Workshop on Essential Abstractions in GCC

Parallelization and Vectorization in GCC

GCC Resource Center

(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

3 July 2011

Outline

- An Overview of Loop Transformations in GCC
- Parallelization and Vectorization based on Lambda Framework
- Loop Transformations in Polytope Model
- Conclusions

Part 1

Parallelization and Vectorization in GCC using Lambda Framework

Loop Transforms in GCC

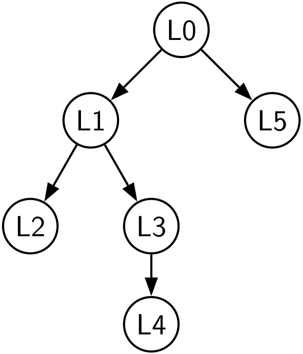
Implementation Issues

- Getting loop information (Loop discovery)
- Finding value spaces of induction variables, array subscript functions, and pointer accesses
- Analyzing data dependence
- Performing linear transformations

Loop Information

```
Loop0
{   Loop1
    {   Loop2
        {
        }
        Loop3
        {  Loop4
           {
           }
        }
    }
    Loop5
    {
    }
}
```

Loop Tree



Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
     struct opt_pass **p = &pass_tree_loop.pass.sub;
     NEXT_PASS (pass_tree_loop_init);
     NEXT_PASS (pass_lim);
     ...
     NEXT_PASS (pass_check_data_deps);
     NEXT_PASS (pass_loop_distribution);
     NEXT_PASS (pass_copy_prop);
     NEXT_PASS (pass_graphite);
       {
          struct opt_pass **p = &pass_graphite.pass.sub;
          NEXT_PASS (pass_graphite_transforms);
          ...
       }
     NEXT_PASS (pass_iv_canon);
     NEXT_PASS (pass_if_conversion);
     NEXT_PASS (pass_vectorize);
       {
          struct opt_pass **p = &pass_vectorize.pass.sub;
          NEXT_PASS (pass_lower_vector_ssa);
          NEXT_PASS (pass_dce_loop);
       }
     NEXT_PASS (pass_predcom);
     NEXT_PASS (pass_complete_unroll);
     NEXT_PASS (pass_slp_vectorize);
     NEXT_PASS (pass_parallelize_loops);
     NEXT_PASS (pass_loop_prefetch);
     NEXT_PASS (pass_iv_optimize);
     NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
      struct opt_pass **p = &pass_tree_loop.pass.sub;
      NEXT_PASS (pass_tree_loop_init);
      NEXT_PASS (pass_lim);
      ...
      NEXT_PASS (pass_check_data_deps);
      NEXT_PASS (pass_loop_distribution);
      NEXT_PASS (pass_copy_prop);
      NEXT_PASS (pass_graphite);
        {
          struct opt_pass **p = &pass_graphite.pass.sub;
          NEXT_PASS (pass_graphite_transforms);
          ...
        }
      NEXT_PASS (pass_iv_canon);
      NEXT_PASS (pass_if_conversion);
      NEXT_PASS (pass_vectorize);
        {
          struct opt_pass **p = &pass_vectorize.pass.sub;
          NEXT_PASS (pass_lower_vector_ssa);
          NEXT_PASS (pass_dce_loop);
        }
      NEXT_PASS (pass_predcom);
      NEXT_PASS (pass_complete_unroll);
      NEXT_PASS (pass_slp_vectorize);
      NEXT_PASS (pass_parallelize_loops);
      NEXT_PASS (pass_loop_prefetch);
      NEXT_PASS (pass_iv_optimize);
      NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
    struct opt_pass **p = &pass_tree_loop.pass.sub;
    NEXT_PASS (pass_tree_loop_init);
    NEXT_PASS (pass_lim);
    ...
    NEXT_PASS (pass_check_data_deps);
    NEXT_PASS (pass_loop_distribution);
    NEXT_PASS (pass_copy_prop);
    NEXT_PASS (pass_graphite);
      {
        struct opt_pass **p = &pass_graphite.pass.sub;
        NEXT_PASS (pass_graphite_transforms);
        ...
      }
    NEXT_PASS (pass_iv_canon);
    NEXT_PASS (pass_if_conversion);
    NEXT_PASS (pass_vectorize);
      {
        struct opt_pass **p = &pass_vectorize.pass.sub;
        NEXT_PASS (pass_lower_vector_ssa);
        NEXT_PASS (pass_dce_loop);
      }
    NEXT_PASS (pass_predcom);
    NEXT_PASS (pass_complete_unroll);
    NEXT_PASS (pass_slp_vectorize);
    NEXT_PASS (pass_parallelize_loops);
    NEXT_PASS (pass_loop_prefetch);
    NEXT_PASS (pass_iv_optimize);
    NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC: Our Focus

Data Dependence	Pass variable name	`pass_check_data_deps`
	Enabling switch	`-fcheck-data-deps`
	Dump switch	`-fdump-tree-ckdd`
	Dump file extension	`.ckdd`
Loop Distribution	Pass variable name	`pass_loop_distribution`
	Enabling switch	`-ftree-loop-distribution`
	Dump switch	`-fdump-tree-ldist`
	Dump file extension	`.ldist`
Vectorization	Pass variable name	`pass_vectorize`
	Enabling switch	`-ftree-vectorize`
	Dump switch	`-fdump-tree-vect`
	Dump file extension	`.vect`
Parallelization	Pass variable name	`pass_parallelize_loops`
	Enabling switch	`-ftree-parallelize-loops=n`
	Dump switch	`-fdump-tree-parloops`
	Dump file extension	`.parloops`

Compiling for Emitting Dumps

- Other necessary command line switches
  - `-O3 -fdump-tree-all`
    `-O3` enables `-ftree-vectorize`. Other flags must be enabled explicitly
- Processor related switches to enable transformations apart from analysis
  - `-mtune=pentium -msse4`
- Other useful options
  - Suffixing `-all` to all dump switches
  - `-S` to stop the compilation with assembly generation
  - `--verbose-asm` to see more detailed assembly dump

Representing Value Spaces of Variables and Expressions

Chain of Recurrences: 3-tuple $\langle$Starting Value, modification, stride$\rangle$

```
for (i=3; i<=15; i=i+3)
{
    for (j=11; j>=1; j=j-2)
    {
        A[i+1][2*j-1] = ...
    }
}
```

Entity	CR
Induction variable `i`	$\{3, +, 3\}$
Induction variable `j`	$\{11, +, -2\}$
Index expression `i+1`	$\{4, +, 3\}$
Index expression `2*j-1`	$\{21, +, -4\}$

Advantages of Chain of Recurrences

CR can represent any affine expression

⇒ Accesses through pointers can also be tracked

```
int A[256], B[256];
int i, *p;
p = B;
for(i=1; i<200; i++)
{
  *(p++) = A[i] + *p;
  A[i] = *p;
}
```

Advantages of Chain of Recurrences

CR can represent any affine expression

⇒ Accesses through pointers can also be tracked

```
int A[256], B[256];
int i, *p;
p = B;
for(i=1; i<200; i++)
{
  *(p++) = A[i] + *p;
  A[i] = *p;
}
```

{&B,+,4bytes}

Advantages of Chain of Recurrences

CR can represent any affine expression
⇒ Accesses through pointers can also be tracked

```
int A[256], B[256];
int i, *p;
p = B;
for(i=1; i<200; i++)
{
  *(p++) = A[i] + *p;
  A[i] = *p;
}
```

{&B,+,4bytes}

{&B+4bytes,+,4bytes}

Example 1: Observing Data Dependence

Step 0: Compiling

```
int a[200];
int main()
{
   int i;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

```
gcc -fcheck-data-deps -fdump-tree-ckdd-all -O3 -S datadep.c
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program

```
int a[200];
int main()
{
   int i;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

<pre>int a[200];
int main()
{
    int i;
    for (i=0; i<150; i++)
    {
        a[i] = a[i+1] + 2;
    }
    return 0;
}</pre>
	<pre><bb 3>:
	  # i_13 = PHI <i_3(4), 0(2)>
	  i_3 = i_13 + 1;
	  D.1955_4 = a[i_3];
	  D.1956_5 = D.1955_4 + 2;
	  a[i_13] = D.1956_5;
	  if (i_3 != 150)
	    goto <bb 4>;
	  else
	    goto <bb 5>;
	<bb 4>:
	  goto <bb 3>;</pre>

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program

```
int a[200];
int main()
{
    int i;
    for (i=0; i<150; i++)
    {
        a[i] = a[i+1] + 2;
    }
    return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

<pre>int a[200];
int main()
{
   int i;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}</pre>
	<pre><bb 3>:
	  # i_13 = PHI <i_3(4), 0(2)>
	  i_3 = i_13 + 1;
	  D.1955_4 = a[i_3];
	  D.1956_5 = D.1955_4 + 2;
	  a[i_13] = D.1956_5;
	  if (i_3 != 150)
	    goto <bb 4>;
	  else
	    goto <bb 5>;
	<bb 4>:
	  goto <bb 3>;</pre>

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
(scalar_evolution = {0, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
(scalar_evolution = {1, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                 address: 4
aligned to: 128
(chrec = {1, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                 address:  0
aligned to: 128
base_object: a[0]
(chrec = {0, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View
• Relevant assignment is `a[i]` $=$ `a[i+1]` $+ 2$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View
• Relevant assignment is `a[i]` $=$ `a[i+1]` $+ 2$ • Solve for $0 \leq x, y < 150$ $$y = x + 1$$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View
• Relevant assignment is `a[i]` $=$ `a[i+1]` $+ 2$ • Solve for $0 \leq x, y < 150$ $y = x + 1$ $\Rightarrow x - y + 1 = 0$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View

- Relevant assignment is
  `a[i]` $=$ `a[i+1]` $+ 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View

- Relevant assignment is
  `a[i]` $=$ `a[i+1]` $+ 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$$\boxed{x - y + 1}$$

Min: -148 Max: +150

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View

- Relevant assignment is
  `a[i] = a[i+1] + 2`
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$$\boxed{x - y + 1}$$

Min: -148 Max: $+150$

RHS belongs to $[-148, +150]$
and dependence may exist

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is
  `a[i]` $=$ `a[i+1]` $+ 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x-y+1 &= 0 \end{aligned}$$

- Find min and max of LHS

$\boxed{x - y + 1}$

Min: -148 Max: $+150$

RHS belongs to $[-148, +150]$
and dependence may exist

CFG View

- ```
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  ```
  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is
  a[i] = a[i+1] + 2
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$$\boxed{x - y + 1}$$

Min: -148 Max: +150

RHS belongs to $[-148, +150]$
and dependence may exist

CFG View

- ```
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  ```
- Chain of recurrences are
  For a[i_3]: {1, +, 1}_1
  For a[i_13]: {0, +, 1}_1
  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is

  a[i] $=$ a[i+1] $+ 2$

- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$$\boxed{x - y + 1}$$

Min: -148 Max: $+150$

RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  ```

- Chain of recurrences are

  For `a[i_3]`: `{1, +, 1}_1`

  For `a[i_13]`: `{0, +, 1}_1`

- Solve for $0 \leq$ `x_1` < 150

  `1 + 1*x_1` $-$ `0 + 1*x_1` $= 0$
  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is
  a[i] = a[i+1] + 2
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$$\boxed{x - y + 1}$$

Min: -148 Max: +150

RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  ```
- Chain of recurrences are
  For `a[i_3]`: `{1, +, 1}_1`
  For `a[i_13]`: `{0, +, 1}_1`
- Solve for $0 \leq$ `x_1` < 150
  `1 + 1*x_1` $-$ `0 + 1*x_1` $= 0$
- Min of LHS is -148, Max is +150
  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is
  a[i] $=$ a[i+1] $+ 2$
- Solve for $0 \le x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$$\boxed{x - y + 1}$$

Min: -148 Max: $+150$

RHS belongs to $[-148, +150]$
and dependence may exist

CFG View

- ```
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  ```
- Chain of recurrences are
  For a[i_3]: {1, +, 1}_1
  For a[i_13]: {0, +, 1}_1
- Solve for $0 \le$ x_1 < 150
  1 + 1*x_1 $-$ 0 + 1*x_1 $= 0$
- Min of LHS is -148, Max is $+150$
- Dependence may exist
  ```

Example 1: Observing Data Dependence

Step 4: Observing the data dependence information

```
iterations_that_access_an_element_twice_in_A: [1 + 1 * x_1]
last_conflict: 149
iterations_that_access_an_element_twice_in_B: [0 + 1 * x_1]
last_conflict: 149
Subscript distance: 1
```

```
inner loop index: 0
loop nest: (1)
distance_vector: 1
direction_vector: +
```

Example 2: Observing Vectorization and Parallelization

Step 0: Compiling the code with `-O3`

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

- Additional options for parallelization
  `-ftree-parallelize-loops=2 -fdump-tree-parloops-all`
- Additional options for vectorization
  `-fdump-tree-vect-all -msse4`

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.2836_3 = b[i_11];
  a[i_11] = D.2836_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.2836_3 = b[i_11];
  a[i_11] = D.2836_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

<pre>int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}</pre>
	<pre><bb 3>:
	  # i_11 = PHI <i_4(4), 0(2)>
	  D.2836_3 = b[i_11];
	  a[i_11] = D.2836_3;
	  i_4 = i_11 + 1;
	  if (i_4 != 256)
	    goto <bb 4>;
	  else
	    goto <bb 5>;
	<bb 4>:
	  goto <bb 3>;</pre>

Example 2: Observing Vectorization and Parallelization

Step 2: Observing the final decision about vectorization

```
parvec.c:5: note: LOOP VECTORIZED.
parvec.c:2: note: vectorized 1 loops in function.
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.2836_3 = b[i_11];
  a[i_11] = D.2836_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.7_10 = &b;
  vect_pa.12_15 = &a;
<bb 3>:
  # vect_pb.4_6 = PHI <vect_pb.4_13,
                        vect_pb.7_10>
  # vect_pa.9_16 = PHI <vect_pa.9_17,
                        vect_pa.12_15>
  vect_var_.8_14 = MEM[vect_pb.4_6];
  MEM[vect_pa.9_16] = vect_var_.8_14;
  vect_pb.4_13 = vect_pb.4_6 + 16;
  vect_pa.9_17 = vect_pa.9_16 + 16;
  ivtmp.13_19 = ivtmp.13_18 + 1;
  if (ivtmp.13_19 < 64)
    goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph	Transformed control flow graph

<pre><bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.2836_3 = b[i_11];
  a[i_11] = D.2836_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;</pre>
	<pre><bb 2>:
	  vect_pb.7_10 = &b;
	  vect_pa.12_15 = &a;
	<bb 3>:
	  # vect_pb.4_6 = PHI <vect_pb.4_13,
	                    vect_pb.7_10>
	  # vect_pa.9_16 = PHI <vect_pa.9_17,
	                    vect_pa.12_15>
	  vect_var_.8_14 = MEM[vect_pb.4_6];
	  MEM[vect_pa.9_16] = vect_var_.8_14;
	  vect_pb.4_13 = vect_pb.4_6 + 16;
	  vect_pa.9_17 = vect_pa.9_16 + 16;
	  ivtmp.13_19 = ivtmp.13_18 + 1;
	  if (ivtmp.13_19 < 64)
	    goto <bb 4>;</pre>

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.2836_3 = b[i_11];
  a[i_11] = D.2836_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.7_10 = &b;
  vect_pa.12_15 = &a;
<bb 3>:
  # vect_pb.4_6 = PHI <vect_pb.4_13,
                  vect_pb.7_10>
  # vect_pa.9_16 = PHI <vect_pa.9_17,
                  vect_pa.12_15>
  vect_var_.8_14 = MEM[vect_pb.4_6];
  MEM[vect_pa.9_16] = vect_var_.8_14;
  vect_pb.4_13 = vect_pb.4_6 + 16;
  vect_pa.9_17 = vect_pa.9_16 + 16;
  ivtmp.13_19 = ivtmp.13_18 + 1;
  if (ivtmp.13_19 < 64)
    goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.2836_3 = b[i_11];
  a[i_11] = D.2836_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.7_10 = &b;
  vect_pa.12_15 = &a;
<bb 3>:
  # vect_pb.4_6 = PHI <vect_pb.4_13,
                       vect_pb.7_10>
  # vect_pa.9_16 = PHI <vect_pa.9_17,
                       vect_pa.12_15>
  vect_var_.8_14 = MEM[vect_pb.4_6];
  MEM[vect_pa.9_16] = vect_var_.8_14;
  vect_pb.4_13 = vect_pb.4_6 + 16;
  vect_pa.9_17 = vect_pa.9_16 + 16;
  ivtmp.13_19 = ivtmp.13_18 + 1;
  if (ivtmp.13_19 < 64)
    goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.2836_3 = b[i_11];
  a[i_11] = D.2836_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.7_10 = &b;
  vect_pa.12_15 = &a;
<bb 3>:
  # vect_pb.4_6 = PHI <vect_pb.4_13,
                       vect_pb.7_10>
  # vect_pa.9_16 = PHI <vect_pa.9_17,
                       vect_pa.12_15>
  vect_var_.8_14 = MEM[vect_pb.4_6];
  MEM[vect_pa.9_16] = vect_var_.8_14;
  vect_pb.4_13 = vect_pb.4_6 + 16;
  vect_pa.9_17 = vect_pa.9_16 + 16;
  ivtmp.13_19 = ivtmp.13_18 + 1;
  if (ivtmp.13_19 < 64)
    goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
     /* execute the loop body to be parallelized */
}
```

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
     /* execute the loop body to be parallelized */
}
```

- All threads are executed in parallel

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Get the number of threads

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Get thread identity

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Perform load calculations

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Assign start iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Assign end iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Start execution of iterations of the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.1956_3 = b[i_11];
  a[i_11] = D.1956_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 5>:
  i.8_21 = (int) ivtmp.7_18;
  D.2010_23 = *b.10_4[i.8_21];
  *a.11_5[i.8_21] = D.2010_23;
  ivtmp.7_19 = ivtmp.7_18 + 1;
  if (D.2006_16 > ivtmp.7_19)
    goto <bb 5>;
  else
    goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.1956_3 = b[i_11];
  a[i_11] = D.1956_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 5>:
  i.8_21 = (int) ivtmp.7_18;
  D.2010_23 = *b.10_4[i.8_21];
  *a.11_5[i.8_21] = D.2010_23;
  ivtmp.7_19 = ivtmp.7_18 + 1;
  if (D.2006_16 > ivtmp.7_19)
    goto <bb 5>;
  else
    goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.1956_3 = b[i_11];
  a[i_11] = D.1956_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 5>:
  i.8_21 = (int) ivtmp.7_18;
  D.2010_23 = *b.10_4[i.8_21];
  *a.11_5[i.8_21] = D.2010_23;
  ivtmp.7_19 = ivtmp.7_18 + 1;
  if (D.2006_16 > ivtmp.7_19)
    goto <bb 5>;
  else
    goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

```<bb 3>:```
``` # i_11 = PHI <i_4(4), 0(2)>```
``` D.1956_3 = b[i_11];```
``` a[i_11] = D.1956_3;```
``` i_4 = i_11 + 1;```
``` if (i_4 != 256)```
```   goto <bb 4>;```
``` else```
```   goto <bb 5>;```
```<bb 4>:```
``` goto <bb 3>;```
	```<bb 5>:```
	``` i.8_21 = (int) ivtmp.7_18;```
	``` D.2010_23 = *b.10_4[i.8_21];```
	``` *a.11_5[i.8_21] = D.2010_23;```
	``` ivtmp.7_19 = ivtmp.7_18 + 1;```
	``` if (D.2006_16 > ivtmp.7_19)```
	``` goto <bb 5>;```
	``` else```
	``` goto <bb 3>;```

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.1956_3 = b[i_11];
  a[i_11] = D.1956_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 5>:
  i.8_21 = (int) ivtmp.7_18;
  D.2010_23 = *b.10_4[i.8_21];
  *a.11_5[i.8_21] = D.2010_23;
  ivtmp.7_19 = ivtmp.7_18 + 1;
  if (D.2006_16 > ivtmp.7_19)
    goto <bb 5>;
  else
    goto <bb 3>;
```

Example 3: Vectorization but No Parallelization

Step 0: Compiling with

`-O3 -fdump-tree-vect-all -msse4`

```
int a[624];
int main()
{
    int i;
    for (i=0; i<619; i++)
    {
        a[i] = a[i+4];
    }
    return 0;
}
```

Example 3: Vectorization but No Parallelization

Step 1: Observing the final decision about vectorization

```
vecnopar.c:5: note: LOOP VECTORIZED.
vecnopar.c:2: note: vectorized 1 loops in function.
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_5(4), 0(2)>
  D.2834_3 = i_12 + 4;
  D.2835_4 = a[D.2834_3];
  a[i_12] = D.2835_4;
  i_5 = i_12 + 1;
  if (i_5 != 619)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 2>:
  vect_pa.10_26 = &a[4];
  vect_pa.15_30 = &a;
<bb 3>:
  # vect_pa.7_27 = PHI <vect_pa.7_28,
                        vect_pa.10_26>
  # vect_pa.12_31 = PHI <vect_pa.12_32,
                        vect_pa.15_30>
  vect_var_.11_29 = MEM[vect_pa.7_27];
  MEM[vect_pa.12_31] = vect_var_.11_29;
  vect_pa.7_28 = vect_pa.7_27 + 16;
  vect_pa.12_32 = vect_pa.12_31 + 16;
  ivtmp.16_34 = ivtmp.16_33 + 1;
  if (ivtmp.16_34 < 154)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_5(4), 0(2)>
  D.2834_3 = i_12 + 4;
  D.2835_4 = a[D.2834_3];
  a[i_12] = D.2835_4;
  i_5 = i_12 + 1;
  if (i_5 != 619)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 2>:
  vect_pa.10_26 = &a[4];
  vect_pa.15_30 = &a;
<bb 3>:
  # vect_pa.7_27 = PHI <vect_pa.7_28,
                        vect_pa.10_26>
  # vect_pa.12_31 = PHI <vect_pa.12_32,
                        vect_pa.15_30>
  vect_var_.11_29 = MEM[vect_pa.7_27];
  MEM[vect_pa.12_31] = vect_var_.11_29;
  vect_pa.7_28 = vect_pa.7_27 + 16;
  vect_pa.12_32 = vect_pa.12_31 + 16;
  ivtmp.16_34 = ivtmp.16_33 + 1;
  if (ivtmp.16_34 < 154)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_5(4), 0(2)>
  D.2834_3 = i_12 + 4;
  D.2835_4 = a[D.2834_3];
  a[i_12] = D.2835_4;
  i_5 = i_12 + 1;
  if (i_5 != 619)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 2>:
  vect_pa.10_26 = &a[4];
  vect_pa.15_30 = &a;
<bb 3>:
  # vect_pa.7_27 = PHI <vect_pa.7_28,
                        vect_pa.10_26>
  # vect_pa.12_31 = PHI <vect_pa.12_32,
                        vect_pa.15_30>
  vect_var_.11_29 = MEM[vect_pa.7_27];
  MEM[vect_pa.12_31] = vect_var_.11_29;
  vect_pa.7_28 = vect_pa.7_27 + 16;
  vect_pa.12_32 = vect_pa.12_31 + 16;
  ivtmp.16_34 = ivtmp.16_33 + 1;
  if (ivtmp.16_34 < 154)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph	Vectorized Control Flow Graph

<pre><bb 3>:
  # i_12 = PHI <i_5(4), 0(2)>
  D.2834_3 = i_12 + 4;
  D.2835_4 = a[D.2834_3];
  a[i_12] = D.2835_4;
  i_5 = i_12 + 1;
  if (i_5 != 619)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;</pre>
	<pre><bb 2>:
	  vect_pa.10_26 = &a[4];
	  vect_pa.15_30 = &a;
	<bb 3>:
	  # vect_pa.7_27 = PHI <vect_pa.7_28,
	                        vect_pa.10_26>
	  # vect_pa.12_31 = PHI <vect_pa.12_32,
	                        vect_pa.15_30>
	  vect_var_.11_29 = MEM[vect_pa.7_27];
	  MEM[vect_pa.12_31] = vect_var_.11_29;
	  vect_pa.7_28 = vect_pa.7_27 + 16;
	  vect_pa.12_32 = vect_pa.12_31 + 16;
	  ivtmp.16_34 = ivtmp.16_33 + 1;
	  if (ivtmp.16_34 < 154)
	    goto <bb 4>;</pre>

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_5(4), 0(2)>
  D.2834_3 = i_12 + 4;
  D.2835_4 = a[D.2834_3];
  a[i_12] = D.2835_4;
  i_5 = i_12 + 1;
  if (i_5 != 619)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 2>:
  vect_pa.10_26 = &a[4];
  vect_pa.15_30 = &a;
<bb 3>:
  # vect_pa.7_27 = PHI <vect_pa.7_28,
                        vect_pa.10_26>
  # vect_pa.12_31 = PHI <vect_pa.12_32,
                        vect_pa.15_30>
  vect_var_.11_29 = MEM[vect_pa.7_27];
  MEM[vect_pa.12_31] = vect_var_.11_29;
  vect_pa.7_28 = vect_pa.7_27 + 16;
  vect_pa.12_32 = vect_pa.12_31 + 16;
  ivtmp.16_34 = ivtmp.16_33 + 1;
  if (ivtmp.16_34 < 154)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_5(4), 0(2)>
  D.2834_3 = i_12 + 4;
  D.2835_4 = a[D.2834_3];
  a[i_12] = D.2835_4;
  i_5 = i_12 + 1;
  if (i_5 != 619)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 2>:
  vect_pa.10_26 = &a[4];
  vect_pa.15_30 = &a;
<bb 3>:
  # vect_pa.7_27 = PHI <vect_pa.7_28,
                       vect_pa.10_26>
  # vect_pa.12_31 = PHI <vect_pa.12_32,
                       vect_pa.15_30>
  vect_var_.11_29 = MEM[vect_pa.7_27];
  MEM[vect_pa.12_31] = vect_var_.11_29;
  vect_pa.7_28 = vect_pa.7_27 + 16;
  vect_pa.12_32 = vect_pa.12_31 + 16;
  ivtmp.16_34 = ivtmp.16_33 + 1;
  if (ivtmp.16_34 < 154)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

- Step 3: Observing the conclusion about dependence information

```
inner loop index: 0
loop nest: (1 )
distance_vector: 4
direction_vector: +
```

- Step 4: Observing the final decision about parallelization

```
FAILED: data dependencies exist across iterations
```

Example 4: No Vectorization and No Parallelization

Step 0: Compiling the code with `-O3`

```
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<216; i++)
    {
        a[i+2] = b[i] + 5;
        b[i+3] = a[i] + 10;
    }
    return 0;
}
```

- Additional options for parallelization
  `-ftree-parallelize-loops=2 -fdump-tree-parloops-all`
- Additional options for vectorization
  `-fdump-tree-vect-all -msse4`

Example 4: No Vectorization and No Parallelization

- Step 1: Observing the final decision about vectorization

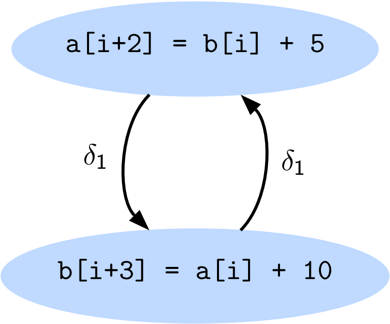
```
noparvec.c:5: note: vectorized 0 loops in function.
```

- Step 2: Observing the final decision about parallelization

```
FAILED: data dependencies exist across iterations
```

Example 4: No Vectorization and No Parallelization

Step 3: Understanding the dependencies that prohibit vectorization and parallelization



Advanced Issues in Vectorization

Alignment by Peeling

```
int a[256];
int main ()
{
   int i;
   for (i=4; i<253; i++)
      a[i-3] = a[i-3] + a[i+2];
}
```

Advanced Issues in Vectorization

Alignment by Peeling

```
int a[256];
int main ()
{
   int i;
   for (i=4; i<253; i++)
      a[i-3] = a[i-3] + a[i+2];
}
```

a[1] = a[1] + a[6]

Advanced Issues in Vectorization

Alignment by Peeling

```
int a[256];
int main ()
{
   int i;
   for (i=4; i<253; i++)
      a[i-3] = a[i-3] + a[i+2];
}
```

`a[1] = a[1] + a[6]`

Peel Factor = 3

Advanced Issues in Vectorization

Alignment by Peeling

```
int a[256];
int main ()
{
   int i;
   for (i=4; i<253; i++)
      a[i-3] = a[i-3] + a[i+2];
}
```

`a[1] = a[1] + a[6]`

Peel Factor $= 3$

Advanced Issues in Vectorization

Alignment by Peeling

```
int a[256];
int main ()
{
   int i;
   for (i=4; i<253; i++)
      a[i-3] = a[i-3] + a[i+2];
}
```

`a[1] = a[1] + a[6]`

Peel Factor $= 2$

Advanced Issues in Vectorization

Alignment by Peeling

```
int a[256];
int main ()
{
   int i;
   for (i=4; i<253; i++)
      a[i-3] = a[i-3] + a[i+2];
}
```

```
a[1] = a[1] + a[6]
```

Maximize alignment with minimal peel factor

Advanced Issues in Vectorization

Alignment by Peeling

```
int a[256];
int main ()
{
   int i;
   for (i=4; i<253; i++)
      a[i-3] = a[i-3] + a[i+2];
}
```

Peel the loop by 3

Advanced Issues in Vectorization

An aligned vectorized code can consist of three parts

- Peeled Prologue - Scalar code for alignment
- Vectorized body - Iterations that are vectorized
- Epilogue - Residual scalar iterations

Advanced Issues in Vectorization

Loop Versioning

How do we vectorize a loop that has

- unaligned data references
- undetermined data dependence relation

```
int a[256];
int main ()
{
   int i;
   for (i=0; i<100; i++)
      a[i] = a[i*2];
}
```

Advanced Issues in Vectorization

Loop Versioning

How do we vectorize a loop that has

- unaligned data references
- undetermined data dependence relation

```
int a[256];
int main ()
{
   int i;
   for (i=0; i<100; i++)
      a[i] = a[i*2];
}
```

```
"Bad distance vector for a[i] and a[i*2]"
```

Advanced Issues in Vectorization

- Generate two versions of the loop, one which is vectorized and one which is not.
- A test is then generated to control the execution of desired version. The test checks for the alignment of all of the data references that may or may not be aligned.
- An additional sequence of runtime tests is generated for each pairs of data dependence relations whose independence was undetermined or unproven.
- The vectorized version of loop is executed only if both alias and alignment tests are passed.

When to Vectorize?

Vectorization is profitable when

$$SIC * niters + SOC > VIC * \left(\frac{niters - PL\_ITERS - EP\_ITERS}{VF}\right) + VOC$$

`SIC` = scalar iteration cost
`VIC` = vector iteration cost
`VOC` = vector outside cost
`VF` = vectorization factor
`PL_ITERS` = prologue iterations
`EP_ITERS` = epilogue iterations
`SOC` = scalar outside cost

Part 2

Loop Transformations in Polytope Model

Problems with Classical Loop Nest Transforms

Loop nest optimization is a combinatorial problem. Due to the growing complexity of modern architectures, it involves two increasingly difficult tasks:

- Analyzing the profitability of sequences of transformations to enhance parallelism, locality, and resource usage
- the construction and exploration of search space of legal transformation sequences

Problems with Classical Loop Nest Transforms

Loop nest optimization is a combinatorial problem. Due to the growing complexity of modern architectures, it involves two increasingly difficult tasks:

- Analyzing the profitability of sequences of transformations to enhance parallelism, locality, and resource usage
- the construction and exploration of search space of legal transformation sequences

Practical optimizing and parallelizing compilers restore to a predefined set of enabling

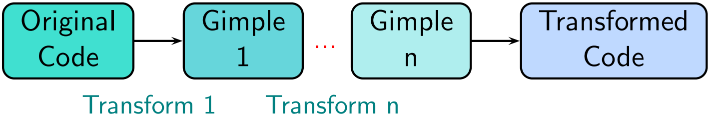
Problems with Classical Loop Nest Transforms

Loop transformations on Lambda Framework were discontinued in gcc-4.6.0 for the following reasons:

- Difficult to undo loop transformations - transforms are applied on the syntactic form
- Difficult to compose transformations - intermediate translation to a syntactic form is necessary after each transformation
- Ordering of transformations is fixed

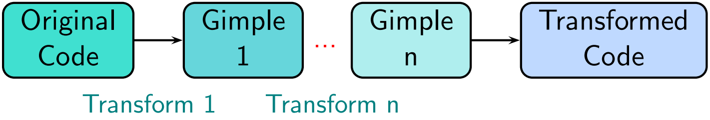
Problems with Classical Loop Nest Transforms

Traditional Loop Transforms:

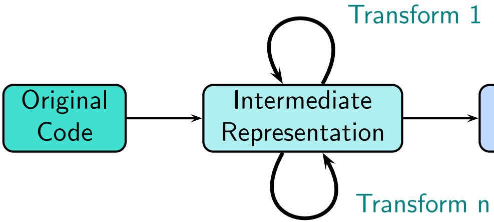


Problems with Classical Loop Nest Transforms

Traditional Loop Transforms:



Expected Loop Transforms with Composition:



Requirement

GCC requires a rich algebraic representation that

- Provides a solution to *phase-ordering* problem - facilitate efficient exploration and configuration of multiple transformation sequences
- Decouples the transformations from the syntatic form of program, avoiding code size explosion
- Performs only legal transformation sequences
- Provides precise performance models and profitability prediction heuristics

Solution : Polyhedral Representation

- Polytope Model is a mathematical framework for loop nest optimizations
- The loop bounds parametrized as inequalities form a convex polyhedron
- An affine scheduling function specifies the scanning order of integral points

Solution : Polyhedral Representation

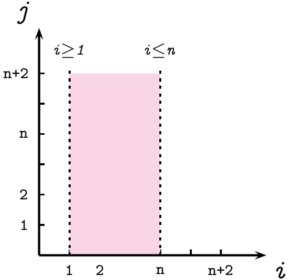
- Polytope Model is a mathematical framework for loop nest optimizations
- The loop bounds parametrized as inequalities form a convex polyhedron
- An affine scheduling function specifies the scanning order of integral points

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
       if (i<=n-j+2)
          S1;
```

Solution : Polyhedral Representation

- Polytope Model is a mathematical framework for loop nest optimizations
- The loop bounds parametrized as inequalities form a convex polyhedron
- An affine scheduling function specifies the scanning order of integral points

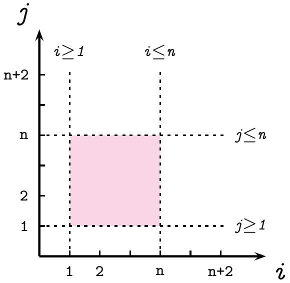
```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
       if (i<=n-j+2)
          S1;
```



Solution : Polyhedral Representation

- Polytope Model is a mathematical framework for loop nest optimizations
- The loop bounds parametrized as inequalities form a convex polyhedron
- An affine scheduling function specifies the scanning order of integral points

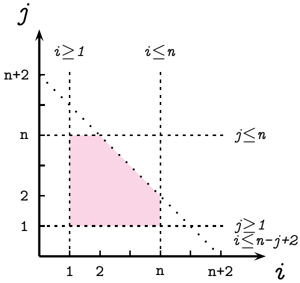
```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
       if (i<=n-j+2)
          S1;
```



Solution : Polyhedral Representation

- Polytope Model is a mathematical framework for loop nest optimizations
- The loop bounds parametrized as inequalities form a convex polyhedron
- An affine scheduling function specifies the scanning order of integral points

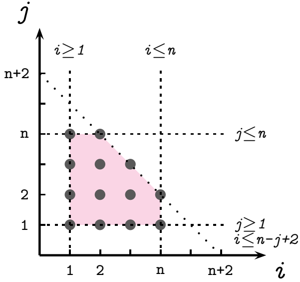
```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
       if (i<=n-j+2)
          S1;
```



Solution : Polyhedral Representation

- Polytope Model is a mathematical framework for loop nest optimizations
- The loop bounds parametrized as inequalities form a convex polyhedron
- An affine scheduling function specifies the scanning order of integral points

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        if (i<=n-j+2)
            S1;
```



GRAPHITE

GRAPHITE is the interface for polyhedra representation of GIMPLE

goal: more high level loop optimizations

GRAPHITE

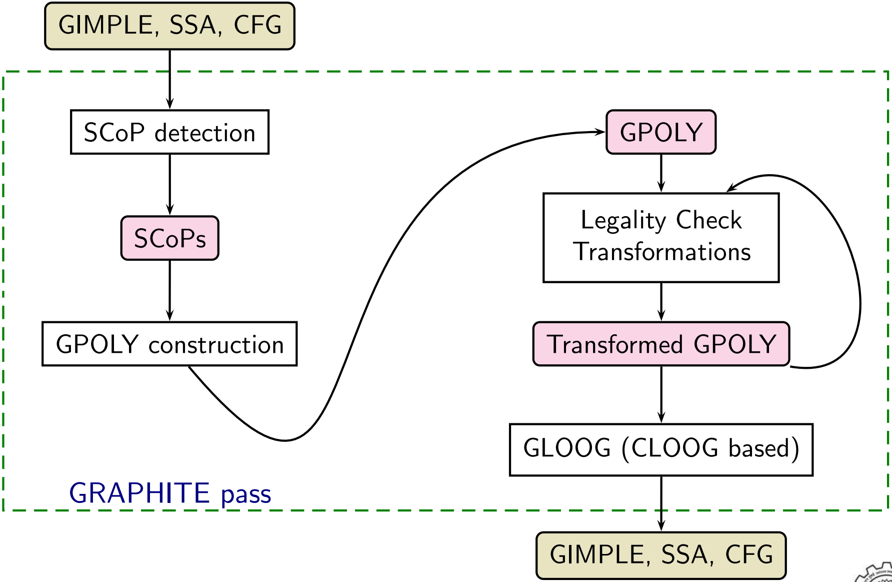
GRAPHITE is the interface for polyhedra representation of GIMPLE

goal: more high level loop optimizations

Tasks of GRAPHITE Pass:

- Extract the *polyhedral model* representation out of GIMPLE
- Perform the various optimizations and analyses on this polyhedral model representation
- Regenerate the GIMPLE three-address code that corresponds to transformations on the polyhedral model

Compilation Workflow



What Code Can be Represented?

The target of polyhedral representation are sequence of loop nests with

- Affine loop bounds (e.g. i $<$ 4$*$n$+$4$*$j-1)
- Affine array accesses (e.g. A[3i$+$1])
- Constant loop strides (e.g. i $+=$ 2)
- Conditions containing comparisons ($<$,$\leq$,$>$,$\geq$,$==$,!$=$) between affine functions
- Invariant global parameters

What Code Can be Represented?

The target of polyhedral representation are sequence of loop nests with

- Affine loop bounds (e.g. i $<$ 4$*$n$+$4$*$j-1)
- Affine array accesses (e.g. A[3i$+$1])
- Constant loop strides (e.g. i $+=$ 2)
- Conditions containing comparisons ($<$,$\leq$,$>$,$\geq$,$==$,!$=$) between affine functions
- Invariant global parameters

Non-rectangular, non-perfectly nested loops are also represented polyhedrally for optimization

GPOLY

GPOLY : the polytope representation in GRAPHITE, currently implemented by the Parma Polyhedra Library (PPL)

- SCoP - The optimization unit (e.g. a loop with some basic blocks)
  scop := ([*black box*])
- Black Box - An operation (e.g. basic block with one or more statements) where the memory accesses are known
  black box := (*iteration domain*, *scattering matrix*, [*data reference*])
- Iteration Domain - The set of loop iterations for the black box
- Data Reference - The memory cells accessed by the black box
- Scattering Matrix - Defines the execution order of statement iterations (e.g. schedule)

Building SCoPs

- SCoPs built on top of the CFG
- Basic blocks with side-effect statements are split
- All basic blocks belonging to a SCoP are dominated by entry, and postdominated by exit of the SCoP

Building SCoPs

- SCoPs built on top of the CFG
- Basic blocks with side-effect statements are split
- All basic blocks belonging to a SCoP are dominated by entry, and postdominated by exit of the SCoP

```
int a[256][256], b[245], c[145], n;
int main ()
{
    int i, j;
    for (i=0; i<n; i++) {
        for (j=0; j<62; j++) {
            a[i][j] = a[i+1][j+2];
            a[j][i+7] = b[j];
        }
        c[i] = a[i][i+14];
    }
}
```

Building SCoPs

- SCoPs built on top of the CFG
- Basic blocks with side-effect statements are split
- All basic blocks belonging to a SCoP are dominated by entry, and postdominated by exit of the SCoP

```
int a[256][256], b[245], c[145], n;
int main ()
{
    int i, j;
    for (i=0; i<n; i++) {
        for (j=0; j<62; j++) {
            a[i][j] = a[i+1][j+2];
            a[j][i+7] = b[j];
        }
        c[i] = a[i][i+14];
    }
}
```

global parameter

Building SCoPs

- SCoPs built on top of the CFG
- Basic blocks with side-effect statements are split
- All basic blocks belonging to a SCoP are dominated by entry, and postdominated by exit of the SCoP

```
int a[256][256], b[245], c[145], n;
int main ()
{
    int i, j;
    for (i=0; i<n; i++) {
        for (j=0; j<62; j++) {
            a[i][j] = a[i+1][j+2];
            a[j][i+7] = b[j];
        }
        c[i] = a[i][i+14];
    }
}
```

global parameter

SCoP

Building SCoPs

- SCoPs built on top of the CFG
- Basic blocks with side-effect statements are split
- All basic blocks belonging to a SCoP are dominated by entry, and postdominated by exit of the SCoP

```
int a[256][256], b[245], c[145], n;
int main ()
{
    int i, j;
    for (i=0; i<n; i++) {
        for (j=0; j<62; j++) {
            a[i][j] = a[i+1][j+2];
            a[j][i+7] = b[j];
        }
        c[i] = a[i][i+14];
    }
}
```

global parameter

SCoP

bbs 1

bbs 2

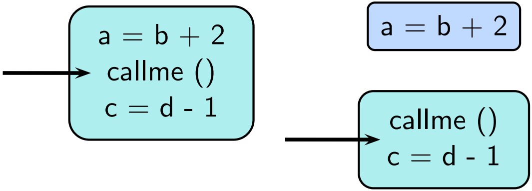
Example : Building SCoPs

Splitting basic blocks:

```
a = b + 2
callme ()
c = d - 1
```

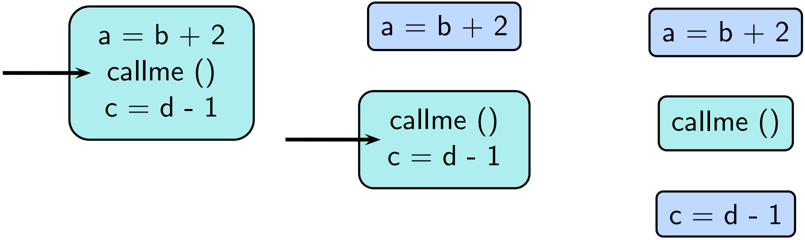
Example : Building SCoPs

Splitting basic blocks:



Example : Building SCoPs

Splitting basic blocks:



Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{\texttt{i} \mid \mathcal{D}^S \times (\texttt{i},\texttt{g},\texttt{1})^T \geq 0\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
        A[2*i][j+1] = ...;
```

$$\begin{bmatrix} \texttt{i} & \texttt{j} & \texttt{m} & \texttt{n} & \texttt{cst} \\ \hline & & & & \\ & & & & \end{bmatrix} \geq 0$$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{\mathtt{i} \mid \mathcal{D}^S \times (\mathtt{i},\mathtt{g},1)^T \geq 0\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
       A[2*i][j+1] = ...;
```

$$\begin{array}{ccccc} \mathtt{i} & \mathtt{j} & \mathtt{m} & \mathtt{n} & \mathtt{cst} \end{array}$$
$$\left[\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ & & & & \\ & & & & \end{array}\right] \geq 0$$

$\mathtt{i} \geq 0$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{\texttt{i} \mid \mathcal{D}^S \times (\texttt{i,g,1})^T \geq 0\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
       A[2*i][j+1] = ...;
```

$$\begin{array}{ccccc} i & j & m & n & cst \end{array}$$
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ & & & & \\ & & & & \end{bmatrix} \geq 0$$

$\texttt{i} \leq \texttt{m - 1}$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{\mathtt{i} \mid \mathcal{D}^S \times (\mathtt{i},\mathtt{g},1)^T \geq 0\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
        A[2*i][j+1] = ...;
```

$$\begin{array}{c} \begin{array}{ccccc} \mathtt{i} & \mathtt{j} & \mathtt{m} & \mathtt{n} & \mathtt{cst} \end{array} \\ \left[\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & -5 \end{array}\right] \geq 0 \end{array}$$

$\mathtt{j} \geq 5$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{\mathtt{i} \mid \mathcal{D}^S \times (\mathtt{i,g,1})^T \geq 0\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
        A[2*i][j+1] = ...;
```

$$\begin{array}{ccccc} \mathtt{i} & \mathtt{j} & \mathtt{m} & \mathtt{n} & \mathtt{cst} \end{array}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & -5 \\ 0 & -1 & 0 & 1 & -1 \end{bmatrix} \geq 0$$

$\mathtt{j} \leq \mathtt{n - 1}$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)

$$\mathcal{F} = \{(\mathtt{i,a,s}) \mid \mathcal{F} \times (\mathtt{i,a,s,g,1})^T \geq 0\}$$

```
for (i=1; i<m; i++)
   for (j=5; j<n; j++)
      A[2*i][j+1] = ...;
```

$$\left[\begin{array}{ccccc} \mathtt{i} & \mathtt{j} & \mathtt{m} & \mathtt{n} & \mathtt{cst} \\ \hline & & & & \\ \end{array}\right]$$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)

$$\mathcal{F} = \{(\mathtt{i,a,s}) \mid \mathcal{F} \times (\mathtt{i,a,s,g,1})^T \geq 0\}$$

```
for (i=1; i<m; i++)
   for (j=5; j<n; j++)
      A[2*i][j+1] = ...;
```

$$\begin{array}{ccccc} \mathtt{i} & \mathtt{j} & \mathtt{m} & \mathtt{n} & \mathtt{cst} \end{array}$$
$$\left[\begin{array}{ccccc} 2 & 0 & 0 & 0 & 0 \\ & & & & \end{array}\right]$$

$2 * \mathtt{i}$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)

$$\mathcal{F} = \{(\mathtt{i,a,s}) \mid \mathcal{F} \times (\mathtt{i,a,s,g,1})^T \geq 0\}$$

```
for (i=1; i<m; i++)
   for (j=5; j<n; j++)
      A[2*i][j+1] = ...;
```

$$\begin{bmatrix} \mathtt{i} & \mathtt{j} & \mathtt{m} & \mathtt{n} & \mathtt{cst} \\ \hline 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

j + 1

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)
- Scattering Function (scheduling order)

$$\theta = \{(\texttt{t,i}) \mid \theta \times (\texttt{t,i,g,1})^T \geq \texttt{0}\}$$

sequence [s_1, s_2]:
$\mathcal{S}[s_1] = \mathrm{t}$, $\mathcal{S}[s_2] = \mathrm{t} + 1$

loop [$loop_1\ s\ end_1$] : i_1 indexes loop_1 iterations
$\mathcal{S}[loop_1] = \mathrm{t}$, $\mathcal{S}[s] = (\mathrm{t}, \mathrm{i}_1, 0)$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)
- Scattering Function (scheduling order)

$$\theta = \{(\mathtt{t,i}) \mid \theta \times (\mathtt{t,i,g,1})^T \geq 0\}$$

```
for (i=1;i<=N;i++) {
   for (j=1;j<=i-1;j++) {
      a[i][i] -= a[i][j];
      a[j][i] += a[i][j];
   }
   a[i][i] = sqrt(a[i][i]);
}
```

Scattering Function

$$\theta_{S1}(i,j)^T = (0,i,0,j,0)^T$$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)
- Scattering Function (scheduling order)

$$\theta = \{(\mathtt{t,i}) \mid \theta \times (\mathtt{t,i,g,1})^T \geq \mathtt{0}\}$$

```
for (i=1;i<=N;i++) {
   for (j=1;j<=i-1;j++) {
      a[i][i] -= a[i][j];
      a[j][i] += a[i][j];
   }
   a[i][i] = sqrt(a[i][i]);
}
```

Scattering Function

$$\theta_{S2}(i,j)^T = (0,i,0,j,1)^T$$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)
- Scattering Function (scheduling order)

$$\theta = \{(\mathtt{t,i}) \mid \theta \times (\mathtt{t,i,g,1})^T \geq \mathtt{0}\}$$

```
for (i=1;i<=N;i++) {
   for (j=1;j<=i-1;j++) {
      a[i][i] -= a[i][j];
      a[j][i] += a[i][j];
   }
   a[i][i] = sqrt(a[i][i]);
}
```

Scattering Function

$\theta_{S3}(i,j)^T = (0,i,1)^T$

Polyhedral Dependence Analysis in GRAPHITE

- An *instancewise dependence analysis* - dependences between source and sink represented as polyhedra
- Scalar dependences are treated as zero-dimensional arrays
- Global parameters are handled
- Can take care of conditional and some form of triangular loops, as the information can be safely integrated with the iteration domain
- High cost, and therefore dependence is computed only to validate a transformation

Legality of Transformations

Original Code

```
int A[256][256];
int main ()
{
   for (j=0; j<n; j++){
       for (i=0; i<n; i++){
            A[i][j] = A[j][i];
       }
   }
}
```

$pdr_0 =$ `A[j][i]`
$pdr_1 =$ `A[i][j]`

Memory location `A[0][1]` is read at pdr_0 when $j = 0$ and later written at pdr_1 when $j = 1$

Dependence : Write after Read

Legality of Transformations

Original Code

```
int A[256][256];
int main ()
{
    for (j=0; j<n; j++){
        for (i=0; i<n; i++){
            A[i][j] = A[j][i];
        }
    }
}
```

Loop Interchange

```
int A[256][256];
int main ()
{
    for (i=0; j<n; i++){
        for (j=0; j<n; j++){
            A[i][j] = A[j][i];
        }
    }
}
```

Are the dependences preserved after the transformation?

Legality of Transformations

Original Code

```
int A[256][256];
int main ()
{
   for (j=0; j<n; j++){
      for (i=0; i<n; i++){
          A[i][j] = A[j][i];
      }
   }
}
```

Loop Interchange

```
int A[256][256];
int main ()
{
   for (i=0; j<n; i++){
      for (j=0; j<n; j++){
          A[i][j] = A[j][i];
      }
   }
}
```

Are the dependences preserved after the transformation?

No! `A[0][1]` is first written at pdr_1 when i $=0$, and then read at pdr_0 when i $=1$

Dependence : Read after Write

Legality of Transformations

- A transformation is legal if the dependences are preserved - for any dependence instance, the source and sink remain same across transformation
- If the dependence is reversed, source becomes sink and sink becomes source in the transformed space
- GRAPHITE captures this notion in *Violated Dependence Analysis*. A reverse data dependence polyhedron is constructed in the transformed scattering from sink to source, and it is intersected with the original polyhedron
- If the intersection is non-empty, atleast one pair of iterations is executed in wrong order, rendering the transformation illegal

Parallelization with GRAPHITE

- The GRAPHITE pass without optimizations is run (`GIMPLE` $\rightarrow$ `POLY` $\rightarrow$ `GIMPLE`)
- During this conversion, data dependence is performed using *instancewise data dependence analysis*
- This dependence result is used to determine if the loop can be parallelized

Parallelization with GRAPHITE

- The GRAPHITE pass without optimizations is run (`GIMPLE` $\rightarrow$ `POLY` $\rightarrow$ `GIMPLE`)
- During this conversion, data dependence is performed using *instancewise data dependence analysis*
- This dependence result is used to determine if the loop can be parallelized

Benefits:

- Stronger dependence analysis, can detect parallelism in loops with invariant parameters
- Conditional loops and some triangular loops can be parallelized after loop distribution

Parallelization with GRAPHITE

- The GRAPHITE pass without optimizations is run (`GIMPLE` $\rightarrow$ `POLY` $\rightarrow$ `GIMPLE`)
- During this conversion, data dependence is performed using *instancewise data dependence analysis*
- This dependence result is used to determine if the loop can be parallelized

Benefits:

- Stronger dependence analysis, can detect parallelism in loops with invariant parameters
- Conditional loops and some triangular loops can be parallelized after loop distribution

`Extra Compilation flag : -floop-parallelize-all`

Loop Tranformations in GRAPHITE

Loop transforms implemented in GRAPHITE:

- loop interchange
- loop blocking and loop stripmining
- loop flattening

These transformations are mostly used to improve scope of parallelization or vectorization. Application of such transformations must not violate the dependences

Loop Tranformations in GRAPHITE

Loop transforms implemented in GRAPHITE:

- loop interchange
- loop blocking and loop stripmining
- loop flattening

These transformations are mostly used to improve scope of parallelization or vectorization. Application of such transformations must not violate the dependences

Cost Model:

- Cost models are used to check the profitability of transformation.
- For example, loops are interchanged only if the sum total of inner loop's strides are greater than the outer loop

Loop Interchange in GRAPHITE

Original Code

```
int A[256][256];
int main ()
{
   for (j=0; j<n; j++){
      for (i=1; i<n; i++){
           A[i][j] = A[i-1][j];
      }
   }
}
```

Strides of i $= 255 + 255 = 510$

Strides of j $= 1 + 1 = 2$

Since strides of i $>$ strides of j, interchange loop i with j

Loop Interchange in GRAPHITE

Original Code

```
int A[256][256];
int main ()
{
    for (j=0; j<n; j++){
        for (i=1; i<n; i++){
            A[i][j] = A[i-1][j];
        }
    }
}
```

After Interchange

```
int A[256][256];
int main ()
{
    for (i=1; i<n; i++){
        for (j=0; j<n; j++){
            A[i][j] = A[i-1][j];
        }
    }
}
```

outermost loop has the largest stride

Loop Interchange in GRAPHITE

Original Code

```
for (i=1; i<n; i++){
   for (j=0; j<n; j++){
       A[i][j] = A[i-1][j]
   }
}
```

Outer Loop - dependence on i, can not be parallelized

Inner Loop - parallelizable, but synchronization barrier required

Total number of times synchronization executed = n

Loop Interchange in GRAPHITE

Original Code

```
for (i=1; i<n; i++){
   for (j=0; j<n; j++){
       A[i][j] = A[i-1][j]
   }
}
```

After Interchange

```
for (j=0; i<n; i++){
   for (i=1; j<n; j++){
       A[i][j] = A[i-1][j]
   }
}
```

Outer Loop - parallelizable

Total number of times synchronization executed $= 1$

Loop Interchange in GRAPHITE

Original Code

```
for (i=1; i<n; i++){
   for (j=0; j<n; j++){
       A[i][j] = A[i-1][j]
   }
}
```

After Interchange

```
for (j=0; i<n; i++){
   for (i=1; j<n; j++){
       A[i][j] = A[i-1][j]
   }
}
```

Outer Loop - parallelizable

Total number of times synchronization executed = 1

Is this loop interchange profitable in GRAPHITE?

Loop Regeneration

- *Chunky Loop Generator* (CLooG) is used to regenerate the loop
- It scans the integral points of the polyhedra to recreate loop bounds

Loop Regeneration

- *Chunky Loop Generator* (CLooG) is used to regenerate the loop
- It scans the integral points of the polyhedra to recreate loop bounds

Original Program

```
for (i=0; i<250; i++)
   for (j=0; j<200; j++) {
      if (j < k+3)
         S1;
   }
```

Loop Regeneration

- *Chunky Loop Generator* (CLooG) is used to regenerate the loop
- It scans the integral points of the polyhedra to recreate loop bounds

Original Program

```
for (i=0; i<250; i++)
    for (j=0; j<200; j++) {
        if (j < k+3)
            S1;
    }
```

Loop generated by CLooG

```
for (i=0; i<=249; i++) {
    for (j=0; j<=min(k+2,199); j++) {
        S1;
    }
}
```

Loop Regeneration

- *Chunky Loop Generator* (CLooG) is used to regenerate the loop
- It scans the integral points of the polyhedra to recreate loop bounds

Original Program

```
for (i=0; i<250; i++)
    for (j=0; j<200; j++) {
        if (j < k+3)
            S1;
    }
```

Loop generated by CLooG

```
for (i=0; i<=249; i++) {
    for (j=0; j<=min(k+2,199); j++) {
        S1;
    }
}
```

Merge conditional code with loop bounds if possible

GRAPHITE Conclusions

Advantages of GRAPHITE

- Better data dependence analysis - handles conditional codes, parametric invariants
- Makes auto-parallelization more efficient
- Composition of transforms is possible

GRAPHITE Conclusions

Advantages of GRAPHITE

- Better data dependence analysis - handles conditional codes, parametric invariants
- Makes auto-parallelization more efficient
- Composition of transforms is possible

Future Scope

- Making instancewise dependence analysis algorithmically cheaper
- Automating the search most profitable transform composition sequence
- Developing efficient cost models
- Exploring scalability issues

Parallelization and Vectorization in GCC : Conclusions

- Chain of recurrences seems to be a useful generalization
- Interaction between different passes is not clear due to fixed order
- Auto-vectorization and auto-parallelization can be improved by enhancing the dependence analysis framework
- Efficient cost models are needed to automate legal transformation composition
- GRAPHITE seems to be a promising mathematical abstraction

Last but not the least ...

Thank You!