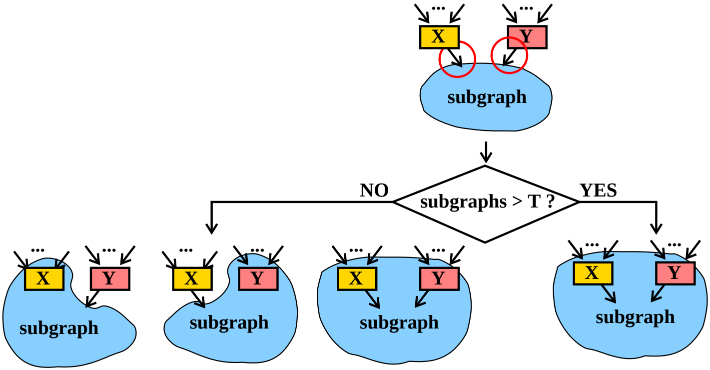
- After T subgraphs, attach all neighbors

Fast Subgraph (Cuts) Generation Algorithm



- After T subgraphs, attach all neighbors

Fast Subgraph (Cuts) Generation Algorithm



- After T subgraphs, attach all neighbors
- Complexity reduced to linear $O(T + N)$

Experimental Setup

- Implemented TSLP in the trunk version of the LLVM 3.6 compiler.

Experimental Setup

- Implemented TSLP in the trunk version of the LLVM 3.6 compiler.
- Target: Intel Core i5-4570 @ 3.2Ghz

Experimental Setup

- Implemented TSLP in the trunk version of the LLVM 3.6 compiler.
- Target: Intel Core i5-4570 @ 3.2Ghz
- Compiler flags: -O3 -allow-partial-unroll -march=core-avx2 -mtune-core-i7

Experimental Setup

- Implemented TSLP in the trunk version of the LLVM 3.6 compiler.
- Target: Intel Core i5-4570 @ 3.2Ghz
- Compiler flags: -O3 -allow-partial-unroll -march=core-avx2 -mtune-core-i7
- Kernels, SPEC 2006 and NPB2.3-C
- We evaluated the following cases:

Experimental Setup

- Implemented TSLP in the trunk version of the LLVM 3.6 compiler.
- Target: Intel Core i5-4570 @ 3.2Ghz
- Compiler flags: -O3 -allow-partial-unroll -march=core-avx2 -mtune-core-i7
- Kernels, SPEC 2006 and NPB2.3-C
- We evaluated the following cases:
  1. All loop, SLP and TSLP vectorizers disabled (O3)

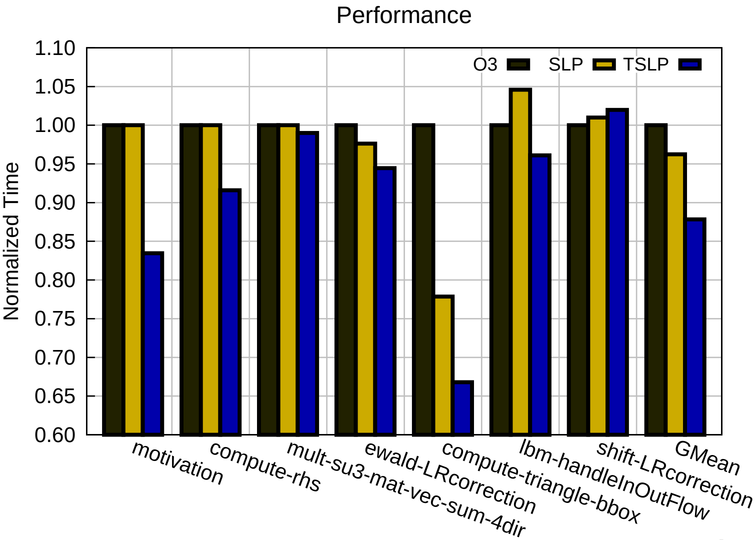
Experimental Setup

- Implemented TSLP in the trunk version of the LLVM 3.6 compiler.
- Target: Intel Core i5-4570 @ 3.2Ghz
- Compiler flags: -O3 -allow-partial-unroll -march=core-avx2 -mtune-core-i7
- Kernels, SPEC 2006 and NPB2.3-C
- We evaluated the following cases:
  1. All loop, SLP and TSLP vectorizers disabled (O3)
  2. O3 + SLP enabled (SLP)

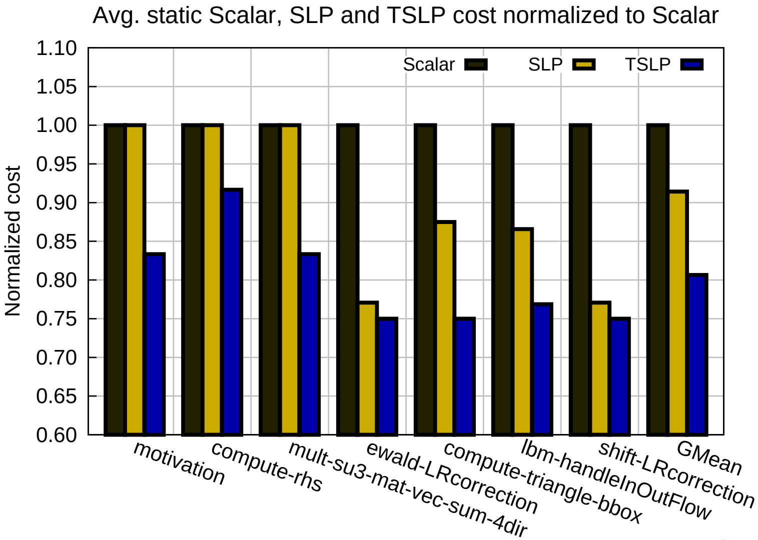
Experimental Setup

- Implemented TSLP in the trunk version of the LLVM 3.6 compiler.
- Target: Intel Core i5-4570 @ 3.2Ghz
- Compiler flags: -O3 -allow-partial-unroll -march=core-avx2 -mtune-core-i7
- Kernels, SPEC 2006 and NPB2.3-C
- We evaluated the following cases:
  1. All loop, SLP and TSLP vectorizers disabled (O3)
  2. O3 + SLP enabled (SLP)
  3. O3 + TSLP enabled (TSLP)

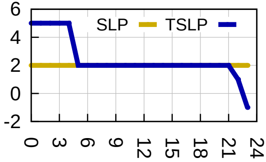
TSLP increases performance



TSLP static cost savings



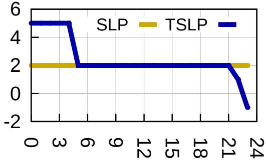
TSLP TotalCost exploration

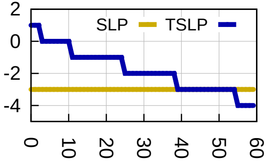


mult-su3-mat-vec-sum

- SLP non-profitable, TSLP profitable

TSLP TotalCost exploration

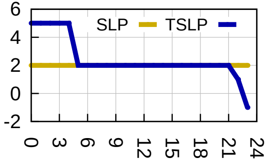




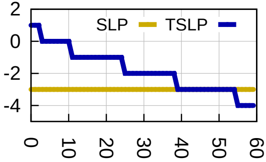
mult-su3-mat-vec-sum   lbm-handleInOutFlow-3

- SLP non-profitable, TSLP profitable
- SLP profitable, but TSLP more profitable

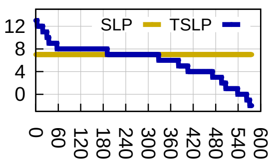
TSLP TotalCost exploration



mult-su3-mat-vec-sum



lbm-handleInOutFlow-3



lab-handleInOutFlow-5

- SLP non-profitable, TSLP profitable
- SLP profitable, but TSLP more profitable
- TSLP exploration gradually improves cost

Conclusion

- TSLP improves vectorization coverage compared to the state-of-the-art

Conclusion

- TSLP improves vectorization coverage compared to the state-of-the-art
- Removes non-profitalbe code regions by:

Conclusion

- TSLP improves vectorization coverage compared to the state-of-the-art
- Removes non-profitalbe code regions by:
  - Evaluating a number of possible cuts

Conclusion

- TSLP improves vectorization coverage compared to the state-of-the-art
- Removes non-profitalbe code regions by:
  - Evaluating a number of possible cuts
  - Estimating their cost

Conclusion

- TSLP improves vectorization coverage compared to the state-of-the-art
- Removes non-profitalbe code regions by:
  - Evaluating a number of possible cuts
  - Estimating their cost
  - Applying the cut with the minimal cost

Conclusion

- TSLP improves vectorization coverage compared to the state-of-the-art
- Removes non-profitalbe code regions by:
  - Evaluating a number of possible cuts
  - Estimating their cost
  - Applying the cut with the minimal cost
- TSLP performs better compared to SLP on commodity SIMD-capable hardware