Improving the Effectiveness and Generality of GCC Auto-Vectorization

Sameera Deshpande
(Imagination Technologies)
sameera.deshpande@imgtec.com

Prachi Godbole
(Imagination Technologies)
prachi.godbole@imgtec.com

Uday Khedker
(Indian Institute of Technology, Bombay)
uday@cse.iitb.ac.in

August 7, 2015

Overview

1. Introduction
   - Introduction to nomenclature
   - Target Instructions
2. Motivating Example
3. Limitations
4. Primitive Reordering Operations
5. Algorithm
6. Highlights
7. Future Work

Introduction - Autovectorization in GCC

- Loop vectorization - Inter-loop vectorization
  - Classic single statement vectorization
  - Interleaved data vectorization
- Superword Level Parallelism (SLP) - Sequencial vectorization
- Loop-aware SLP - Intra-loop vectorization

Loop vectorization - Classic

: Inter-loop vectorization

Source Code

```
FOR (i = 0; i < N; i++)
{
  D[i] = S1[i] + S2[i];
}
```

Transformed code

```
FOR (I=0; I < N/4; I++)
{
  VS1 = S1[4*I...4*I+3];
  VS2 = S2[4*I...4*I+3];
  VD[I] = VS1 + VS2;
}
```

Loop vectorization - Interleaved data

: Inter-loop vectorization

Source Code

```
FOR (i=0;i<N;i++)
{
  D[2*i] = S1[2*i+1] + 5;
  D[2*i+1] = S1[2*i] + 6;
}
```

Transformed code

```
FOR (I=0;I<N;I++)
{
  VS1 = S1[4*I,...,4*I+3]
  VS1od = extract_odd(VS1);
  VS1ev = extract_even(VS1);
  VT1od = VS1od + {5,5};
  VT1ev = VS1ev + {6,6};
  VD[I] = vec_permute(VT1od,
          VT1ev, {0,2,1,3});
}
```

Superword Level Parallelism(SLP)

: Sequencial vectorization

Source Code

```
FOR (i = 0; i < N; i++)
{
  D[4*i] = S1[4*i] + 1;
  D[4*i+1] = S1[4*i+1] + 3;
  D[4*i+2] = S1[4*i+2] + 2;
  D[4*i+3] = S1[4*i+3] + 4;
}
```

Transformed code

```
FOR (I=0; I < N; I++)
{
  VS1 = S1[4*I...4*I+3];
  VD[I] = VS1 + {1,3,2,4};
}
```

Loop-aware SLP

: Intra-loop vectorization

Source Code

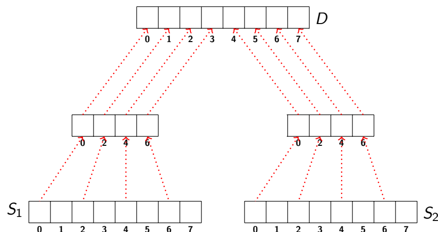
```
FOR (i=0;i<N;i++)
{
  D[2*i] = S1[2*i] + 5;
  D[2*i+1] = S1[2*i+1] + 6;
}
```

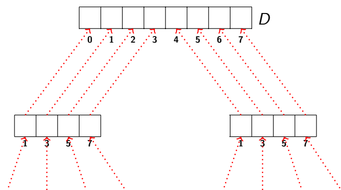
Transformed code

```
FOR (I=0; I < N/2; I++)
{
  VS1 = S1[4*I...4*I+3];
  VD[I] = VS1 + {5,6,5,6};
}
```

Nomenclature for target instructions

PCKEV - Pack even elements of two vectors PCKOD - Pack odd elements of two vectors ILVHI - Interleave upper half of two vectors ILVLO - Interleave lower half of two vectors





Target Specification

- Sequential access of memory.
- Each target vector register of size 4 words.
- Vector arithmetic and logic instructions are permute order preserving instructions.
- Target vector instructions altering permute order:
  - PCKEV - Pack even elements of two vector registers
  - PCKOD - Pack odd elements of two vector registers
  - ILVHI - Interleave upper half of two vector registers
  - ILVLO - Interleave lower half of two vector registers

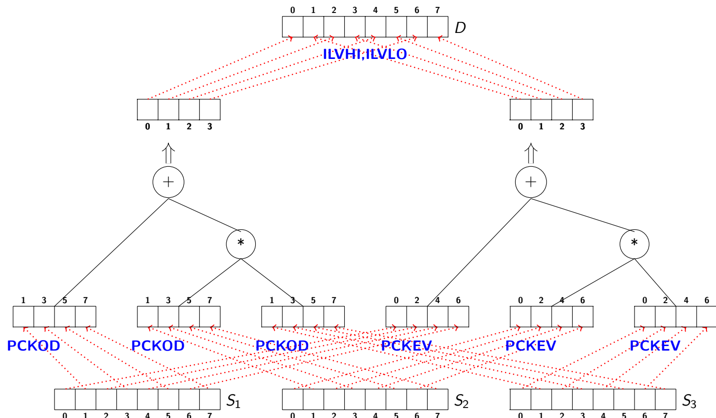
Motivating Example

```
for (i=0; i < N; i++)
{
  D[2*i] = S1[2*i+1] + S2[2*i+1] * S3[2*i+1];
  D[2*i+1] = S1[2*i] + S2[2*i] * S3[2*i];
}
```

Motivating Example

Interleaved data vectorization in GCC

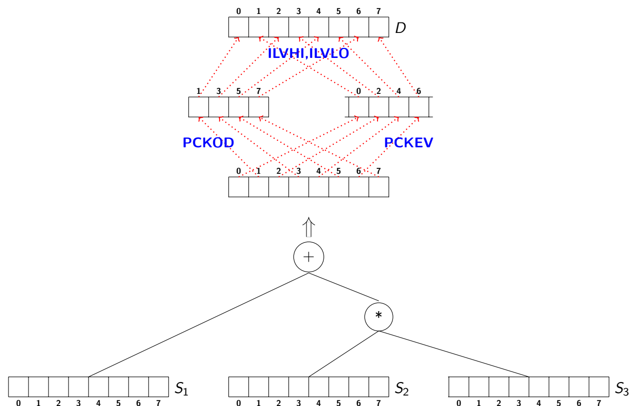
```
for (i=0; i < N; i++)
{
  D[2*i] = S1[2*i+1] + S2[2*i+1] * S3[2*i+1];
  D[2*i+1] = S1[2*i] + S2[2*i] * S3[2*i];

}
```



Motivating Example

Intuitive Solution

```
for (i=0; i < N; i++)
{
  D[2*i] = S1[2*i+1] + S2[2*i+1] * S3[2*i+1];
  D[2*i+1] = S1[2*i] + S2[2*i] * S3[2*i];

}
```



Motivating Example

Intuitive Solution

```
for (i=0; i < N; i++)
{
 D[2*i] = S1[2*i+1] * S2[2*i+1] + S3[2*i+1];
 D[2*i+1] = S1[2*i] * S2[2*i] + S3[2*i];

}
```

⇒

```
for (i=0; i < N; i++)
{
 T[2*i+1] = S1[2*i+1] * S2[2*i+1] + S3[2*i+1];
 T[2*i] = S1[2*i] * S2[2*i] + S3[2*i];
 D[2*i] = T[2*i+1];
 D[2*i+1] = T[2*i];

}
```

⇓

```
for (i=0; i < N; i++)
{
 T[2*i] = S1[2*i] * S2[2*i] + S3[2*i];
 T[2*i+1] = S1[2*i+1] * S2[2*i+1] + S3[2*i+1];
 D[2*i] = T[2*i+1];
 D[2*i+1] = T[2*i];

}
```

⇐

```
for (I=0; I < N/2; I++)
{
 VS1 = S1[4*I...4*I+3];
 VS2 = S2[4*I...4*I+3];
 VS3 = S3[4*I...4*I+3];
 VT = VS1 * VS2 + VS3;
 VT_ev = extract_even(VT);
 VT_od = extract_odd(VT);
 VD[I] = vec_permute(VT_od, VT_ev, {0,2,1,3});

}
```

Motivating Example

GCC vs Intuitive autovectorization

```
for (i=0; i < N; i++)
{
  D[2*i] = S1[2*i+1] * S2[2*i+1] + S3[2*i+1];
  D[2*i+1] = S1[2*i] * S2[2*i] + S3[2*i];

}
```

```
for (I=0; I < N/2; I++)
{
  VS1 = S1[4*I...4*I+3];
  VS2 = S2[4*I...4*I+3];
  VS3 = S3[4*I...4*I+3];
  VS1ev = extract_even(VS1);
  VS1od = extract_odd(VS1);
  VS2ev = extract_even(VS2);
  VS2od = extract_odd(VS2);
  VS3ev = extract_even(VS3);
  VS3od = extract_odd(VS3);
  VTev = VS1ev * VS2ev + VS3ev;
  VTod = VS1od * VS2od + VS3od;
  VD[I] = vec_permute(VTod,VTev,{0,2,1,3});

}
```

```
for (I=0; I < N/2; I++)
{
  VS1 = S1[4*I...4*I+3];
  VS2 = S2[4*I...4*I+3];
  VS3 = S3[4*I...4*I+3];
  VT = VS1 * VS2 + VS3;
  VTev = extract_even(VT);
  VTod = extract_odd(VT);
  VD[I] = vec_permute(VTod,VTev,{0,2,1,3});

}
```

Inherent Limitations of Autovectorization in GCC

- Machine Independent permute order generation.
- Loop aware SLP only for continous increasing memory accesses.
- Interleaved data vectorization has very rigid structure.
- No cost effective instruction selection.
- **No unified representation across all autovectorization algorithms.**
- **Loop information in source-code lost while enumerating permute order.**
- **Loop is either broken across body or across iterations. Integrated view not possible.**

Solution?

Unified Representation using four primitive reordering operations across autovectorization algorithms and target instructions, encapsulating loop information for better optimization

Loop Representation

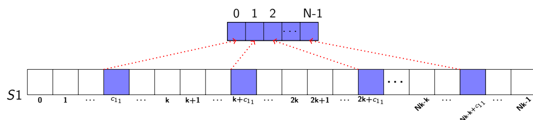
$$
\begin{aligned}
& FOR(i = 0; i < N; i++) \\
& \quad \{ \\
& \quad stmt_1 : D[k*i + d_1] = op_1(S_1[k*i + c_{1\,1}], \ldots S_m[k*i + c_{1\,m}]); \\
& \quad stmt_2 : D[k*i + d_2] = op_2(S_1[k*i + c_{2\,1}], \ldots S_m[k*i + c_{2\,m}]); \\
& \quad \ldots \\
& \quad stmt_k : D[k*i + d_k] = op_k(S_1[k*i + c_{k\,1}], \ldots S_m[k*i + c_{k\,m}]); \\
& \quad \}
\end{aligned}
$$

Primitive Reordering Operations

Extract

$$
\begin{aligned}
& FOR(i = 0; i < N; i++) \\
& \quad \{ \\
& \quad stmt_1 : \ldots = op_1(S_1[k * i + c_{1_1}], \ldots); \\
& \quad \}
\end{aligned}
$$

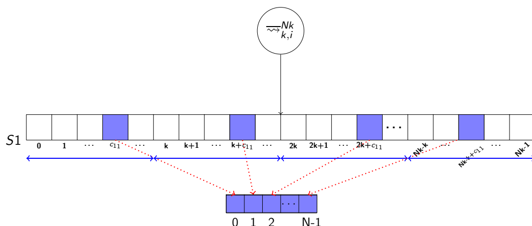
i=0 i=1 i=2 i=N-1



Primitive Reordering Operations

Extract

- $\mathbf{Extract^{Nk}_{k,i}}$: k-ary operation which divides source of size Nk into N parts, and accumulates i^{th} element from each part to form a vector of size N.

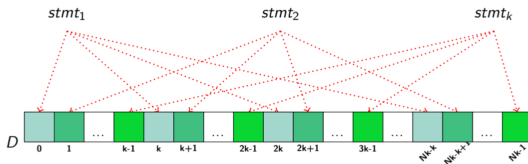


Primitive Reordering Operations

Interleave

$$
\begin{aligned}
&FOR(i = 0; i < N; i++)\\
&\quad \{\\
&\quad stmt_1 : D[k * i + 0] = \dots;\\
&\quad stmt_2 : D[k * i + 1] = \dots;\\
&\quad \dots\\
&\quad stmt_k : D[k * i + k - 1] = \dots;\\
&\quad \}
\end{aligned}
$$

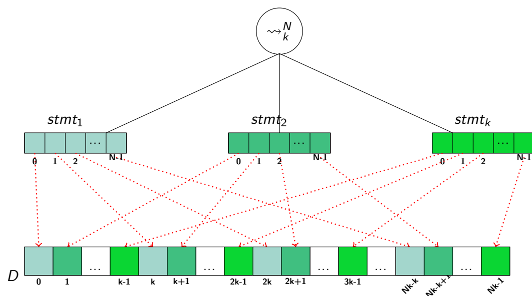
i=0 i=1 i=2 i=N-1



Primitive Reordering Operations

Interleave

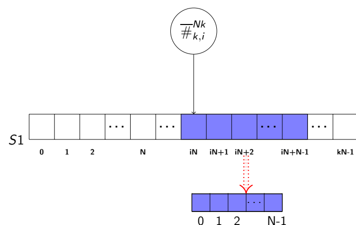
- **Interleave$_{\mathbf{k}}^{\mathbf{N}}$**: k-ary operation which takes k vectors of size N as input, and interleaves elements of each input such that $\left(\left(x_{i,j}\right)_{i=1}^{k}\right)_{j=1}^{N}$



Primitive Reordering Operations

Split

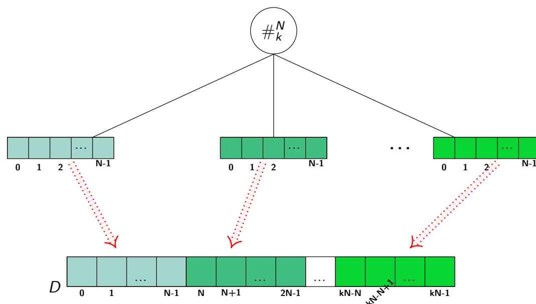
- $\mathbf{Split^{Nk}_{k,i}}$: k-ary operation which divides source of size Nk into k parts, and returns i^{th} part of size N.



Primitive Reordering Operations

Concatenate

- **Concatenate$_{\mathbf{k}}^{\mathbf{N}}$**: k-ary operation which takes k vectors of size N as input, and joins elements of each input one after the other.



Target Instructions Representation

- *PCKEV* : Pack even elements of vectors
- *PCKOD* : Pack odd elements of vectors
- *ILVLO* : Interleave lower elements of vectors
- *ILVHI* : Interleave higher elements of vectors

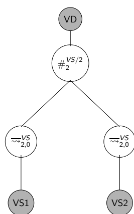


Figure: PCKEV

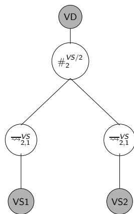


Figure: PCKOD

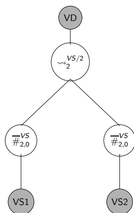


Figure: ILVLO

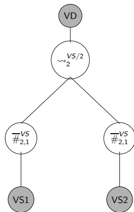


Figure: ILVHI

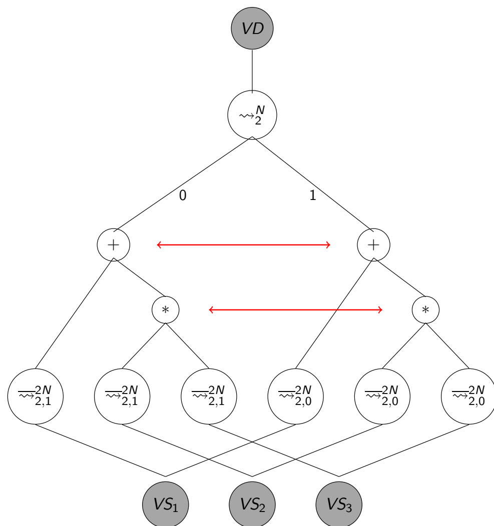
Algorithm

- Phase 1: Represent loop L as an AST T_d per destination D using combination of $\rightsquigarrow_k^N$, $\#_k^N$, $\overline{\#}_{k,i}^{kN}$ and $\overline{\rightsquigarrow}_{k,i}^{kN}$ operands.
- Phase 2: Target independent optimizations.
- Phase 3: Apply reduction/promotion rules on T_d such that arity k and vector size N of $\rightsquigarrow_k^N$, $\#_k^N$, $\overline{\#}_{k,i}^{kN}$ and $\overline{\rightsquigarrow}_{k,i}^{kN}$ operands reduced/promoted to arity m and vector size VS of target.
- Phase 4: Use tree-tiling algorithm for cost-effective instruction selection.

Representation of Motivating Example

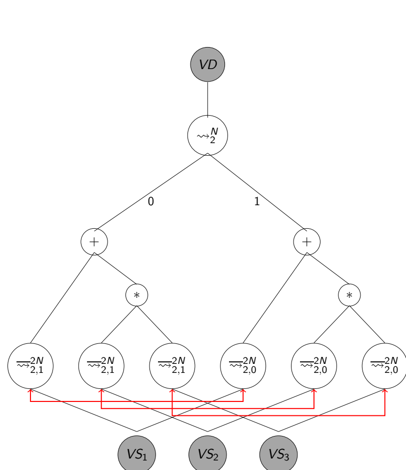
Phase 1: Translation of loop

```
for (i=0; i < N; i++)
{
  D[2*i] = S1[2*i+1] + S2[2*i+1] * S3[2*i+1];
  D[2*i+1] = S1[2*i] + S2[2*i] * S3[2*i];

}
```



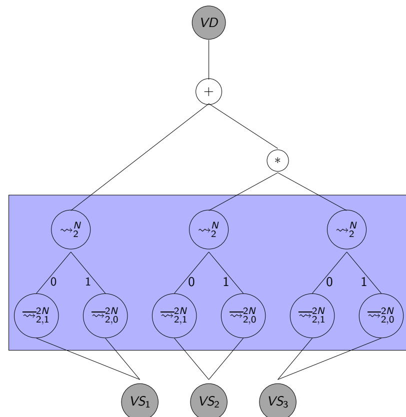
Representation of Motivating Example

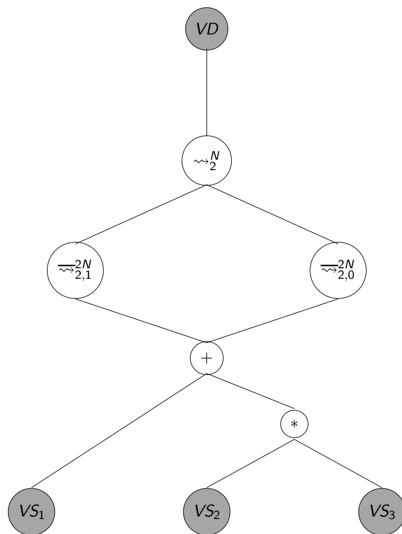
Phase 2: Target independent optimization



Representation of Motivating Example

Phase 2: Target independent optimization continued

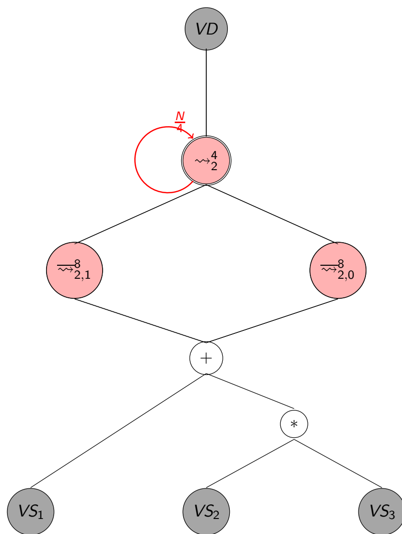




Representation of Motivating Example

Phase 3: Vectorsize Reduction from VS=N to VS=4

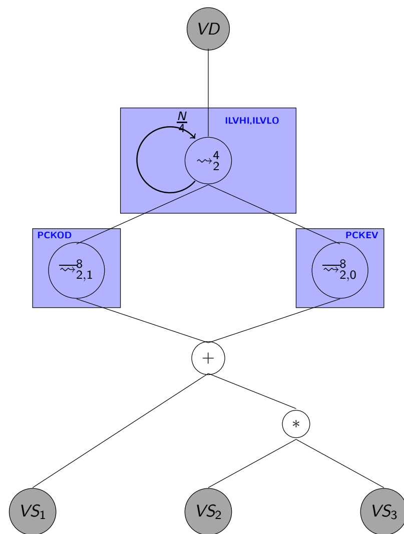
```
for (i=0; i < N; i++)
{
  D[2*i] = S1[2*i+1] + S2[2*i+1] * S3[2*i+1];
  D[2*i+1] = S1[2*i] + S2[2*i] * S3[2*i];

}
```



Representation of Motivating Example

Phase 4: Target instruction selection

```
for (i=0; i < N; i++)
{
  D[2*i] = S1[2*i+1] + S2[2*i+1] * S3[2*i+1];
  D[2*i+1] = S1[2*i] + S2[2*i] * S3[2*i];

}
```



Motivating Example

GCC vs our autovectorization

```
 for (i=0; i < N; i++)
{
  D[2*i] = S1[2*i+1] * S2[2*i+1] + S3[2*i+1];
  D[2*i+1] = S1[2*i] * S2[2*i] + S3[2*i];

}
```

```
for (I=0; I < N/2; I++)
{
  VS1 = S1[4*I...4*I+3];
  VS2 = S2[4*I...4*I+3];
  VS3 = S3[4*I...4*I+3];
  VS1ev = extract_even(VS1);
  VS1od = extract_odd(VS1);
  VS2ev = extract_even(VS2);
  VS2od = extract_odd(VS2);
  VS3ev = extract_even(VS3);
  VS3od = extract_odd(VS3);
  VTev = VS1ev * VS2ev + VS3ev;
  VTod = VS1od * VS2od + VS3od;
  VD[I] = vec_permute(VTod,VTev,{0,2,1,3});

}
```

```
for (I=0; I < N/2; I++)
{
  VS1 = S1[4*I...4*I+3];
  VS2 = S2[4*I...4*I+3];
  VS3 = S3[4*I...4*I+3];
  VT = VS1 * VS2 + VS3;
  VTev = extract_even(VT);
  VTod = extract_odd(VT);
  VD[I] = vec_permute(VTod,VTev,{0,2,1,3});

}
```

Another Example

GCC vs Intuitive autovectorization

```
for (i = 0; i ¡ N; i++)
{
  D[4*i+0] = S_1[4*i+0] + S_1[4*i+1];
  D[4*i+1] = S_1[4*i+2] + S_1[4*i+3];
  D[4*i+2] = S_2[4*i+0] + S_2[4*i+1];
  D[4*i+3] = S_2[4*i+2] + S_2[4*i+3];
}
```

Another Example

GCC vs Intuitive autovectorization

```
for (I=0; I < N/2; I++)
{
  VS1 = S1[8*I...8*I+7];
  VS2 = S2[8*I...8*I+7];
  VS1ev = extract_even(VS1);
  VS1od = extract_odd(VS1);
  VS2ev = extract_even(VS2);
  VS2od = extract_odd(VS2);
  VS10 = extract_even(VS1ev);
  VS11 = extract_odd(VS1od);
  VS20 = extract_even(VS2ev);
  VS21 = extract_odd(VS2od);
  VS12 = extract_even(VS1ev);
  VS13 = extract_odd(VS1od);
  VS22 = extract_even(VS2ev);
  VS23 = extract_odd(VS2od);
  VT0 = VS10 + VS11;
  VT1 = VS12 + VS13;
  VT2 = VS20 + VS21;
  VT3 = VS22 + VS23;
  ;
  VT02 = vec_permute(VT0, VT2, {0,2,1,3});
  VT13 = vec_permute(VT1, VT3, {0,2,1,3});
  VD[I] = vec_permute(VT02, VT13, {0,2,1,3});

}
```

```
for (I=0; I < N/2; I++)
{
  VS1 = S1[8*I...8*I+7];
  VS2 = S2[8*I...8*I+7];
  VS1ev = extract_even(VS1);
  VS1od = extract_odd(VS1);
  VS2ev = extract_even(VS2);
  VS2od = extract_odd(VS2);
  VTev = vec_permute(VS1ev,VS2ev,{0,1,4,5});
  VTod = vec_permute(VS1od,VS2od,{2,3,6,7});
  VD[I] = VTev + VTod;

}
```

Highlights

- **Whole loop** is represented using $\rightsquigarrow_k^N$ and $\overline{\rightsquigarrow}_{k,i}^{nk}$ operations only.
- All auto-vectorization transformations can be performed by **reduction/promotion rules**.
- All **primitive reordering** operations can be **moved across order-preserving operations** for better optimizations.
- **Target reordering instructions** can be represented using **primitive reordering** operations.
- **Cost aware reorder instruction selection** can be performed for effecient code generation.
- **Loop transformations** can be selectively performed based on target - using reduction/promotion rules.
- New vectorization algorithm can be added by introducing **new rules**.
- All previous **transformations are recorded** in representation.

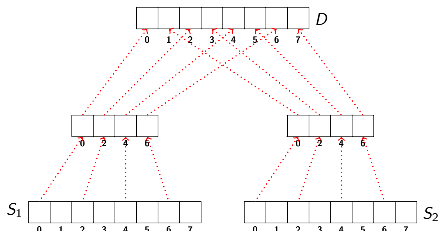
Way Ahead. . .

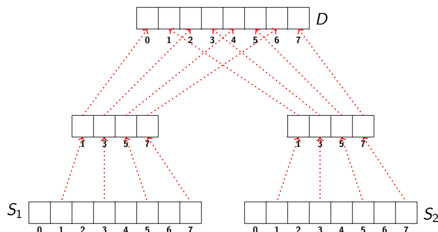
- Define conversion rules for primitive operations.
- Evolve Promotion/reduction rules for better optimizations.
- Improve cost-effective permute order selection algorithm based on tree-tiling.
- Eliminate current restrictions on acceptable loop, so that
  - Nested loops are optimized.
  - Multidimentional arrays are represented.
  - Reduction chains are representable.
  - Permissible dependences are handled gracefully.
  - Voids on destination are handled gracefully.
  - Irregular accesses are handled gracefully.
  - Conditional code within loop is handled gracefully.

Thank You!

Extra Slides

ILVEV - Interleave even elements of two vectors ILVOD - Interleave odd elements of two vectors





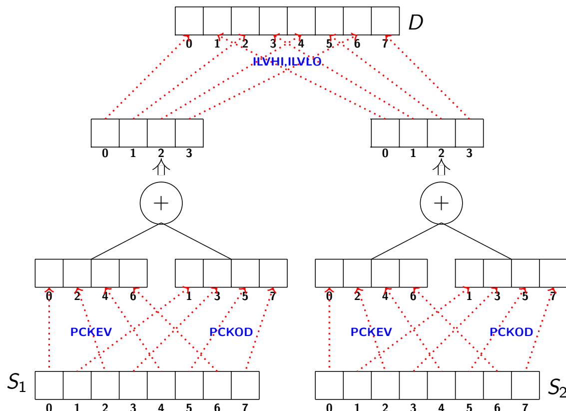
Example demonstrating target optimization

```
for (I=0; I<N; I++)
{
  D[2*I] = S1[2*I] + S1[2*I+1];
  D[2*I+1] = S2[2*I] + S2[2*I+1];
}
```

Example demonstrating target optimization

Interleaved data vectorization in GCC

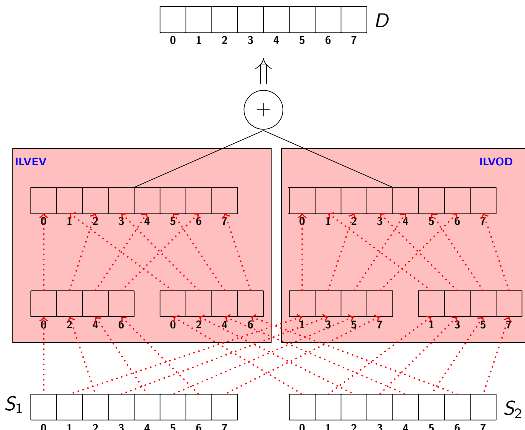
```
for (I=0; I<N; I++)
{
  D[2*I] = S1[2*I] + S1[2*I+1];
  D[2*I+1] = S2[2*I] + S2[2*I+1];

}
```



Example demonstrating target optimization

Intuitive Solution

```
for (I=0; I<N; I++)
{
  D[2*I] = S1[2*I] + S1[2*I+1];
  D[2*I+1] = S2[2*I] + S2[2*I+1];

}
```



Motivating Example

Intuitive Solution

```
for (I=0; I<N; I++)
{
  D[2*I] = S1[2*I] + S1[2*I+1];
  D[2*I+1] = S2[2*I] + S2[2*I+1];

}
```

⇒

```
for (I=0; I<N; I++)
{
  S1_ev = S1[2*I];
  S2_ev = S2[2*I];
  S1_od = S1[2*I+1];
  S2_od = S2[2*I+1];
  D[2*I] = S1_ev + S1_od;
  D[2*I+1] = S2_ev + S2_od;

}
```

⇓

```
for (I=0; I<N; I+=2)
{
  VS1_ev = S1[2*I,2*I+2];
  VS2_ev = S2[2*I,2*I+2];
  VS1_od = S1[2*I+1,2*I+3];
  VS2_od = S2[2*I+1,2*I+3];
  VD[I] = VS1_ev + VS1_od;
  VD[I+1] = VS2_ev + VS2_od;

}
```

⇐

```
for (I=0; I<N; I+=2)
{
  VS1 = S1[2*I...2*I+3];
  VS2 = S2[2*I...2*I+3];
  VT_ev = vec_permute(VS1, VS2,{0,2,4,6})
  VT_od = vec_permute(VS1, VS2,{1,3,5,7})
  VD[I] = VT_ev + VT_od;

}
```

Motivating Example

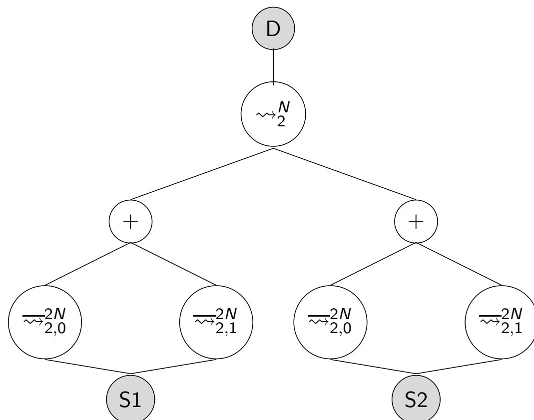
GCC vs Intuitive autovectorization

```
for (I=0; I<N; I++)
{
  D[2*I] = S1[2*I] + S1[2*I+1];
  D[2*I+1] = S2[2*I] + S2[2*I+1];

}
```

```
for (I=0; I<N; I+=2)
{
  VS1 = S1[2*I...2*I+3];
  VS2 = S2[2*I...2*I+3];
  VS1_ev = extract_even(VS1)
  VS1_od = extract_odd(VS1)
  VS2_ev = extract_even(VS2)
  VS2_od = extract_odd(VS2)
  VT_ev = VS1_ev + VS1_od;
  VT_od = VS2_ev + VS2_od;
  VT_ev = vec_permute(VT_ev, VT_od,{0,2,1,3})

}
```

```
for (I=0; I<N; I+=2)
{
  VS1 = S1[2*I...2*I+3];
  VS2 = S2[2*I...2*I+3];
  VT_ev = vec_permute(VS1, VS2,{0,2,4,6})
  VT_od = vec_permute(VS1, VS2,{1,3,5,7})
  VD[I] = VT_ev + VT_od;

}
```

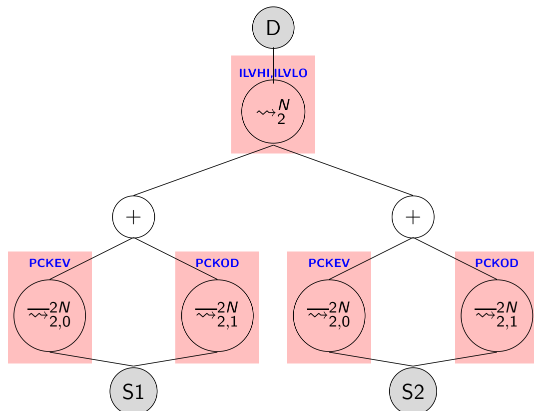
```
for (I=0; I<N; I++)
{
  D[2*I] = S1[2*I] + S1[2*I+1];
  D[2*I+1] = S2[2*I] + S2[2*I+1];

}
```



Representation of Motivating Example

Cost computation for order-tree

```
for (I=0; I<N; I++)
{
  D[2*I] = S1[2*I] + S1[2*I+1];
  D[2*I+1] = S2[2*I] + S2[2*I+1];

}
```



Optimized loop

```
for (I=0; I<N; I++)
{
  D[2*I] = S1[2*I] + S1[2*I+1];
  D[2*I+1] = S2[2*I] + S2[2*I+1];

}
```