PostSLP: Cross-Region Vectorization of Fully or Partially Vectorized Code

Vasileios Porpodas and Pushkar Ratnalikar

Intel Corporation, USA
{vasileios.porpodas, pushkar.v.ratnalikar}@intel.com

LCPC 2019

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.

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

Two types of vectorizers

- The traditional Loop Vectorizer (LV)

Two types of vectorizers

- The traditional Loop Vectorizer (LV)
- Straight-line code Vectorizer (SLP)

Two types of vectorizers

- The traditional Loop Vectorizer (LV)
- Straight-line code Vectorizer (SLP)
  - Acronym: Superword Level Parallelism

Two types of vectorizers

- The traditional Loop Vectorizer (LV)
- Straight-line code Vectorizer (SLP)
  - Acronym: Superword Level Parallelism
- Same goal, different means:

Two types of vectorizers

- The traditional Loop Vectorizer (LV)
- Straight-line code Vectorizer (SLP)
  - Acronym: Superword Level Parallelism
- Same goal, different means:
  - LV: parallelism exposed by loops

Two types of vectorizers

- The traditional Loop Vectorizer (LV)
- Straight-line code Vectorizer (SLP)
  - Acronym: Superword Level Parallelism
- Same goal, different means:
  - LV: parallelism exposed by loops
  - SLP: parallelism in straight-line code (e.g., basic-blocks)

SLP compared to Loop Vectorization

- SLP vectorizes across instructions, *NOT* iterations

SLP compared to Loop Vectorization

- SLP 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

- SLP 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]
```

LV

Loop Vectorization (LV) with VF = 4

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

SLP compared to Loop Vectorization

- SLP vectorizes across instructions, *NOT* iterations

Loop Vectorization (LV) with VF = 4

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

LV

```
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

- SLP 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]
```

LV

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]
```

SLP compared to Loop Vectorization

- SLP 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]
```

LV

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]
```

SLP compared to Loop Vectorization

- SLP 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]
```

LV

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]
```

SLP compared to Loop Vectorization

- SLP 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]
```

LV

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 compared to Loop Vectorization

- SLP 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]
```

LV

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

SLP Vectorizer with VF = 4

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

SLP compared to Loop Vectorization

- SLP 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]
```

LV

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

SLP Vectorizer with VF = 4

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

SLP compared to Loop Vectorization

- SLP 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]
```

LV

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

SLP Vectorizer with VF = 4

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

Introduction to SLP

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

Introduction to SLP

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

Introduction to SLP

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

Introduction to SLP

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

Introduction to SLP

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

Introduction to SLP

- Superword Level Parallelism [Larsen et al. PLDI'00]
- Graph isomorphism problem
- 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
- Algorithms can also do SLP-aware LV

Why do we need PostSLP ?

- Missed opportunities by both SLP and LV

Why do we need PostSLP ?

- Missed opportunities by both SLP and LV
  1. SLP regions restricted by seed selection and graph formation

Why do we need PostSLP ?

- Missed opportunities by both SLP and LV
  1. SLP regions restricted by seed selection and graph formation
  2. SLP can partially vectorize code, leaving instructions scalar

Why do we need PostSLP ?

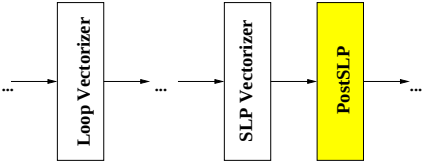
- Missed opportunities by both SLP and LV
  1. SLP regions restricted by seed selection and graph formation
  2. SLP can partially vectorize code, leaving instructions scalar
  3. LV may vectorize the loop such that the largest datatype fits the architecture

Why do we need PostSLP ?

- Missed opportunities by both SLP and LV
  1. SLP regions restricted by seed selection and graph formation
  2. SLP can partially vectorize code, leaving instructions scalar
  3. LV may vectorize the loop such that the largest datatype fits the architecture
- PostSLP: SLP-style pass capable of mixed vectorization of scalars and/or vectors

Why do we need PostSLP ?

- Missed opportunities by both SLP and LV
  1. SLP regions restricted by seed selection and graph formation
  2. SLP can partially vectorize code, leaving instructions scalar
  3. LV may vectorize the loop such that the largest datatype fits the architecture
- PostSLP: SLP-style pass capable of mixed vectorization of scalars and/or vectors
- Runs after both vectorizers



1/3 Missed opportunities by SLP due to regions

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

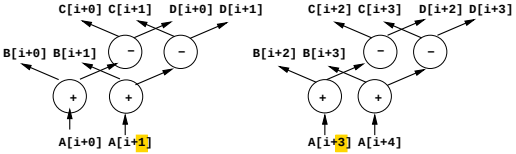
1/3 Missed opportunities by SLP due to regions

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

1/3 Missed opportunities by SLP due to regions

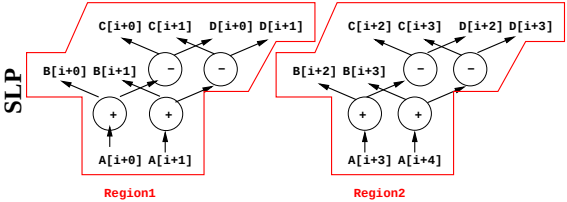
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

SLP



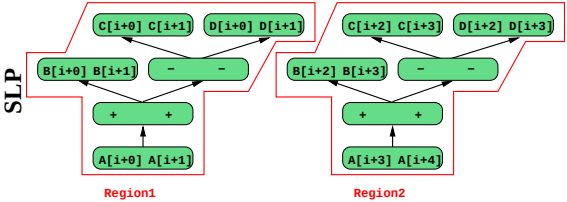
1/3 Missed opportunities by SLP due to regions

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```



1/3 Missed opportunities by SLP due to regions

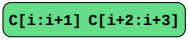
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```



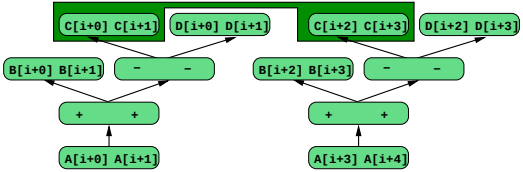
1/3 Missed opportunities by SLP due to regions

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

PostSLP



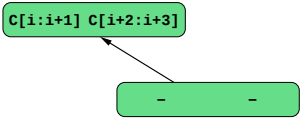
SLP



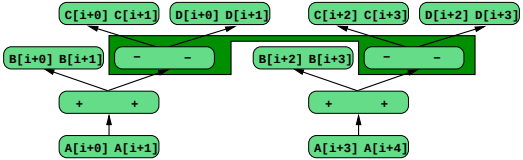
1/3 Missed opportunities by SLP due to regions

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

PostSLP



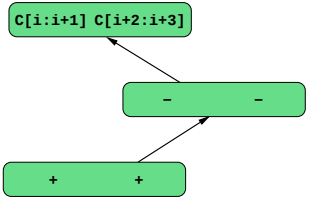
SLP



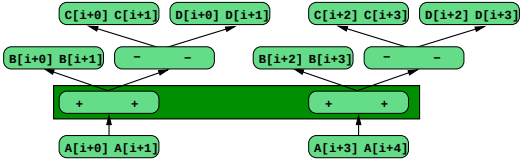
1/3 Missed opportunities by SLP due to regions

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

PostSLP

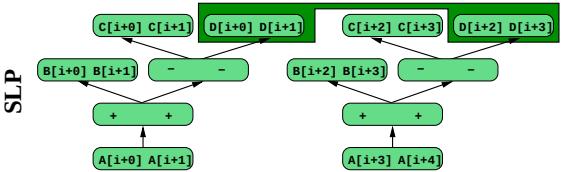


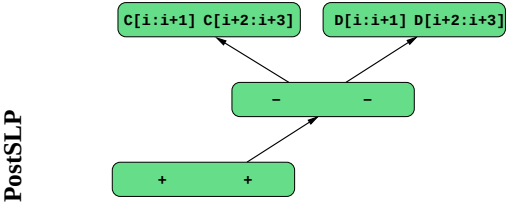
SLP



1/3 Missed opportunities by SLP due to regions

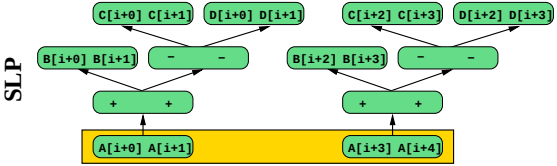
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

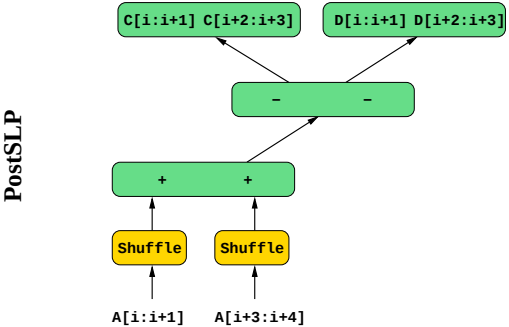




1/3 Missed opportunities by SLP due to regions

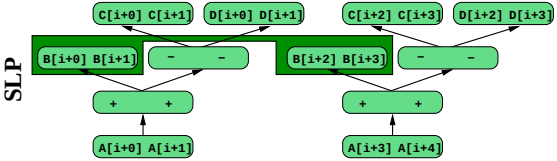
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

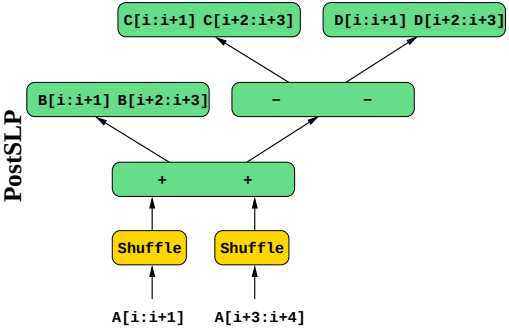




1/3 Missed opportunities by SLP due to regions

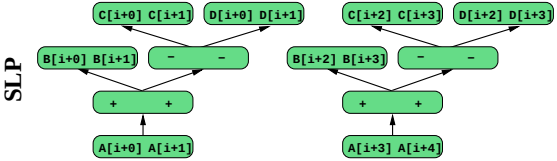
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

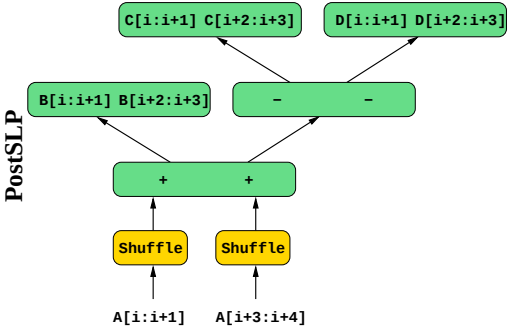




1/3 Missed opportunities by SLP due to regions

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```






```
Tmp = B[i:i+3] + C[i:i+3] - D[i:i+3]
A[i+0:i+1] = shuffle<0:1>(Tmp)
A[i+3:i+4] = shuffle<2:3>(Tmp)
```

2/3 Partially Vectorized Code by SLP

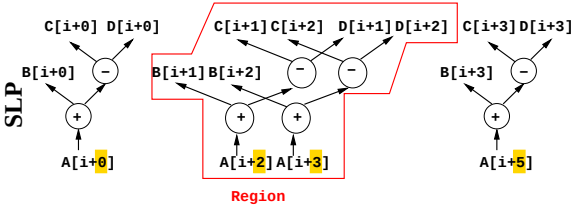
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

2/3 Partially Vectorized Code by SLP

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

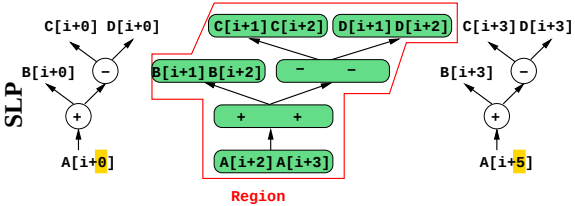
2/3 Partially Vectorized Code by SLP

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```



2/3 Partially Vectorized Code by SLP

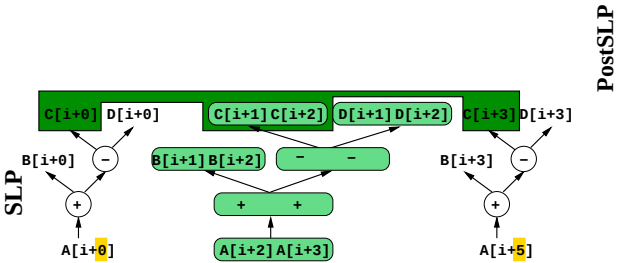
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```



2/3 Partially Vectorized Code by SLP

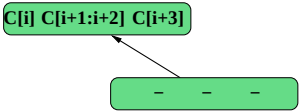
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

C[i] C[i+1:i+2] C[i+3]



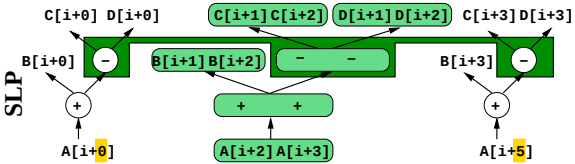
2/3 Partially Vectorized Code by SLP

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```



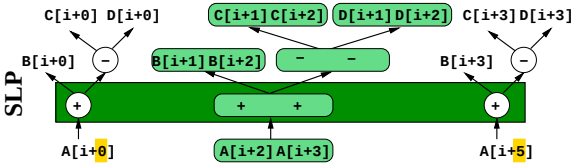
PostSLP

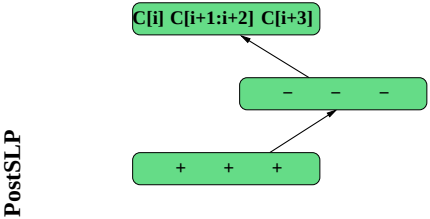
SLP



2/3 Partially Vectorized Code by SLP

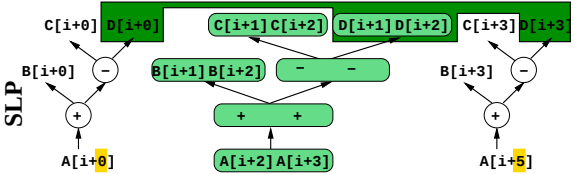
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

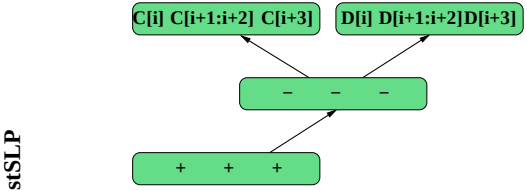




2/3 Partially Vectorized Code by SLP

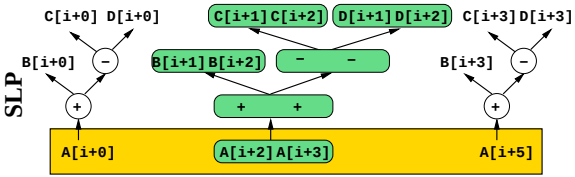
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

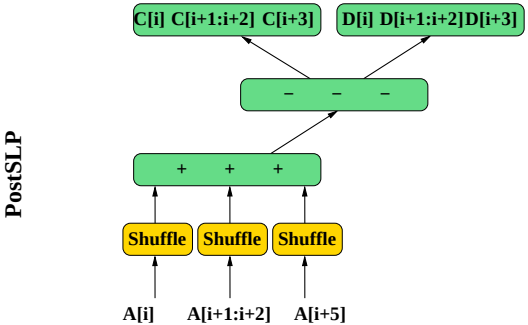




2/3 Partially Vectorized Code by SLP

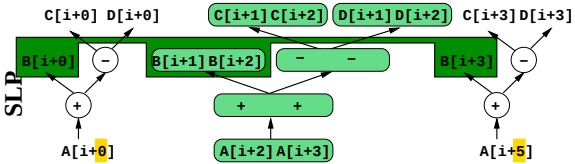
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

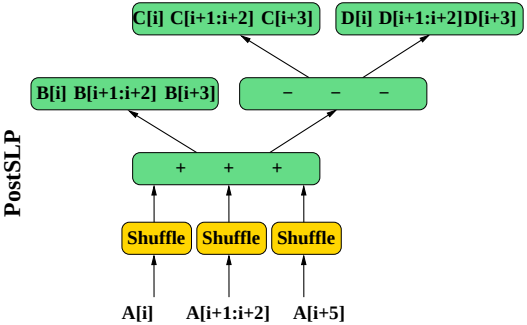




2/3 Partially Vectorized Code by SLP

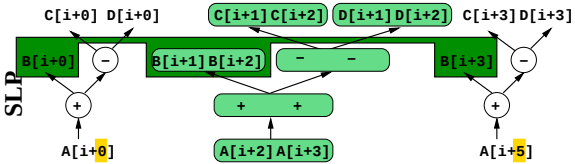
```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

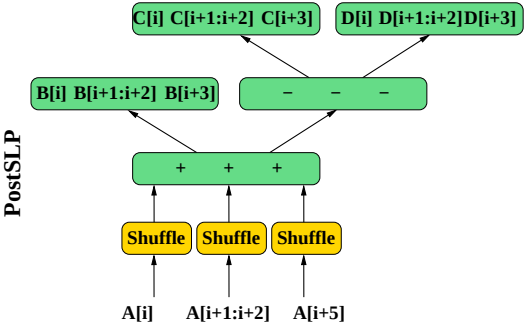




2/3 Partially Vectorized Code by SLP

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```






```
Tmp = B[i:i+3] + C[i:i+3] - D[i:i+3]
A[i+0] = shuffle<0>(Tmp)
A[i+2:i+3] = shuffle<1:2>(Tmp)
A[i+5] = shuffle<3>(Tmp)
```

3/3 LV Vectorizing for the Widest Data Type

```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

3/3 LV Vectorizing for the Widest Data Type

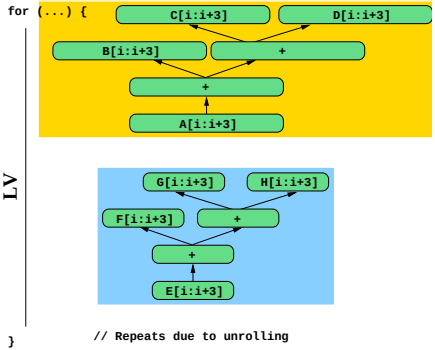
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

3/3 LV Vectorizing for the Widest Data Type

```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

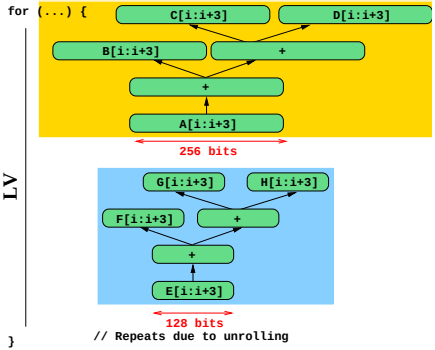
3/3 LV Vectorizing for the Widest Data Type

```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```



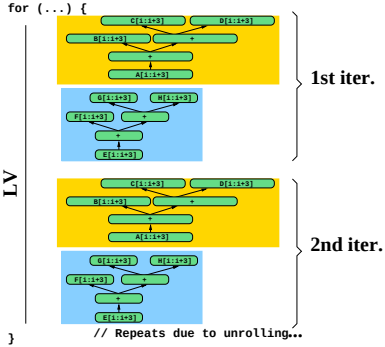
3/3 LV Vectorizing for the Widest Data Type

```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```



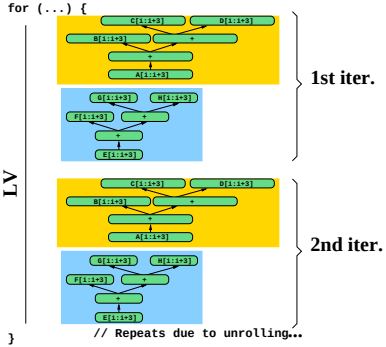
3/3 LV Vectorizing for the Widest Data Type

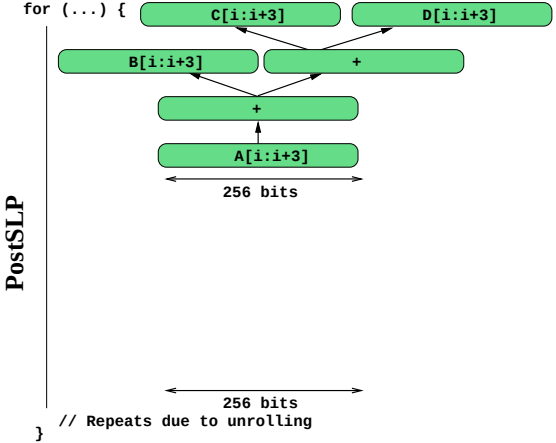
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```



3/3 LV Vectorizing for the Widest Data Type

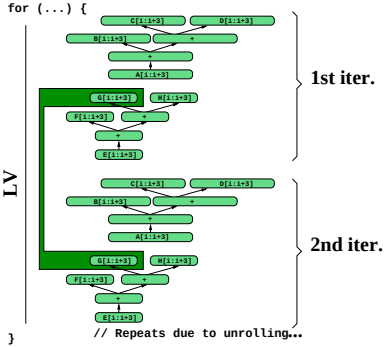
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

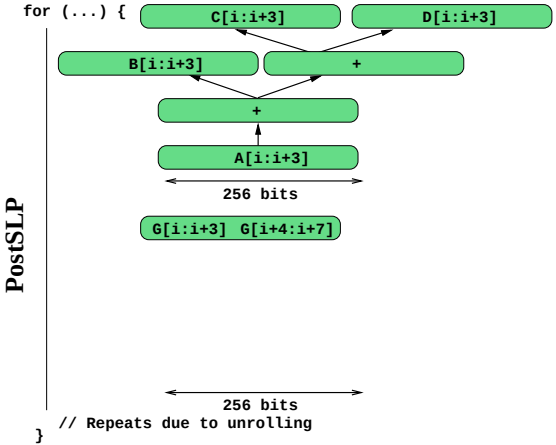




3/3 LV Vectorizing for the Widest Data Type

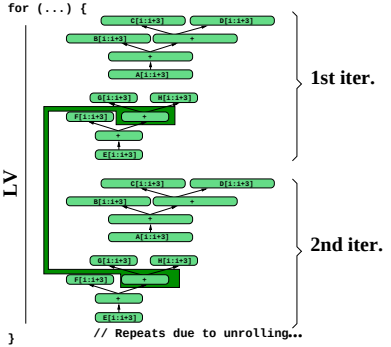
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

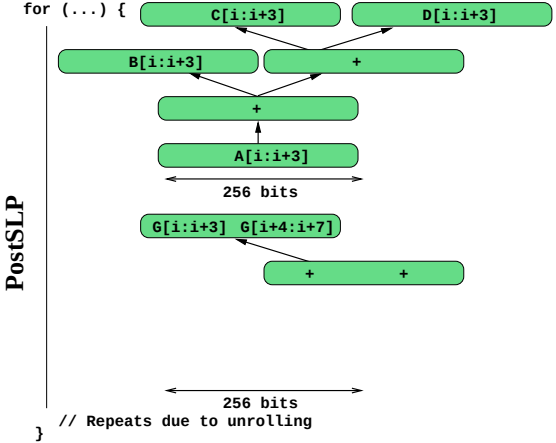




3/3 LV Vectorizing for the Widest Data Type

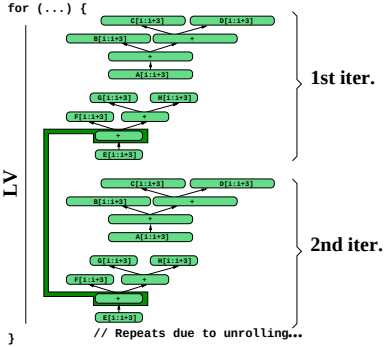
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

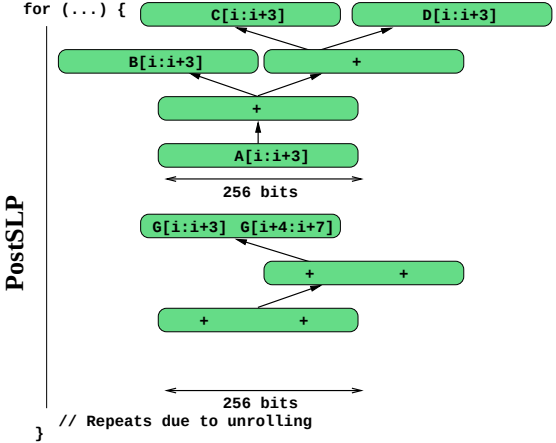




3/3 LV Vectorizing for the Widest Data Type

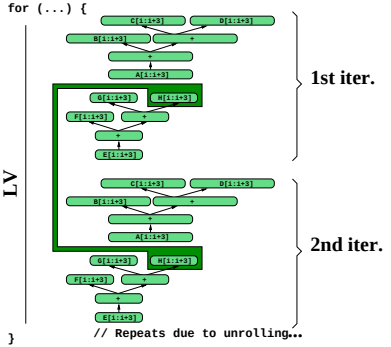
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

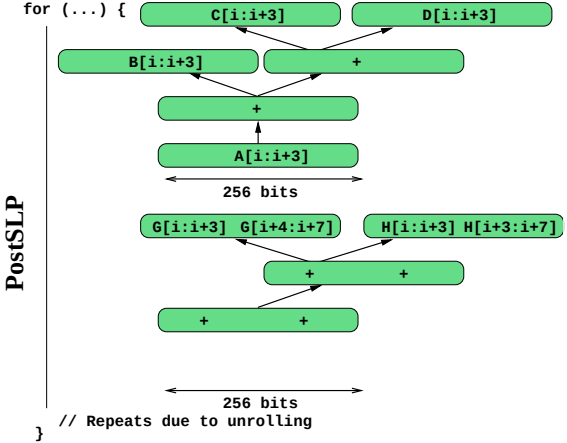




3/3 LV Vectorizing for the Widest Data Type

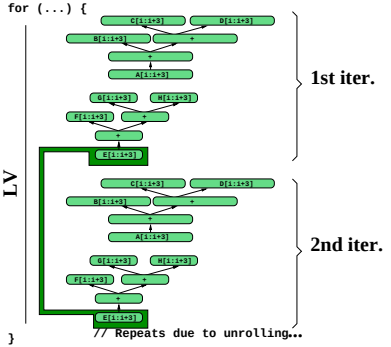
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

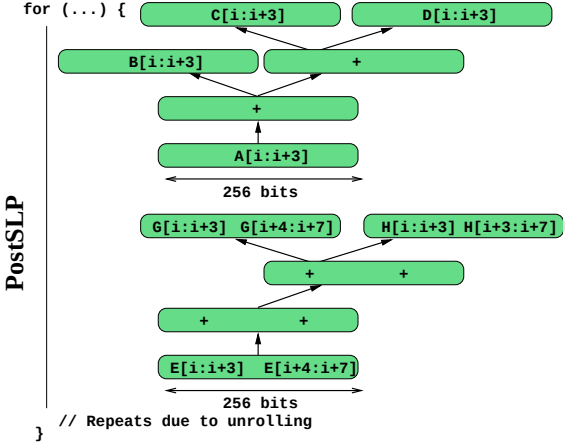




3/3 LV Vectorizing for the Widest Data Type

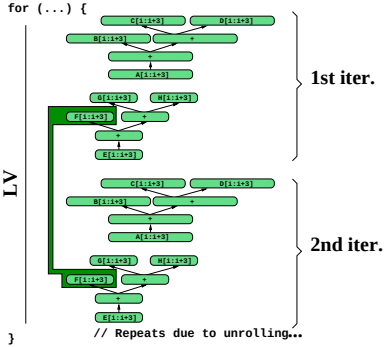
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

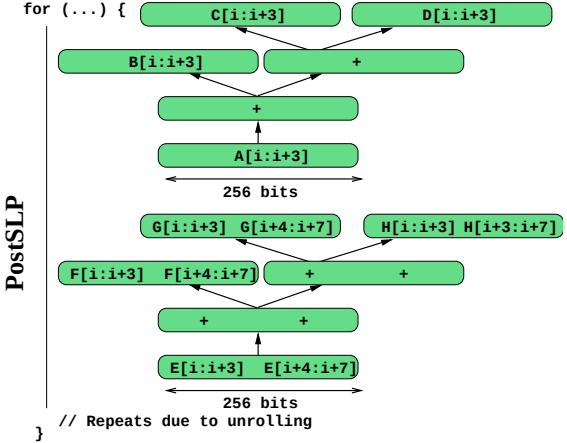




3/3 LV Vectorizing for the Widest Data Type

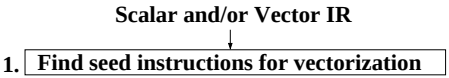
```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```





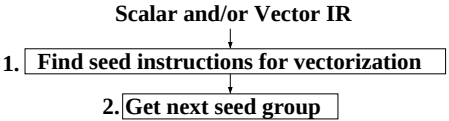
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores



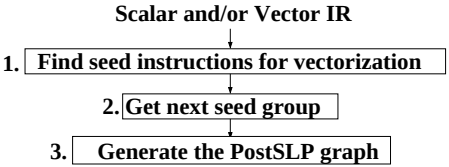
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores



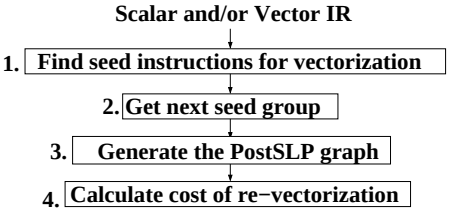
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses



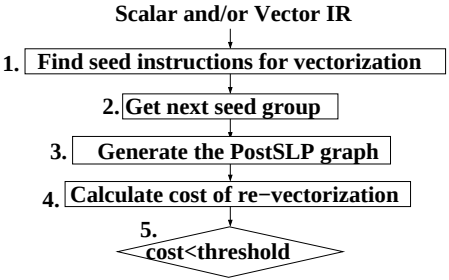
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses
- Cost: weighted instruction count



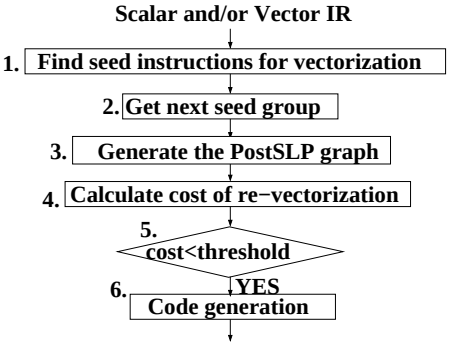
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses
- Cost: weighted instruction count
- Check overall profitability



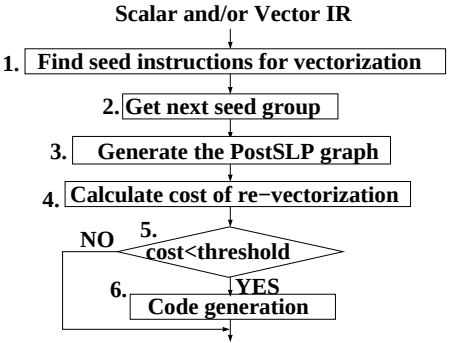
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses
- Cost: weighted instruction count
- Check overall profitability
- Generate code for groups



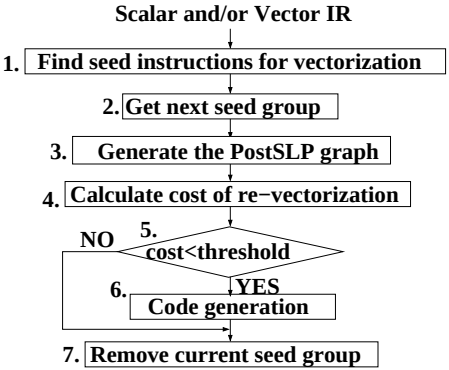
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses
- Cost: weighted instruction count
- Check overall profitability
- Generate code for groups



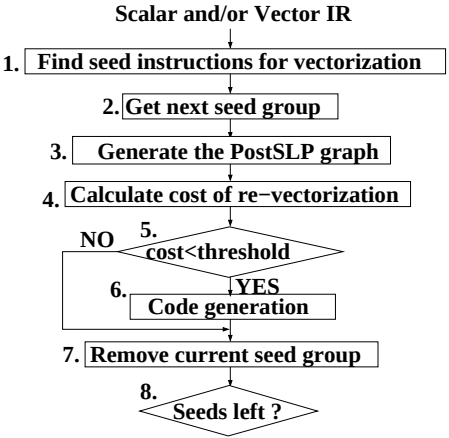
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses
- Cost: weighted instruction count
- Check overall profitability
- Generate code for groups



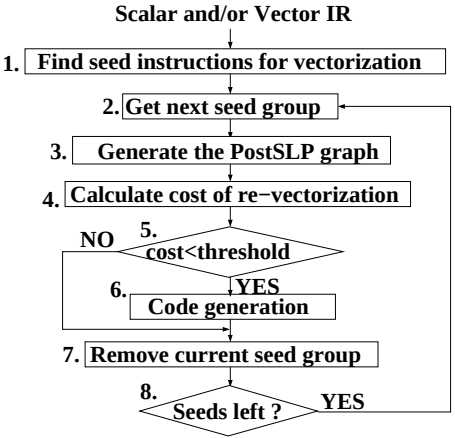
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses
- Cost: weighted instruction count
- Check overall profitability
- Generate code for groups



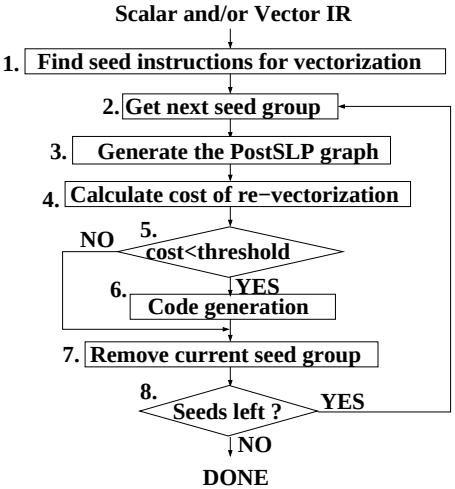
PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses
- Cost: weighted instruction count
- Check overall profitability
- Generate code for groups
- Repeat



PostSLP Algorithm

- Seeds: Consecutive Vector/Scalar Loads and Stores
- Grow vectorization graph towards defs and uses
- Cost: weighted instruction count
- Check overall profitability
- Generate code for groups
- 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
- SPEC CPU2006

Experimental Setup

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

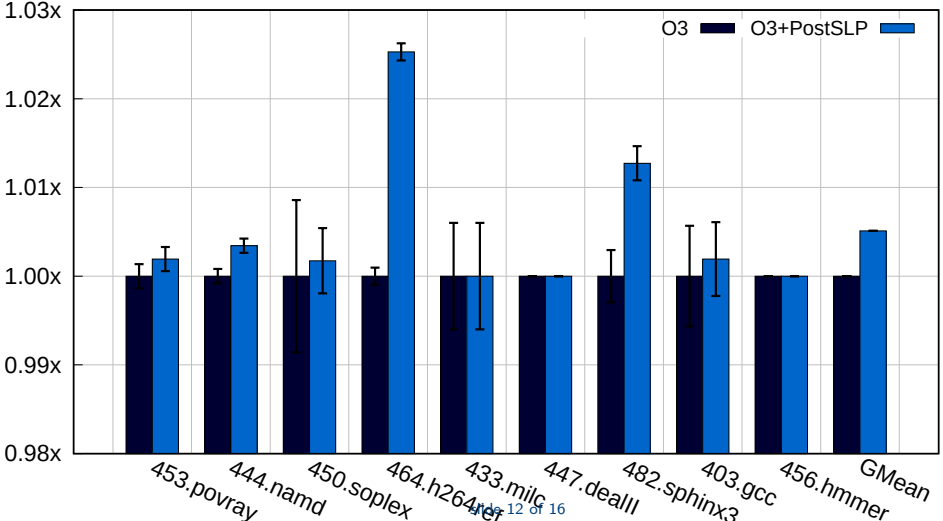
Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU
- Compiler flags: -O3 -ffast-math -march=native -mtune=native
- SPEC CPU2006
- We evaluated the following:
  1. O3 : All vectorizers enabled

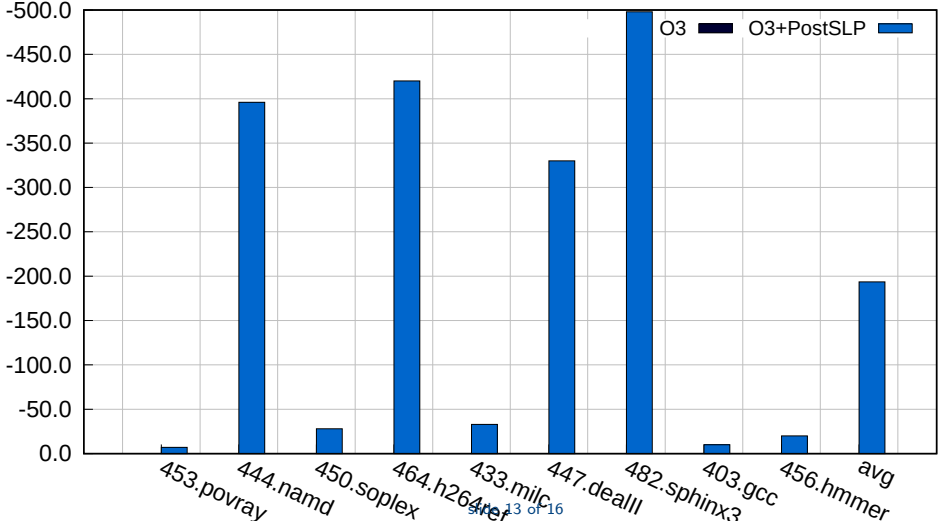
Experimental Setup

- Implemented in LLVM trunk
- Target: Intel® Core™ i5-6440HQ CPU
- Compiler flags: -O3 -ffast-math -march=native -mtune=native
- SPEC CPU2006
- We evaluated the following:
  1. O3 : All vectorizers enabled
  2. O3 + PostSLP : All vectorizers + PostSLP

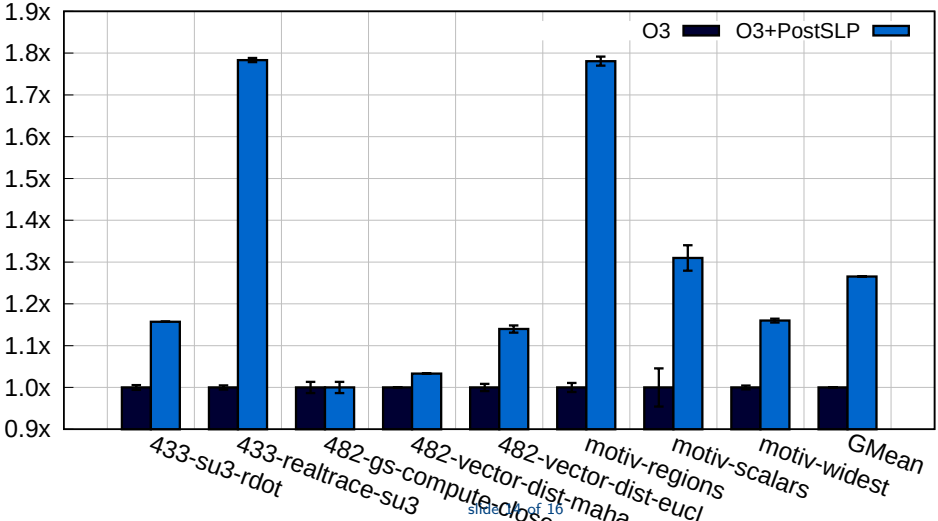
Up to 2.5% Faster in Full Benchmarks



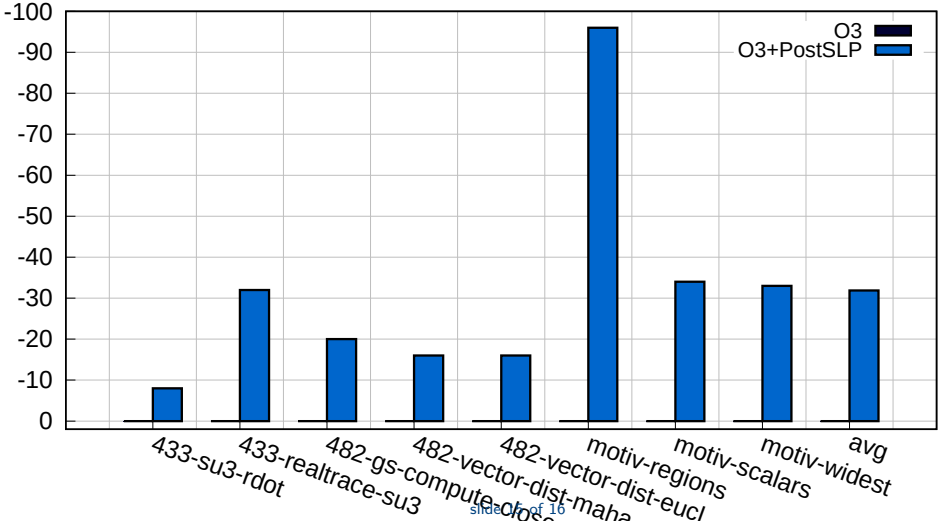
Static Cost Savings in 9 Full Benchmarks



Up to 1.8x Faster in Kernels



Static Cost Savings in Kernels



Conclusion

- Motivated the need for a post-vectorization pass

Conclusion

- Motivated the need for a post-vectorization pass
- Missed opportunities by SLP or LV

Conclusion

- Motivated the need for a post-vectorization pass
- Missed opportunities by SLP or LV
- PostSLP: A vectorization pass that can vectorize scalars and/or vectors

Conclusion

- Motivated the need for a post-vectorization pass
- Missed opportunities by SLP or LV
- PostSLP: A vectorization pass that can vectorize scalars and/or vectors
- Implementation in LLVM

Conclusion

- Motivated the need for a post-vectorization pass
- Missed opportunities by SLP or LV
- PostSLP: A vectorization pass that can vectorize scalars and/or vectors
- Implementation in LLVM
- Up to 2.5% speedup in Full SPEC CPU2006 benchmarks

Conclusion

- Motivated the need for a post-vectorization pass
- Missed opportunities by SLP or LV
- PostSLP: A vectorization pass that can vectorize scalars and/or vectors
- Implementation in LLVM
- Up to 2.5% speedup in Full SPEC CPU2006 benchmarks
- Negligible compilation time increase