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 2012
Outline

- Transformation for parallel and vector execution
- Data dependence
- Auto-parallelization and auto-vectorization in Lambda Framework
- Conclusion
The Scope of This Tutorial

• What this tutorial does not address
  ▶ Details of algorithms, code and data structures used for parallelization and vectorization
  ▶ Machine level issues related to parallelization and vectorization

• What this tutorial addresses
  ▶ GCC’s approach of discovering and exploiting parallelism
  ▶ Illustrated using carefully chosen examples
Part 1

Transformations for Parallel and Vector Execution
Vectorization: SISD $\Rightarrow$ SIMD

- Parallelism in executing operation on shorter operands (8-bit, 16-bit, 32-bit operands)
- Existing 32 or 64-bit arithmetic units used to perform multiple operations in parallel

A 64 bit word $\equiv$ a vector of $2 \times (32 \text{ bits})$, $4 \times (16 \text{ bits})$, or $8 \times (8 \text{ bits})$
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

- A[0..N]
- B[0..N]
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

A[0..N]  B[0..N]  . . .
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

A[0..N]  B[0..N]  Iteration #  1
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

Iteration # 1 2
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

```
A[0..N]  B[0..N]
```

Iteration # 1 2 3
**Example 1**

Vectorization  \( (\text{SISD} \Rightarrow \text{SIMD}) \) : Yes
Parallelization  \( (\text{SISD} \Rightarrow \text{MIMD}) \) : Yes

**Original Code**

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

---

**Iteration #**  1  2  3  4
Example 1

Vectorization (SISD \Rightarrow \text{SIMD}) : Yes  
Parallelization (SISD \Rightarrow \text{MIMD}) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

A[0..N]  B[0..N]
Iteration #  1  2  3  4  5  6  ...
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location
Example 1

**Vectorization** (SISD $\Rightarrow$ SIMD) : Yes

Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

A[0..N]  B[0..N]

Iteration #  1  2  3  4  5  6  7  8  9  10
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location
Example 1

Vectorization \((\text{SISD} \Rightarrow \text{SIMD})\) : Yes
Parallelization \((\text{SISD} \Rightarrow \text{MIMD})\) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location...
### Example 1

**Vectorization (SISD ⇒ SIMD) : Yes**  
**Parallelization (SISD ⇒ MIMD) : Yes**

#### Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

#### Vectorized Code

```c
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
```

![Diagram](image-url)
Example 1

Vectorization  (SISD $\Rightarrow$ SIMD) : Yes
Parallelization  (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i+=4)
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Vectorized Code

```c
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
```

Vectorization Factor

A[0..N]  B[0..N]  Iteration #

- Iteration 1
- Iteration 2
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Vectorized Code

```c
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
```

A[0..N] B[0..N] Iteration #
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location...
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location.
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Parallelized Code

```c
int A[N], B[N], i;
for-all (i=1 to N)
    A[i] = A[i] + B[i-1];
```

A[0..N] → B[0..N] → Iteration #: 1

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Example 1: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

```
A[0..N]  B[0..N]  ...
```

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

| Nature of accesses in our example |
|------------------|------------------|------------------|
| Iteration $i$    | Iteration $i + k$| Observation |
| Read             | Write            | Read           |
| Write            | Read             | Write           |
| Write            | Write            | Read           |
| Read             | Read             | Read           |

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
<td>Iteration $i + k$</td>
<td>Observation</td>
</tr>
<tr>
<td>Read</td>
<td>Write</td>
<td>No</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>No</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td></td>
</tr>
</tbody>
</table>

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

A[0..N] . . .
B[0..N] . . .
Example 1: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes

Parallelization (SISD $\Rightarrow$ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

A[0..N]  B[0..N]

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 1: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved.
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N] . . .
B[0..N] . . .
```
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N]  
B[0..N]  
Iteration #
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

Iteration # 1
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N]  B[0..N]
```

Iteration # 1 2
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location...

A[0..N]...

B[0..N]...

Iteration # 1 2 3...
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location.

```
A[0..N]  B[0..N]
0  1  2  3  4  ...
```

Iteration #
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N] B[0..N]...

Iteration # 1 2 3 4 5...
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N] B[0..N]

Iteration # 1 2 3 4 5 6
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

Iteration #  1  2  3  4  5  6  7  8
**Example 2**

Vectorization (SISD $\Rightarrow$ SIMD) : Yes

Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location...

A[0..N]...

B[0..N]...

Iteration # 1 2 3 4 5 6 7 8 9
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N]  . . .  B[0..N]  . . .

Iteration #  1  2  3  4  5  6  7  8  9  10
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N]  B[0..N]  Iteration #  1  2  3  4  5  6  7  8  9  10  11
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N]  B[0..N]

Iteration #  1  2  3  4  5  6  7  8  9  10  11  12  ...
Example 2

Vectorization \((\text{SISD} \Rightarrow \text{SIMD})\) : Yes
Parallelization \((\text{SISD} \Rightarrow \text{MIMD})\) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction
Example 2

Vectorization \((\text{SISD} \Rightarrow \text{SIMD})\) : Yes
Parallelization \((\text{SISD} \Rightarrow \text{MIMD})\) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction.
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction
- Read-writes across multiple instructions executing in parallel may not be synchronized
Example 2: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location
Example 2: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observation</td>
</tr>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

int A[N], B[N], i;
for (i = 0; i < N; i++)
Example 2: The Moral of the Story

Vectorization \((\text{SISD} \Rightarrow \text{SIMD})\) : Yes
Parallelization \((\text{SISD} \Rightarrow \text{MIMD})\) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration (i)</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>
Example 2: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes

Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 2: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
<td>Iteration $i + k$</td>
</tr>
<tr>
<td>Read</td>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
</tr>
</tbody>
</table>

Original Code
```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

...
Example 2: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

int A[N], B[N], i;
for (i=0; i<N; i++)

...
Example 3

Vectorization  (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

```
A[0..N]    ... 
B[0..N]    ...
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

```
A[0..N]   B[0..N]
```

Iteration #
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

```
A[0..N]   B[0..N]
```

Iteration # 1
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

Observe reads and writes into a given location
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];

Observe reads and writes into a given location

A[0..N]

B[0..N]

Iteration #  1  2  3  4  5
Example 3

Vectorization  (SISD ⇒ SIMD) : No
Parallelization  (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

```
A[0..N]  . . .
B[0..N]  . . .
```

Iteration #: 1 2 3 4 5 6
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

A[0..N]  B[0..N]

Iteration # 1 2 3 4 5 6 7
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

```
A[0..N]  B[0..N]  Iteration #
```

```
1  2  3  4  5  6  7  8  9  ...
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

Observe reads and writes into a given location
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

![Diagram showing array accesses and iterations]
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

Observe reads and writes into a given location.
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

```
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

```
A[0..N]
```
```
B[0..N]
```
```
Iteration #
```
```
1 2 3 4 5 6 7 8 9 10 11 12 ...
```
**Example 3**

Vectorization (SISD $\Rightarrow$ SIMD) : No

Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

### Nature of accesses in our example

<table>
<thead>
<tr>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td>No</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>Yes</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td>No</td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td></td>
</tr>
</tbody>
</table>

### Access Diagram

- **A[0..N]**
- **B[0..N]**
- **Iteration #'s**: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
**Example 3**

Vectorization (SISD $\Rightarrow$ SIMD) : No

Parallelization (SISD $\Rightarrow$ MIMD) : No

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

int A[N], B[N];

for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 4

Vectorization  (SISD $\Rightarrow$ SIMD)  : No
Parallelization  (SISD $\Rightarrow$ MIMD)  : Yes
Example 4

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : Yes

- This case is not possible
Example 4

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : Yes

- This case is not possible
- Vectorization is a limited granularity parallelization
Example 4

Vectorization \ (SISD \Rightarrow \ SIMD) : No
Parallelization \ (SISD \Rightarrow \ MIMD) : Yes

• This case is not possible
• Vectorization is a limited granularity parallelization
• If parallelization is possible then vectorization is trivially possible
Data Dependence

Let statements $S_i$ and $S_j$ access memory location $m$ at time instants $t$ and $t + k$

<table>
<thead>
<tr>
<th>Access in $S_i$</th>
<th>Access in $S_j$</th>
<th>Dependence</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read $m$</td>
<td>Write $m$</td>
<td>Anti (or Pseudo)</td>
<td>$S_i \bar{\delta} S_j$</td>
</tr>
<tr>
<td>Write $m$</td>
<td>Read $m$</td>
<td>Flow (or True)</td>
<td>$S_i \delta S_j$</td>
</tr>
<tr>
<td>Write $m$</td>
<td>Write $m$</td>
<td>Output (or Pseudo)</td>
<td>$S_i \delta^o S_j$</td>
</tr>
<tr>
<td>Read $m$</td>
<td>Read $m$</td>
<td>Does not matter</td>
<td></td>
</tr>
</tbody>
</table>

- Pseudo dependences may be eliminated by some transformations
- True dependence cannot be eliminated
Data Dependence

Consider dependence between statements $S_i$ and $S_j$ in a loop

- **Loop independent dependence.** $t$ and $t + k$ occur in the same iteration of a loop
  - $S_i$ and $S_j$ must be executed sequentially
  - Different iterations of the loop can be parallelized

- **Loop carried dependence.** $t$ and $t + k$ occur in the different iterations of a loop
  - Within an iteration, $S_i$ and $S_j$ can be executed in parallel
  - Different iterations of the loop must be executed sequentially

- $S_i$ and $S_j$ may have both loop carried and loop independent dependences
Dependence in Example 1

- **Program**

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1]; /* S1 */
```

- **Dependence graph**

```
S1

\( \delta_\infty \)
```

- **No loop carried dependence**
  Both vectorization and parallelization are possible
Dependence in Example 1

- Program

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1]; /* S1 */
```

- Dependence graph

- No loop carried dependence
  Both vectorization and parallelization are possible
Dependence in Example 2

- Program

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i] = A[i+1] + B[i]; /* S1 */
```

- Dependence graph

- Loop carried anti-dependence
  Parallelization is not possible
  Vectorization is possible since all reads are done before all writes
Dependence in Example 2

- **Program**

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i] = A[i+1] + B[i]; /* S1 */
```

- **Dependence graph**

  ![Dependence Graph]

  Dependence due to the outermost loop

- **Loop carried anti-dependence**
  Parallelization is not possible
  Vectorization is possible since all reads are done before all writes
Dependence in Example 3

- **Program**

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1]; /* S1 */
```

- **Dependence graph**

- **Loop carried flow-dependence**
  Neither parallelization nor vectorization is possible
### Iteration Vectors and Index Vectors: Example 1

```c
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }
```

<table>
<thead>
<tr>
<th>Index Vector</th>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>0,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>0,1</td>
</tr>
<tr>
<td>0,2</td>
<td>1,2</td>
<td>0,2</td>
</tr>
<tr>
<td>0,3</td>
<td>1,3</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>2,0</td>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
<td>2,1</td>
<td>1,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
<td>1,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>3,0</td>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
<td>3,1</td>
<td>2,1</td>
</tr>
<tr>
<td>2,2</td>
<td>3,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>3,3</td>
<td>2,3</td>
</tr>
<tr>
<td>3,0</td>
<td>4,0</td>
<td>3,0</td>
</tr>
<tr>
<td>3,1</td>
<td>4,1</td>
<td>3,1</td>
</tr>
<tr>
<td>3,2</td>
<td>4,2</td>
<td>3,2</td>
</tr>
<tr>
<td>3,3</td>
<td>4,3</td>
<td>3,3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 1

```c
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHS</td>
<td>RHS</td>
</tr>
<tr>
<td>0, 0</td>
<td>1, 0</td>
</tr>
<tr>
<td>0, 1</td>
<td>1, 1</td>
</tr>
<tr>
<td>0, 2</td>
<td>1, 2</td>
</tr>
<tr>
<td>0, 3</td>
<td>1, 3</td>
</tr>
<tr>
<td>1, 0</td>
<td>2, 0</td>
</tr>
<tr>
<td>1, 1</td>
<td>2, 1</td>
</tr>
<tr>
<td>1, 2</td>
<td>2, 2</td>
</tr>
<tr>
<td>1, 3</td>
<td>2, 3</td>
</tr>
<tr>
<td>2, 0</td>
<td>3, 0</td>
</tr>
<tr>
<td>2, 1</td>
<td>3, 1</td>
</tr>
<tr>
<td>2, 2</td>
<td>3, 2</td>
</tr>
<tr>
<td>2, 3</td>
<td>3, 3</td>
</tr>
<tr>
<td>3, 0</td>
<td>4, 0</td>
</tr>
<tr>
<td>3, 1</td>
<td>4, 1</td>
</tr>
<tr>
<td>3, 2</td>
<td>4, 2</td>
</tr>
<tr>
<td>3, 3</td>
<td>4, 3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 1

for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Dependence exists

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>0,0</td>
<td></td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>0,2</td>
<td>1,2</td>
<td>0,2</td>
<td></td>
</tr>
<tr>
<td>0,3</td>
<td>1,3</td>
<td>0,3</td>
<td></td>
</tr>
<tr>
<td>1,0</td>
<td>2,0</td>
<td>1,0</td>
<td></td>
</tr>
<tr>
<td>1,1</td>
<td>2,1</td>
<td>1,1</td>
<td></td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
<td>1,2</td>
<td></td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>1,3</td>
<td></td>
</tr>
<tr>
<td>2,0</td>
<td>3,0</td>
<td>2,0</td>
<td></td>
</tr>
<tr>
<td>2,1</td>
<td>3,1</td>
<td>2,1</td>
<td></td>
</tr>
<tr>
<td>2,2</td>
<td>3,2</td>
<td>2,2</td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td>3,3</td>
<td>2,3</td>
<td></td>
</tr>
<tr>
<td>3,0</td>
<td>4,0</td>
<td>3,0</td>
<td></td>
</tr>
<tr>
<td>3,1</td>
<td>4,1</td>
<td>3,1</td>
<td></td>
</tr>
<tr>
<td>3,2</td>
<td>4,2</td>
<td>3,2</td>
<td></td>
</tr>
<tr>
<td>3,3</td>
<td>4,3</td>
<td>3,3</td>
<td></td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 1

for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }

Loop carried dependence exists if
• there are two distinct iteration vectors such that
• the index vectors of LHS and RHS are identical

Conclusion: Dependence exists
Iteration Vectors and Index Vectors: Example 1

```c
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }
```

Loop carried dependence exists if
- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

**Conclusion: Dependence exists**

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
</tr>
<tr>
<td>0,2</td>
<td>1,2</td>
</tr>
<tr>
<td>0,3</td>
<td>1,3</td>
</tr>
<tr>
<td>1,0</td>
<td>2,0</td>
</tr>
<tr>
<td>1,1</td>
<td>2,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
</tr>
<tr>
<td>2,0</td>
<td>3,0</td>
</tr>
<tr>
<td>2,1</td>
<td>3,1</td>
</tr>
<tr>
<td>2,2</td>
<td>3,2</td>
</tr>
<tr>
<td>2,3</td>
<td>3,3</td>
</tr>
<tr>
<td>3,0</td>
<td>4,0</td>
</tr>
<tr>
<td>3,1</td>
<td>4,1</td>
</tr>
<tr>
<td>3,2</td>
<td>4,2</td>
</tr>
<tr>
<td>3,3</td>
<td>4,3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i][j] = a[i][j] + 2;
    }
```

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LHS</td>
</tr>
<tr>
<td>0,0</td>
<td>0,0</td>
</tr>
<tr>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>0,2</td>
<td>0,2</td>
</tr>
<tr>
<td>0,3</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
<td>1,1</td>
</tr>
<tr>
<td>1,2</td>
<td>1,2</td>
</tr>
<tr>
<td>1,3</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
<td>2,1</td>
</tr>
<tr>
<td>2,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>3,0</td>
<td>3,0</td>
</tr>
<tr>
<td>3,1</td>
<td>3,1</td>
</tr>
<tr>
<td>3,2</td>
<td>3,2</td>
</tr>
<tr>
<td>3,3</td>
<td>3,3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i][j] = a[i][j] + 2;
    }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LHS</td>
</tr>
<tr>
<td>0,0</td>
<td>0,0</td>
</tr>
<tr>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>0,2</td>
<td>0,2</td>
</tr>
<tr>
<td>0,3</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
<td>1,1</td>
</tr>
<tr>
<td>1,2</td>
<td>1,2</td>
</tr>
<tr>
<td>1,3</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
<td>2,1</td>
</tr>
<tr>
<td>2,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>3,0</td>
<td>3,0</td>
</tr>
<tr>
<td>3,1</td>
<td>3,1</td>
</tr>
<tr>
<td>3,2</td>
<td>3,2</td>
</tr>
<tr>
<td>3,3</td>
<td>3,3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 2

for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i][j] = a[i][j] + 2;
    }

Loop carried dependence exists if

- there are two distinct iteration
  vectors such that

- the index vectors of LHS and RHS
  are identical

Conclusion: No dependence
Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i];    /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T;    /* S3 */
}
```

Dependence Graph
Example 4: Dependence

Program to swap arrays

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;N; i++)</td>
<td>Dependence Graph</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Image]</td>
</tr>
<tr>
<td>T = A[i]; /* S1 */</td>
<td></td>
</tr>
<tr>
<td>A[i] = B[i]; /* S2 */</td>
<td></td>
</tr>
<tr>
<td>B[i] = T; /* S3 */</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Loop independent anti dependence due to A[i]
Example 4: Dependence

Program to swap arrays

```c
for (i=0; i<N; i++) {
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}
```

Dependence Graph

Loop independent anti dependence due to $B[i]$
Example 4: Dependence

Program to swap arrays

for (i=0; i<N; i++)
{
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}

Dependence Graph

Loop independent flow dependence due to T
Example 4: Dependence

Program to swap arrays

for (i=0; i<N; i++)
{
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}

Dependence Graph

Loop carried anti dependence due to T
Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}
```

Dependence Graph

Loop carried output dependence due to T
### Example 4: Dependence

**Program to swap arrays**

```c
for (i=0; i<N; i++)
{
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}
```

---

**Dependence Graph**

![Dependence Graph](image-url)
Data Dependence Theorem

There exists a dependence from statement \( S_1 \) to statement \( S_2 \) in common nest of loops if and only if there exist two iteration vectors \( i \) and \( j \) for the nest, such that

1. \( i < j \) or \( i = j \) and there exists a path from \( S_1 \) to \( S_2 \) in the body of the loop,

2. statement \( S_1 \) accesses memory location \( M \) on iteration \( i \) and statement \( S_2 \) accesses location \( M \) on iteration \( j \), and

3. one of these accesses is a write access.
Anti Dependence and Vectorization

Read precedes Write lexicographically

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: C[i] = A[i+2];
    S2: A[i] = B[i];
}
```
Anti Dependence and Vectorization

Read precedes Write lexicographically

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: C[i] = A[i+2];
    S2: A[i] = B[i];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S1: C[i:i+3] = A[i+2:i+5];
    S2: A[i:i+3] = B[i:i+3];
}
```
Anti Dependence and Vectorization

Write precedes Read lexicographically

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: C[i] = A[i+2];
}
```
Anti Dependence and Vectorization

Write precedes Read lexicographically

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: C[i] = A[i+2];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S2: C[i] = A[i+2];
    S1: A[i] = B[i];
}
```
Anti Dependence and Vectorization

Write precedes Read lexicographically

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: C[i] = A[i+2];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S2: C[i] = A[i+2];
    S1: A[i] = B[i];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S2: C[i:i+3] = A[i+2:i+5];
    S1: A[i:i+3] = B[i:i+3];
}
```
True Dependence and Vectorization

Write precedes Read lexicographically

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S_1: A[i+2] = C[i];
    S_2: B[i] = A[i];
}
```
True Dependence and Vectorization

Write precedes Read lexicographically

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i+2] = C[i];
    S2: B[i] = A[i];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S1: A[i+2:i+5] = C[i:i+3];
    S1: B[i:i+3] = A[i:i+3];
}
```
Multiple Dependences and Vectorization

Anti Dependence and True Dependence

```c
int A[N], i;
for (i=0; i<N; i++) {
    L1: A[i] = A[i+2];
}
```
Ante Dependence and True Dependence

```c
int A[N], i;
for (i=0; i<N; i++) {
    L1: A[i] = A[i+2];
}
```

```c
int A[N], i, temp;
for (i=0; i<N; i++) {
    S1: temp = A[i+2];
    S2: A[i] = temp;
}
```
Multiple Dependences and Vectorization

Anti Dependence and True Dependence

```c
int A[N], i;
for (i=0; i<N; i++) {
    L1: A[i] = A[i+2];
}
```

```c
int A[N], i, temp;
for (i=0; i<N; i++) {
    S1: temp = A[i+2];
    S2: A[i] = temp;
}
```

```c
int A[N], T[N], i;
for (i=0; i<N; i++) {
    S1: T[i] = A[i+2];
    S2: A[i] = T[i];
}
```
Multiple Dependences and Vectorization

Anti Dependence and True Dependence

```c
int A[N], i;
for (i=0; i<N; i++) {
    L1: A[i] = A[i+2];
}
```

```c
int A[N], i, temp;
for (i=0; i<N; i++) {
    S1: temp = A[i+2];
    S2: A[i] = temp;
}
```

```c
int A[N], T[N], i;
for (i=0; i=N; i=i+4) {
    S1: T[i:i+3] = A[i+2:i+5];
    S2: A[i:i+3] = T[i:i+3];
}
```

```c
int A[N], T[N], i;
for (i=0; i<N; i++) {
    S1: T[i] = A[i+2];
    S2: A[i] = T[i];
}
```
Multiple Dependences and Vectorization

True Dependence and Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S_1: A[i] = B[i];
    S_2: B[i+2] = A[i+1];
}
```
Multiple Dependences and Vectorization

True Dependence and Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: B[i+2] = A[i+1];
}
```

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S2: B[i+2] = A[i+1];
    S1: A[i] = B[i];
}
```
Multiple Dependences and Vectorization

True Dependence and Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: B[i+2] = A[i+1];
}
```

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S2: B[i+2] = A[i+1];
    S1: A[i] = B[i];
}
```

```c
int A[N], B[N], i;
for (i=0; i<N; i=i+4) {
    S2: B[i+2:i+5] = A[i+1:i+4];
    S1: A[i:i+3] = B[i:i+3];
}
```
Observation: Feasibility of Vectorization

- If the source statement lexicographically precedes sink statement in the program, they can be vectorized.
True Dependence and Vectorization

Read precedes Write lexicographically

```c
int A[N], i;
for (i=0; i<N; i++) {
}
```
True Dependence and Vectorization

Read precedes Write lexicographically

```c
int A[N], i;
for (i=0; i<N; i++) {
}
```

```c
int A[N], i, temp;
for (i=0; i<N; i++) {
    S1: temp = A[i];
    S2: A[i+5] = temp;
}
```
True Dependence and Vectorization

Read precedes Write lexicographically

```
int A[N], i;
for (i=0; i<N; i++) {
}
```

```
int A[N], i, temp;
for (i=0; i<N; i++) {
    S1: temp = A[i];
    S2: A[i+5] = temp;
}
```

```
int A[N], T[N], i;
for (i=0; i<N; i++) {
    S1: T[i] = A[i];
    S2: A[i+5] = T[i];
}
```
True Dependence and Vectorization

Read precedes Write lexicographically

```
int A[N], i;
for (i=0; i<N; i++) {
}
```

```
int A[N], i, temp;
for (i=0; i<N; i++) {
    S1: temp = A[i];
    S2: A[i+5] = temp;
}
```

```
int A[N], T[N], i;
for (i=0; i<N; i=i+4) {
    S1: T[i:i+3] = A[i:i+3];
    S2: A[i+5:i+8] = T[i:i+3];
}
```

```
int A[N], T[N], i;
for (i=0; i<N; i++) {
    S1: T[i] = A[i];
    S2: A[i+5] = T[i];
}
```
Cyclic Dependences and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S_1: B[i+2] = A[i];
    S_2: A[i+1] = B[i];
}
```
Cyclic Dependences and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+1];
    S2: A[i] = B[i+2];
}
```
Cyclic Dependences and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S_1: B[i+2] = A[i];
    S_2: A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S_1: B[i] = A[i+1];
    S_2: A[i] = B[i+2];
}
```

- Rescheduling of statements will not break the cyclic dependence
- The dependence distance from $S_2$ to $S_1 < VF$
Cyclic Dependences and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S_1: B[i+2] = A[i];
    S_2: A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S_1: B[i] = A[i+1];
    S_2: A[i] = B[i+2];
}
```

- Rescheduling of statements will not break the cyclic dependence
- The dependence distance from $S_2$ to $S_1 < VF$

Cannot Vectorize
Cyclic Dependences and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```
Cyclic Dependences and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S_1: B[i+2] = A[i];
    S_2: A[i+5] = B[i];
}
```

Cyclic Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S_1: B[i] = A[i+1];
    S_2: A[i] = B[i+5];
}
```
Cyclic Dependences and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```

Cyclic Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+1];
    S2: A[i] = B[i+5];
}
```

- Rescheduling of statements will not break the cyclic dependence
- The dependence distance from $S_2$ to $S_1 \geq VF$
Cyclic Dependences and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```

Cyclic Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+1];
    S2: A[i] = B[i+5];
}
```

- Rescheduling of statements will not break the cyclic dependence
- The dependence distance from $S_2$ to $S_1 \geq VF$

Can Vectorize
Observation: Feasibility of Vectorization

- If the source statement lexicographically precedes sink statement in the program, they can be vectorized.
- If the dependence distance for all *backward* dependences between two statements is greater than or equal to Vectorization Factor, the statements can be vectorized.
For feasibility of parallelization, consider the following code snippet:

```c
for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        A[i][j] = A[i][j+1];
```

This code snippet shows a nested loop structure where the outer loop iterates over `i` from 1 to `n`, and the inner loop iterates over `j` from 1 to `n`. The assignment `A[i][j] = A[i][j+1];` updates the element at position `(i, j)` in a 2D array with the value of the next element `(i, j+1)`.

The diagram illustrates the execution flow of this code, with arrows showing the dependency between iterations. Each arrow indicates that the value of `A[i][j]` is assigned from `A[i][j+1]` in the outer loop, maintaining the parallelization potential.

Essential Abstractions in GCC
**Feasibility of Parallelization**

Outer Parallel

```c
for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        A[i][j] = A[i][j+1];
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Feasibility of Parallelization

Outer Parallel

for-all (i=1 to n)
  for (j=1; j<n; j++)
    A[i][j] = A[i][j+1];
Feasibility of Parallelization

Inner Parallel

```
for (i=2; i<n; i++)
    for (j=1; j<n; j++)
        A[i][j] = A[i-1][j];
```
Feasibility of Parallelization

Inner Parallel

```
for (i=2; i<n; i++)
    for (j=1; j<n; j++)
        A[i][j] = A[i-1][j];
```
Feasibility of Parallelization

Inner Parallel

for (i=2; i<n; i++)
    for-all (j=1 to n)
        A[i][j] = A[i-1][j];
Part 2

The Lambda Framework
Lambda Framework for Loop Transforms

- Getting loop information (Loop discovery)
- Finding value spaces of induction variables, array subscript functions, and pointer accesses
- Analyzing data dependence
- Performing loop transformations
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.)
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.)
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.)
# Loop Transformation Passes in GCC: Our Focus

<table>
<thead>
<tr>
<th>Pass Variable Name</th>
<th>Pass Variable Name</th>
<th>Pass Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Dependence</td>
<td>pass_check_data_deps</td>
<td>pass_check_data_deps</td>
</tr>
<tr>
<td>Enabling switch</td>
<td>-fcheck-data-deps</td>
<td>-fcheck-data-deps</td>
</tr>
<tr>
<td>Dump switch</td>
<td>-fdump-tree-ckdd</td>
<td>-fdump-tree-ckdd</td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.ckdd</td>
<td>.ckdd</td>
</tr>
<tr>
<td>Loop Distribution</td>
<td>pass_loop_distribution</td>
<td>pass_loop_distribution</td>
</tr>
<tr>
<td>Enabling switch</td>
<td>-ftree-loop-distribution</td>
<td>-ftree-loop-distribution</td>
</tr>
<tr>
<td>Dump switch</td>
<td>-fdump-tree-ldist</td>
<td>-fdump-tree-ldist</td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.ldist</td>
<td>.ldist</td>
</tr>
<tr>
<td>Vectorization</td>
<td>pass_vectorize</td>
<td>pass_vectorize</td>
</tr>
<tr>
<td>Enabling switch</td>
<td>-ftree-vectorize</td>
<td>-ftree-vectorize</td>
</tr>
<tr>
<td>Dump switch</td>
<td>-fdump-tree-vect</td>
<td>-fdump-tree-vect</td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.vect</td>
<td>.vect</td>
</tr>
<tr>
<td>Parallelization</td>
<td>pass_parallelize_loops</td>
<td>pass_parallelize_loops</td>
</tr>
<tr>
<td>Enabling switch</td>
<td>-ftree-parallelize-loops=n</td>
<td>-ftree-parallelize-loops=n</td>
</tr>
<tr>
<td>Dump switch</td>
<td>-fdump-tree-parloops</td>
<td>-fdump-tree-parloops</td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.parloops</td>
<td>.parloops</td>
</tr>
</tbody>
</table>
Compiling for Emitting Dumps

- Other necessary command line switches
  - `-O2 -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 ⟨Starting Value, modification, stride⟩

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

<table>
<thead>
<tr>
<th>Entity</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Induction variable i</td>
<td>{3, +, 3}</td>
</tr>
<tr>
<td>Induction variable j</td>
<td>{11, +, −2}</td>
</tr>
<tr>
<td>Index expression i+1</td>
<td>{4, +, 3}</td>
</tr>
<tr>
<td>Index expression 2*j−1</td>
<td>{21, +, −4}</td>
</tr>
</tbody>
</table>
Example 1: Observing Data Dependence

Step 0: Compiling

```c
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 -O2 -S datadep.c
```
### Example 1: Observing Data Dependence

**Step 1: Examining the control flow graph**

<table>
<thead>
<tr>
<th>Program</th>
<th>Control Flow Graph</th>
</tr>
</thead>
</table>
| int a[200];
int main()
{
    int i;
    for (i=0; i<150; i++)
    {
        a[i] = a[i+1] + 2;
    }
    return 0;
} | <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

<table>
<thead>
<tr>
<th>Program</th>
<th>Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[200]; int main() { int i; for (i=0; i&lt;150; i++) { a[i] = a[i+1] + 2; } return 0; }</td>
<td>&lt;bb 3&gt;: # i_13 = PHI &lt;i_3(4), 0(2)&gt; 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 &lt;bb 4&gt;; else goto &lt;bb 5&gt;; &lt;bb 4&gt;: goto &lt;bb 3&gt;;</td>
</tr>
</tbody>
</table>
## Example 1: Observing Data Dependence

**Step 1: Examining the control flow graph**

<table>
<thead>
<tr>
<th>Program</th>
<th>Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[200]; int main() { int i; for (i=0; i&lt;150; i++) { a[i] = a[i+1] + 2; } return 0; }</td>
<td>&lt;bb 3&gt;: # i_13 = PHI &lt;i_3(4), 0(2)&gt; 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 &lt;bb 4&gt;; else goto &lt;bb 5&gt;; &lt;bb 4&gt;: goto &lt;bb 3&gt;;</td>
</tr>
</tbody>
</table>
Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

<table>
<thead>
<tr>
<th>Program</th>
<th>Control Flow Graph</th>
</tr>
</thead>
</table>
| int a[200];
int main()
{
    int i;
    for (i=0; i<150; i++)
    {
        a[i] = a[i+1] + 2;
    }
    return 0;
} | <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>;
Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```c
<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)
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: 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 -O2

```c
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 -ftree-vectorize
## Example 2: Observing Vectorization and Parallelization

### Step 1: Examining the control flow graph

<table>
<thead>
<tr>
<th>Program</th>
<th>Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[256], b[256];</td>
<td>&lt;bb 3&gt;:</td>
</tr>
<tr>
<td>int main()</td>
<td># i_11 = PHI &lt;i_4(4), 0(2)&gt;</td>
</tr>
<tr>
<td>{</td>
<td>D.2836_3 = b[i_11];</td>
</tr>
<tr>
<td>int i;</td>
<td>a[i_11] = D.2836_3;</td>
</tr>
<tr>
<td>for (i=0; i&lt;256; i++)</td>
<td>i_4 = i_11 + 1;</td>
</tr>
<tr>
<td>{</td>
<td>if (i_4 != 256)</td>
</tr>
<tr>
<td>a[i] = b[i];</td>
<td>goto &lt;bb 4&gt;;</td>
</tr>
<tr>
<td>}</td>
<td>else</td>
</tr>
<tr>
<td>return 0;</td>
<td>goto &lt;bb 5&gt;;</td>
</tr>
<tr>
<td>}</td>
<td>&lt;bb 4&gt;:</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 3&gt;;</td>
</tr>
</tbody>
</table>
Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

<table>
<thead>
<tr>
<th>Program</th>
<th>Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[256], b[256]; int main()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>&lt;bb 3&gt;:</td>
</tr>
<tr>
<td>int i;</td>
<td># i_11 = PHI &lt;i_4(4), 0(2)&gt;</td>
</tr>
<tr>
<td>for (i=0; i&lt;256; i++)</td>
<td>D.2836_3 = b[i_11];</td>
</tr>
<tr>
<td>{</td>
<td>a[i_11] = D.2836_3;</td>
</tr>
<tr>
<td>a[i] = b[i];</td>
<td>i_4 = i_11 + 1;</td>
</tr>
<tr>
<td>}</td>
<td>if (i_4 != 256)</td>
</tr>
<tr>
<td>return 0;</td>
<td>goto &lt;bb 4&gt;;</td>
</tr>
<tr>
<td>}</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 5&gt;;</td>
</tr>
<tr>
<td></td>
<td>&lt;bb 4&gt;:</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 3&gt;;</td>
</tr>
</tbody>
</table>
Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

<table>
<thead>
<tr>
<th>Program</th>
<th>Control Flow Graph</th>
</tr>
</thead>
</table>
| int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
} | <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 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

<table>
<thead>
<tr>
<th>Original control flow graph</th>
<th>Transformed control flow graph</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;bb 3&gt;</code>:</td>
<td><code>&lt;bb 2&gt;</code>:</td>
</tr>
<tr>
<td># i_11 = PHI &lt;i_4(4), 0(2)&gt;</td>
<td>vect_pb.7.10 = &amp;b;</td>
</tr>
<tr>
<td>D.2836_3 = b[i_11];</td>
<td>vect_pa.12.15 = &amp;a;</td>
</tr>
<tr>
<td>a[i_11] = D.2836_3;</td>
<td><code>&lt;bb 3&gt;</code>:</td>
</tr>
<tr>
<td>i_4 = i_11 + 1;</td>
<td># vect_pb.4.6 = PHI &lt;vect_pb.4.13,</td>
</tr>
<tr>
<td>if (i_4 != 256)</td>
<td>vect_pb.7.10&gt;</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td># vect_pa.9.16 = PHI &lt;vect_pa.9.17,</td>
</tr>
<tr>
<td>else</td>
<td>vect_pa.12.15&gt;</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>vect_var_.8.14 = MEM[vect_pb.4.6];</td>
</tr>
<tr>
<td><code>&lt;bb 4&gt;</code>:</td>
<td>MEM[vect_pa.9.16] = vect_var_.8.14;</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td>vect_pb.4.13 = vect_pb.4.6 + 16;</td>
</tr>
<tr>
<td></td>
<td>vect_pa.9.17 = vect_pa.9.16 + 16;</td>
</tr>
<tr>
<td></td>
<td>ivtmp.13.19 = ivtmp.13.18 + 1;</td>
</tr>
<tr>
<td></td>
<td>if (ivtmp.13.19 &lt; 64)</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 4&gt;;</td>
</tr>
</tbody>
</table>
**Example 2: Observing Vectorization and Parallelization**

**Step 3: Examining the vectorized control flow graph**

<table>
<thead>
<tr>
<th>Original control flow graph</th>
<th>Transformed control flow graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 2&gt;:</td>
</tr>
<tr>
<td># i_11 = PHI &lt;i_4(4), 0(2)&gt; D.2836_3 = b[i_11]; a[i_11] = D.2836_3; i_4 = i_11 + 1; if (i_4 != 256) goto &lt;bb 4&gt;; else goto &lt;bb 5&gt;;</td>
<td>vect_pb.7_10 = &amp;b; vect_pa.12_15 = &amp;a;</td>
</tr>
<tr>
<td></td>
<td>&lt;bb 3&gt;:</td>
</tr>
<tr>
<td># vect_pb.4_6 = PHI &lt;vect_pb.4_13, vect_pb.7_10&gt;</td>
<td>vect_pb.4_6 = vect_pb.4_6 + 16;</td>
</tr>
<tr>
<td></td>
<td>vect_pa.9_16 = PHI &lt;vect_pa.9_17, vect_pb.7_10&gt;</td>
</tr>
<tr>
<td></td>
<td>vect_var_.8_14 = MEM[vect_pb.4_6];</td>
</tr>
<tr>
<td></td>
<td>vect_pb.4_13 = vect_pb.4_6 + 16;</td>
</tr>
<tr>
<td></td>
<td>vect_pa.9_17 = vect_pa.9_16 + 16;</td>
</tr>
<tr>
<td></td>
<td>ivtmp.13_19 = ivtmp.13_18 + 1;</td>
</tr>
<tr>
<td></td>
<td>if (ivtmp.13_19 &lt; 64) goto &lt;bb 4&gt;;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example 2: Observing Vectorization and Parallelization

**Step 3: Examining the vectorized control flow graph**

<table>
<thead>
<tr>
<th>Original control flow graph</th>
<th>Transformed control flow graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;: # i_11 = PHI &lt;i_4(4), 0(2)&gt;</td>
<td>&lt;bb 2&gt;: vect_pb.7_10 = &amp;b;</td>
</tr>
<tr>
<td>D.2836_3 = b[i_11];</td>
<td>vect_pb.7_10 = &amp;b;</td>
</tr>
<tr>
<td>a[i_11] = D.2836_3;</td>
<td>vect_pb.7_10 = &amp;b;</td>
</tr>
<tr>
<td>i_4 = i_11 + 1;</td>
<td>vect_pb.4_6 = PHI &lt;vect_pb.4_13,</td>
</tr>
<tr>
<td>if (i_4 != 256)</td>
<td>vect_pb.7_10&gt;</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td># vect_pb.4_6 = PHI &lt;vect_pb.4_13,</td>
</tr>
<tr>
<td>else</td>
<td>vect_pb.7_10&gt;</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td># vect_pa.9_16 = PHI &lt;vect_pa.9_17,</td>
</tr>
<tr>
<td>&lt;bb 4&gt;: goto &lt;bb 3&gt;;</td>
<td>vect_pa.12_15,</td>
</tr>
<tr>
<td></td>
<td>vect_pb.7_10&gt;</td>
</tr>
<tr>
<td></td>
<td>vect_ra.9_16 = PHI &lt;vect_pa.9_17,</td>
</tr>
<tr>
<td></td>
<td>vect_pb.7_10&gt;</td>
</tr>
<tr>
<td></td>
<td>vect_pb.4_13 = vect_pb.4_6 + 16;</td>
</tr>
<tr>
<td></td>
<td>vect_pb.9_17 = vect_pa.9_16 + 16;</td>
</tr>
<tr>
<td></td>
<td>ivtmp.13_19 = ivtmp.13_18 + 1;</td>
</tr>
<tr>
<td></td>
<td>if (ivtmp.13_19 &lt; 64)</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 4&gt;;</td>
</tr>
</tbody>
</table>
Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

<table>
<thead>
<tr>
<th>Original control flow graph</th>
<th>Transformed control flow graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 2&gt;:</td>
</tr>
<tr>
<td># i(<em>{11}) = PHI (i</em>{4}(4), 0(2))</td>
<td>vect(<em>{pb}.7</em>{10}) = &amp;b;</td>
</tr>
<tr>
<td>D.2836_{3} = b[i_{11}];</td>
<td>vect(<em>{pa}.12</em>{15}) = &amp;a;</td>
</tr>
<tr>
<td>a[i_{11}] = D.2836_{3};</td>
<td></td>
</tr>
<tr>
<td>i_{4} = i_{11} + 1;</td>
<td># vect(<em>{pb}.4</em>{6}) = PHI</td>
</tr>
<tr>
<td>if (i_{4} != 256)</td>
<td>(\langle\text{vect(<em>{pb}.4</em>{13})},\text{vect(<em>{pb}.7</em>{10})}\rangle)</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td># vect(<em>{pa}.9</em>{16}) = PHI</td>
</tr>
<tr>
<td>else</td>
<td>(\langle\text{vect(<em>{pa}.9</em>{17})},\text{vect(<em>{pa}.12</em>{15})}\rangle)</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>vect(<em>{var}.8</em>{14}) = MEM[\text{vect(<em>{pb}.4</em>{6})}];</td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td>MEM[\text{vect(<em>{pa}.9</em>{16})}] = vect(<em>{var}.8</em>{14});</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td>vect(<em>{pb}.4</em>{13}) = vect(<em>{pb}.4</em>{6} + 16);</td>
</tr>
<tr>
<td></td>
<td>vect(<em>{pa}.9</em>{17}) = vect(<em>{pa}.9</em>{16} + 16);</td>
</tr>
<tr>
<td></td>
<td>ivtmp(<em>{13}</em>{19}) = ivtmp(<em>{13}</em>{18} + 1);</td>
</tr>
<tr>
<td></td>
<td>if (ivtmp(<em>{13}</em>{19} &lt; 64))</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 4&gt;;</td>
</tr>
</tbody>
</table>
Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

<table>
<thead>
<tr>
<th>Original control flow graph</th>
<th>Transformed control flow graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 2&gt;:</td>
</tr>
<tr>
<td># i_11 = PHI &lt;i_4(4), 0(2)&gt;</td>
<td>vect_pb.7_10 = &amp;b;</td>
</tr>
<tr>
<td>D.2836_3 = b[i_11];</td>
<td>vect_pa.12_15 = &amp;a;</td>
</tr>
<tr>
<td>a[i_11] = D.2836_3;</td>
<td></td>
</tr>
<tr>
<td>i_4 = i_11 + 1;</td>
<td># vect_pb.4_6 = PHI &lt;vect_pb.4_13,</td>
</tr>
<tr>
<td>if (i_4 != 256)</td>
<td>vect_pb.7_10&gt;</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td># vect_pa.9_16 = PHI &lt;vect_pa.9_17,</td>
</tr>
<tr>
<td>else</td>
<td>vect_pa.12_15&gt;</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>vect_var_.8_14 = MEM[vect_pb.4_6];</td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td>MEM[vect_pa.9_16] = vect_var_.8_14;</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td>vect_pb.4_13 = vect_pb.4_6 + 16;</td>
</tr>
<tr>
<td></td>
<td>vect_pa.9_17 = vect_pa.9_16 + 16;</td>
</tr>
<tr>
<td></td>
<td>ivtmp.13_19 = ivtmp.13_18 + 1;</td>
</tr>
<tr>
<td></td>
<td>if (ivtmp.13_19 &lt; 64)</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 4&gt;;</td>
</tr>
</tbody>
</table>
Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads $t_i$ for $1 \leq i \leq \text{MAX\_THREADS}$
Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads $t_i$ for $1 \leq i \leq \text{MAX\_THREADS}$
- Assigning start and end iteration for each thread
  ⇒ 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 $1 \leq i \leq \text{MAX\_THREADS}$
- Assigning start and end iteration for each thread
  ⇒ Distribute iteration space across all threads
- Create the following code body for each thread $t_i$

```c
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
    /* execute the loop body to be parallelized */
}
```
3 July 2012

gcc-par-vect: The Lambda Framework

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution
• Create threads ti for 1 ≤ i ≤ MAX THREADS
• Assigning start and end iteration for each thread

⇒ Distribute iteration space across all threads
• Create the following code body for each thread ti

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

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

45/81


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.2002_12 = D.2001_11 != 255;
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

```c
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2002_12 = D.2001_11 != 255;
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

```c
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2002_12 = D.2001_11 != 255;
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.2002_12 = D.2001_11 != 255;
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

```c
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2002_12 = D.2001_11 != 255;
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.2002_12 = D.2001_11 != 255;
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.2002_12 = D.2001_11 != 255;
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

<table>
<thead>
<tr>
<th>&lt;bb 3&gt;</th>
<th>Parallel loop body</th>
</tr>
</thead>
<tbody>
<tr>
<td># i_11 = PHI &lt;i_4(4), 0(2)&gt;</td>
<td>&lt;bb 5&gt;:</td>
</tr>
<tr>
<td>D.1956_3 = b[i_11];</td>
<td>i.8_21 = (int) ivtmp.7_18;</td>
</tr>
<tr>
<td>a[i_11] = D.1956_3;</td>
<td>D.2010_23 = *b.10_4[i.8_21];</td>
</tr>
<tr>
<td>i_4 = i_11 + 1;</td>
<td>*a.11_5[i.8_21] = D.2010_23;</td>
</tr>
<tr>
<td>if (i_4 != 256)</td>
<td>ivtmp.7_19 = ivtmp.7_18 + 1;</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>if (D.2006_16 &gt; ivtmp.7_19)</td>
</tr>
<tr>
<td>else</td>
<td>goto &lt;bb 5&gt;;</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>else</td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td>goto &lt;bb 3&gt;;</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td></td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC GCC Resource Center, IIT Bombay
Example 2: Observing Vectorization and Parallelization

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

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Parallel loop body</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 5&gt;:</td>
</tr>
<tr>
<td># i_11 = PHI &lt;i_4(4), 0(2)&gt;</td>
<td>i.8_21 = (int) ivtmp.7_18;</td>
</tr>
<tr>
<td>D.1956_3 = b[i_11];</td>
<td>D.2010_23 = *b.10_4[i.8_21];</td>
</tr>
<tr>
<td>a[i_11] = D.1956_3;</td>
<td>*a.11_5[i.8_21] = D.2010_23;</td>
</tr>
<tr>
<td>i_4 = i_11 + 1;</td>
<td>ivtmp.7_19 = ivtmp.7_18 + 1;</td>
</tr>
<tr>
<td>if (i_4 != 256)</td>
<td>if (D.2006_16 &gt; ivtmp.7_19)</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>goto &lt;bb 5&gt;;</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>goto &lt;bb 5&gt;;</td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td>else</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td>goto &lt;bb 3&gt;;</td>
</tr>
</tbody>
</table>
Step 6: Examining the loop body to be executed by a thread

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Parallel loop body</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;bb 3&gt;</code>:</td>
<td><code>&lt;bb 5&gt;</code>:</td>
</tr>
<tr>
<td># i_11 = PHI <code>&lt;i_4(4), 0(2)&gt;</code></td>
<td>i.8_21 = (int) ivtmp.7_18;</td>
</tr>
<tr>
<td>D.1956_3 = b[i_11];</td>
<td>D.2010_23 = *b.10_4[i.8_21];</td>
</tr>
<tr>
<td>a[i_11] = D.1956_3;</td>
<td>*a.11_5[i.8_21] = D.2010_23;</td>
</tr>
<tr>
<td>i_4 = i_11 + 1;</td>
<td>ivtmp.7_19 = ivtmp.7_18 + 1;</td>
</tr>
<tr>
<td>if (i_4 != 256)</td>
<td>if (D.2006_16 &gt; ivtmp.7_19)</td>
</tr>
<tr>
<td>goto <code>&lt;bb 4&gt;</code>;</td>
<td>goto <code>&lt;bb 5&gt;</code>;</td>
</tr>
<tr>
<td>else</td>
<td>else goto <code>&lt;bb 5&gt;</code>;</td>
</tr>
<tr>
<td></td>
<td>goto <code>&lt;bb 3&gt;</code>;</td>
</tr>
<tr>
<td><code>&lt;bb 4&gt;</code>:</td>
<td>goto <code>&lt;bb 3&gt;</code>;</td>
</tr>
</tbody>
</table>
Example 2: Observing Vectorization and Parallelization

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

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Parallel loop body</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;bb 3&gt;</code>:</td>
<td><code>&lt;bb 5&gt;</code>:</td>
</tr>
<tr>
<td># i_11 = PHI <code>&lt;i_4(4), 0(2)&gt;</code></td>
<td>i.8_21 = (int) ivtmp.7_18;</td>
</tr>
<tr>
<td>D.1956_3 = b[i_11];</td>
<td>D.2010_23 = *b.10_4[i.8_21];</td>
</tr>
<tr>
<td>a[i_11] = D.1956_3;</td>
<td>*a.11_5[i.8_21] = D.2010_23;</td>
</tr>
<tr>
<td>i_4 = i_11 + 1;</td>
<td>ivtmp.7_19 = ivtmp.7_18 + 1;</td>
</tr>
<tr>
<td>if (i_4 != 256)</td>
<td>if (D.2006_16 &gt; ivtmp.7_19)</td>
</tr>
<tr>
<td>goto <code>&lt;bb 4&gt;</code>;</td>
<td>goto <code>&lt;bb 5&gt;</code>;</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>goto <code>&lt;bb 5&gt;</code>;</td>
<td>goto <code>&lt;bb 5&gt;</code>;</td>
</tr>
<tr>
<td><code>&lt;bb 4&gt;</code>:</td>
<td>else</td>
</tr>
<tr>
<td>goto <code>&lt;bb 3&gt;</code>;</td>
<td>goto <code>&lt;bb 3&gt;</code>;</td>
</tr>
</tbody>
</table>
Example 2: Observing Vectorization and Parallelization

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

Control Flow Graph

\[
\begin{align*}
\text{<bb 3>:} \\
& \# i_{11} = \text{PHI} \ <i\_4(4), 0(2)> \\
& D.1956_3 = b[i_{11}]; \\
& a[i_{11}] = D.1956_3; \\
& i_4 = i_{11} + 1; \\
& \text{if (i_4 != 256)} \\
& \quad \text{goto <bb 4>}; \\
& \text{else} \\
& \quad \text{goto <bb 5>}; \\
\text{<bb 4>:} \\
& \quad \text{goto <bb 3>};
\end{align*}
\]

Parallel loop body

\[
\begin{align*}
\text{<bb 5>:} \\
& i.8_{21} = \text{(int) ivtmp.7_{18}}; \\
& D.2010_{23} = *b.10_{4}[i.8_{21}]; \\
& *a.11_{5}[i.8_{21}] = D.2010_{23}; \\
& \text{ivtmp.7_{19} = ivtmp.7_{18} + 1}; \\
& \text{if (D.2006_{16} > ivtmp.7_{19})} \\
& \quad \text{goto <bb 5>}; \\
& \text{else} \\
& \quad \text{goto <bb 3>};
\end{align*}
\]
Example 3: Vectorization but No Parallelization

Step 0: Compiling with

```bash
-O2 -fdump-tree-vect-all -msse4 -ftree-vectorize
```

```c
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

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 2&gt;:</td>
</tr>
<tr>
<td></td>
<td>vect_pa.10.26 = &amp;a[4];</td>
</tr>
<tr>
<td></td>
<td>vect_pa.15.30 = &amp;a;</td>
</tr>
<tr>
<td></td>
<td>&lt;bb 3&gt;:</td>
</tr>
<tr>
<td></td>
<td># vect_pa.7.27 = PHI &lt;vect_pa.7.28, vect_pa.10.26&gt;</td>
</tr>
<tr>
<td></td>
<td># vect_pa.12.31 = PHI &lt;vect_pa.12.32, vect_pa.15.30&gt;</td>
</tr>
<tr>
<td></td>
<td>vect_var_.11.29 = MEM[vect_pa.7.27];</td>
</tr>
<tr>
<td></td>
<td>MEM[vect_pa.12.31] = vect_var_.11.29;</td>
</tr>
<tr>
<td></td>
<td>vect_pa.7.28 = vect_pa.7.27 + 16;</td>
</tr>
<tr>
<td></td>
<td>vect_pa.12.32 = vect_pa.12.31 + 16;</td>
</tr>
<tr>
<td></td>
<td>ivtmp.16.34 = ivtmp.16.33 + 1;</td>
</tr>
<tr>
<td></td>
<td>if (ivtmp.16.34 &lt; 154) goto &lt;bb 4&gt;;}</td>
</tr>
</tbody>
</table>
Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 2&gt;:</td>
</tr>
<tr>
<td># i_12 = PHI &lt;i_5(4), 0(2)&gt;</td>
<td>vect_pa.10_26 = &amp;a[4];</td>
</tr>
<tr>
<td>D.2834_3 = i_12 + 4;</td>
<td>vect_pa.15_30 = &amp;a;</td>
</tr>
<tr>
<td>D.2835_4 = a[D.2834_3];</td>
<td>&lt;bb 3&gt;:</td>
</tr>
<tr>
<td>a[i_12] = D.2835_4;</td>
<td># vect_pa.7_27 = PHI &lt;vect_pa.7_28, vect_pa.10_26&gt;</td>
</tr>
<tr>
<td>i_5 = i_12 + 1;</td>
<td># vect_pa.12_31 = PHI &lt;vect_pa.12_32, vect_pa.15_30&gt;</td>
</tr>
<tr>
<td>if (i_5 != 619)</td>
<td>vect_var_.11_29 = MEM[vect_pa.7_27];</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>MEM[vect_pa.12_31] = vect_var_.11_29;</td>
</tr>
<tr>
<td>else</td>
<td>vect_pa.7_28 = vect_pa.7_27 + 16;</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>vect_pa.12_32 = vect_pa.12_31 + 16;</td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td>ivtmp.16_34 = ivtmp.16_33 + 1;</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td>if (ivtmp.16_34 &lt; 154)</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 4&gt;;</td>
</tr>
</tbody>
</table>
### Example 3: Vectorization but No Parallelization

#### Step 2: Examining vectorization

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{bb 3}$:</td>
<td></td>
</tr>
<tr>
<td># $i_{12} = \text{PHI } i_5(4), 0(2)$</td>
<td></td>
</tr>
<tr>
<td>$D.2834.3 = i_{12} + 4$</td>
<td></td>
</tr>
<tr>
<td>$D.2835.4 = a[D.2834.3]$</td>
<td></td>
</tr>
<tr>
<td>$a[i_{12}] = D.2835.4$</td>
<td></td>
</tr>
<tr>
<td>$i_5 = i_{12} + 1$</td>
<td></td>
</tr>
<tr>
<td>if ($i_5 != 619$)</td>
<td></td>
</tr>
<tr>
<td>\quad goto $\text{bb 4}$</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>\quad goto $\text{bb 5}$</td>
<td></td>
</tr>
<tr>
<td>$\text{bb 4}$:</td>
<td></td>
</tr>
<tr>
<td>goto $\text{bb 3}$</td>
<td></td>
</tr>
</tbody>
</table>

```c
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;
```

```c

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**

<table>
<thead>
<tr>
<th>Block</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;</td>
<td></td>
</tr>
</tbody>
</table>
# 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**

<table>
<thead>
<tr>
<th>Block</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 2&gt;</td>
<td></td>
</tr>
</tbody>
</table>

```c
vect_pa.10_26 = &a[4];
vect_pa.15_30 = &a;
```

<table>
<thead>
<tr>
<th>Block</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;</td>
<td></td>
</tr>
</tbody>
</table>
# 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

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
</table>

<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>;

<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

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td></td>
</tr>
<tr>
<td># i_12 = PHI &lt;i_5(4), 0(2)&gt;</td>
<td></td>
</tr>
<tr>
<td>D.2834_3 = i_12 + 4;</td>
<td></td>
</tr>
<tr>
<td>D.2835_4 = a[D.2834_3];</td>
<td></td>
</tr>
<tr>
<td>a[i_12] = D.2835_4;</td>
<td></td>
</tr>
<tr>
<td>i_5 = i_12 + 1;</td>
<td></td>
</tr>
<tr>
<td>if (i_5 != 619) goto &lt;bb 4&gt;;</td>
<td></td>
</tr>
<tr>
<td>else goto &lt;bb 5&gt;;</td>
<td></td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td></td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td></td>
</tr>
</tbody>
</table>

|                               |                               |
| <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 `-O2`

```c
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<216; i++)
    {
        a[i+2] = b[i] + 5;
        b[i+1] = 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` `-ftree-vectorize`
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 dependences that prohibit vectorization and parallelization

\[
a[i+2] = b[i] + 5
\]

\[
b[i+1] = a[i] + 10
\]
Part 3

Transformations Enhancing Vectorization and Parallelization
Some transformations increase the scope of parallelization and vectorization by either enabling them, or by improving their run time performance. Most important of such transformations are:

- Loop Interchange
- Loop Distribution
- Loop Fusion
- Peeling
Loop Interchange

Loop Interchange for Vectorization

Original Code

```c
for (i=0; i<200; i++) {
    for (j=0; j<200; j++)
        a[j][i] = a[j][i+1];
}
```
Loop Interchange

Loop Interchange for Vectorization

Original Code

```
for (i=0; i<200; i++) {
    for (j=0; j<200; j++)
        a[j][i] = a[j][i+1];
}
```

- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access
Loop Interchange

Loop Interchange for Vectorization

Original Code

```c
for (i=0; i<200; i++) {
    for (j=0; j<200; j++)
        a[j][i] = a[j][i+1];
}
```

- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access
Loop Interchange for Vectorization

- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access
Loop Interchange for Vectorization

Original Code

```c
for (i=0; i<200; i++) {
    for (j=0; j<200; j++)
        a[j][i] = a[j][i+1];
}
```

- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access
Loop Interchange

Loop Interchange for Vectorization

Original Code

```c
for (i=0; i<200; i++) {
    for (j=0; j<200; j++)
        a[j][i] = a[j][i+1];
}
```

- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access
Loop Interchange

Loop Interchange for Vectorization

Original Code

```c
for (i=0; i<200; i++) {
    for (j=0; j<200; j++)
        a[j][i] = a[j][i+1];
}
```

- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access
Loop Interchange

Loop Interchange for Vectorization

Original Code

```c
for (i=0; i<200; i++) {
    for (j=0; j<200; j++)
        a[j][i] = a[j][i+1];
}
```

After Interchange

```c
for (j=0; j<200; j++) {
    for (i=0; i<200; i++)
        a[j][i] = a[j][i+1];
}
```

- Innermost loop is vectorizable
- Loop Interchange improves data locality
Loop Interchange

Loop Interchange for Parallelization

Original Code

```c
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 synchronizations required = $n$
Loop Interchange

Loop Interchange for Parallelization

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

• Outer Loop - parallelizable

Total number of synchronizations required = 1
Loop Distribution

Original Code

```c
for (i=0; i<230; i++) {
    S1 : a[i+3] = a[i];
    S2 : b[i] = c[i];
}
```

- True dependence in $S_1$, no dependence in $S_2$
- Loop cannot be vectorized or parallelized, but $S_2$ can be vectorized and parallelized independently
Loop Distribution

Original Code

for (i=0; i<230; i++) {
    S1 : a[i+3] = a[i];
    S2 : b[i] = c[i];
}

- True dependence in $S_1$, no dependence in $S_2$
- Loop cannot be vectorized or parallelized, but $S_2$ can be vectorized and parallelized independently

Compile with

gcc -O2 -ftree-loop-distribution -fdump-tree-ldist
Loop Distribution

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Distributed Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 6&gt;:</td>
</tr>
<tr>
<td># i_13 = PHI &lt;i_6(4), 0(2)&gt;</td>
<td># i_11 = PHI &lt;i_18(7), 0(2)&gt;</td>
</tr>
<tr>
<td>D.2692_3 = i_13 + 3;</td>
<td>D.2692_12 = i_11 + 3;</td>
</tr>
<tr>
<td>D.2693_4 = a[i_13];</td>
<td>D.2693_7 = a[i_11];</td>
</tr>
<tr>
<td>D.2694_5 = c[i_13];</td>
<td>i_18 = i_11 + 1;</td>
</tr>
<tr>
<td>b[i_13] = D.2694_5;</td>
<td>if (i_18 != 230)</td>
</tr>
<tr>
<td>i_6 = i_13 + 1;</td>
<td>goto &lt;bb 6&gt;;</td>
</tr>
<tr>
<td>if (i_6 != 230)</td>
<td>else</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>goto &lt;bb 5&gt;;</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td></td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td>&lt;bb 8&gt;:</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td># i_13 = PHI &lt;i_6(4), 0(8)&gt;</td>
</tr>
<tr>
<td></td>
<td>D.2694_5 = c[i_13];</td>
</tr>
<tr>
<td></td>
<td>b[i_13] = D.2694_5;</td>
</tr>
<tr>
<td></td>
<td>i_6 = i_13 + 1;</td>
</tr>
<tr>
<td></td>
<td>if (i_6 != 230)</td>
</tr>
<tr>
<td></td>
<td>goto &lt;bb 8&gt;;</td>
</tr>
</tbody>
</table>
Loop Distribution

Control Flow Graph

<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.2692_3 = i_13 + 3;
  D.2693_4 = a[i_13];
  a[D.2692_3] = D.2693_4;
  D.2694_5 = c[i_13];
  b[i_13] = D.2694_5;
  i_6 = i_13 + 1;
  if (i_6 != 230)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;

Distributed Control Flow Graph

<bb 6>:
  # i_11 = PHI <i_18(7), 0(2)>
  D.2692_12 = i_11 + 3;
  D.2693_7 = a[i_11];
  a[D.2692_12] = D.2693_7;
  i_18 = i_11 + 1;
  if (i_18 != 230)
    goto <bb 6>;
<bb 8>:
  # i_13 = PHI <i_6(4), 0(8)>
  D.2694_5 = c[i_13];
  b[i_13] = D.2694_5;
  i_6 = i_13 + 1;
  if (i_6 != 230)
    goto <bb 8>;

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
## Loop Distribution

### Control Flow Graph

```plaintext
<bb 3>:

# i_13 = PHI <i_6(4), 0(2)>
D.2692_3 = i_13 + 3;
D.2693_4 = a[i_13];
a[D.2692_3] = D.2693_4;
D.2694_5 = c[i_13];
b[i_13] = D.2694_5;
i_6 = i_13 + 1;
if (i_6 != 230)
    goto <bb 3>;
else
    goto <bb 5>;

<bb 4>:

    goto <bb 3>;
```

### Distributed Control Flow Graph

```plaintext
<bb 6>:

# i_11 = PHI <i_18(7), 0(2)>
D.2692_12 = i_11 + 3;
D.2693_7 = a[i_11];
a[D.2692_12] = D.2693_7;
i_18 = i_11 + 1;
if (i_18 != 230)
    goto <bb 6>;
```

```plaintext
<bb 8>:

# i_13 = PHI <i_6(4), 0(8)>
D.2694_5 = c[i_13];
b[i_13] = D.2694_5;
i_6 = i_13 + 1;
if (i_6 != 230)
    goto <bb 8>;
```
Loop Distribution

After Distribution

```c
for (i=0; i<230; i++)
    S1 : a[i+3] = a[i];
for (i=0; i<230; i++)
    S2 : b[i] = c[i];
```

- $S_2$ can now be independently parallelized or vectorized
- $S_1$ runs sequentially
Loop Fusion for Locality

Original Code

```c
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = b[i];
for (k=0; k<n; k++)
    for (l=0; l<n; l++)
        b[k] = a[k][l];
```

- Large reuse distance for array `a` and `b`, high chances of cache miss
- If loops `i` and `k` are parallelized, 2 synchronizations required
- Outer loops `i` and `k` can be fused
- Fusing inner loops `j` and `l` will introduce a spurious backward dependence on `b`
Loop Fusion for Locality

Original Code

```c
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = b[i];
for (k=0; k<n; k++)
    for (l=0; l<n; l++)
        b[k] = a[k][l];
```

Fused Code

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
        a[i][j] = b[i];
    for (l=0; l<n; l++)
        b[i] = a[i][l];
}
```

- Reduced reuse distance for array `a` and `b`, low chances of cache miss
- If outer loop `i` is parallelized, only 1 synchronization required
Peeling for Vectorization

- Cyclic Dependence, dependence distance for *backward* dependence = 3 < VF
- Cannot vectorize
Peeling for Vectorization

**Transformed Code**

```c
for (i=0; i<2; i++)
  S2: b[i+3] = a[i];
for (i=2; i<n-2; i++) {
  S1: a[i] = b[i-2];
  S2: b[i+3] = a[i];
}
```

- Cyclic Dependence, dependence distance for *backward* dependence $= 5 > VF$
- Can vectorize
Peeling

Peeling for Parallelization

Original Code

```c
for (i=1; i<n; i++)
{
    S1: a[i] = b[i];
    S2: c[i] = a[i-1];
}
```

- dependence on i, can not be parallelized

Total number of synchronizations required = \( n \)
Peeling for Parallelization

Original Code

```c
for (i=1; i<n; i++) {
    S_1: a[i] = b[i];
    S_2: c[i] = a[i-1];
}
```

Transformed Code

```c
c[1] = a[0];
for (i=1; i<n-1; i++) {
    S_1: a[i] = b[i];
    S_2: c[i+1] = a[i];
}
```

- Outer Loop parallelizable

Total number of synchronizations required = 1
Part 4

Advanced Issues in Vectorization and Parallelization
Advanced Issues in Vectorization and Parallelization

• What code can be vectorized?
• How to force the alignment of data accesses for
  ▶ compile time misalignment
  ▶ run time misalignment
• How to handle undetermined aliases?
• When is vectorization profitable?
• When is parallelization profitable?
Advanced Issues in Vectorization and Parallelization

- What code can be vectorized?
- How to force the alignment of data accesses for
  - compile time misalignment
  - run time misalignment
- How to handle undetermined aliases?
- When is vectorization profitable?
- When is parallelization profitable?

Understanding the cost model of vectorizer and parallelizer
Unvectorizable Loops

```c
int *a, *b;
int main() {
    while (*a != NULL)
    {
        *a++ = *b--;
    }
}
```
Unvectorizable Loops

```c
int *a, *b;
int main() {
    while (*a != NULL)
    {
        *a++ = *b--;
    }
}
```

novec.c:6: note: not vectorized: number of iterations cannot be computed.
Reducing Compile Time Misalignment by Peeling

```c
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<203; i++)
        a[i+2] = b[i+2];
}
```
Reducing Compile Time Misalignment by Peeling

```c
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<203; i++)
        a[i+2] = b[i+2];
}
```

peel.c:5: note: misalign = 8 bytes of ref b[D.2836_4]
peel.c:5: note: misalign = 8 bytes of ref a[D.2836_4]
Reducing Compile Time Misalignment by Peeling

Observing the final decision about alignment

peel.c:5: note: Try peeling by 2
peel.c:5: note: Alignment of access forced using peeling.
peel.c:5: note: Peeling for alignment will be applied.

peel.c:5: note: known peeling = 2.
peel.c:5: note: niters for prologue loop: 2
peel.c:5: note: Cost model analysis:
    prologue iterations: 2
    epilogue iterations: 1
Reducing Compile Time Misalignment by Peeling

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
Reducing Compile Time Misalignment by Peeling

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 3&gt;:</td>
</tr>
<tr>
<td># i_12 = PHI &lt;i_6(4), 0(2)&gt;</td>
<td># ivtmp.8.27 = PHI &lt;ivtmp.8.28(4), 0(2)&gt;</td>
</tr>
<tr>
<td>D.2690_4 = i_12 + 2;</td>
<td>D.2908_16 = i_7 + 2;</td>
</tr>
<tr>
<td>D.2691_5 = b[D.2690_4];</td>
<td>D.2909_17 = b[D.2908_16];</td>
</tr>
<tr>
<td>i_6 = i_12 + 1;</td>
<td>ivtmp.8.28 = ivtmp.8.27 + 1;</td>
</tr>
<tr>
<td>if (i_6 ! = 203)</td>
<td>if (ivtmp.8.28 &lt; 2)</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>goto &lt;bb 3&gt;;</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>goto &lt;bb 5&gt;;</td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td></td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td></td>
</tr>
</tbody>
</table>

2 Iterations of Prologue
## Reducing Compile Time Misalignment by Peeling

### Control Flow Graph

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
</tr>
</thead>
</table>
| \[
\text{<bb 3>:}
\begin{align*}
\text{# i}_{12} &= \text{PHI} \ <i_{6}(4), 0(2)> \\
\text{D.2690}_4 &= i_{12} + 2; \\
\text{D.2691}_5 &= \text{b}[\text{D.2690}_4]; \\
\text{a}[\text{D.2690}_4] &= \text{D.2691}_5; \\
i_{6} &= i_{12} + 1; \\
\text{if (i}_{6} \neq 203) &\text{ goto <bb 4>; } \\
\text{else} &\text{ goto <bb 5>;} \\
\text{<bb 4>:} &\text{ goto <bb 3>;}
\end{align*}
\] |

### Vectorized Control Flow Graph

<table>
<thead>
<tr>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
\text{<bb 5>:} \\
\text{vect_pb.15}_4 &= \&\text{b}[4]; \\
\text{vect_pa.20}_8 &= \&\text{a}[4]; \\
\text{<bb 6>:} \\
\text{# vect_pb.12}_5 &= \text{PHI} <\text{vect_pb.12}_6, \\
\text{vect_pb.15}_4> \\
\text{# vect_pa.17}_9 &= \text{PHI} <\text{vect_pa.17}_3, \\
\text{vect_pa.20}_8> \\
\text{vect_var_.16}_7 &= \text{MEM}[\text{vect_pb.12}_5]; \\
\text{MEM}[\text{vect_pa.17}_9] &= \text{vect_var_.16}_7; \\
\text{vect_pb.12}_6 &= \text{vect_pb.12}_5 + 16; \\
\text{vect_pa.17}_3 &= \text{vect_pa.17}_9 + 16; \\
\text{ivtmp.21}_52 &= \text{ivtmp.21}_51 + 1; \\
\text{if (ivtmp.21}_52 < 50) &\text{ goto <bb 10>;}
\end{align*}
\] |

200 Iterations of Vector Code
### Reducing Compile Time Misalignment by Peeling

#### Control Flow Graph

```plaintext
<bb 3>:
  # i_12 = PHI <i_6(4), 0(2)>
  D.2690_4 = i_12 + 2;
  D.2691_5 = b[D.2690_4];
  a[D.2690_4] = D.2691_5;
  i_6 = i_12 + 1;
  if (i_6 != 203)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

#### Vectorized Control Flow Graph

```plaintext
<bb 7>:
  tmp.10_42 = ivtmp.8_28 + 200;
<bb 8>:
  # i_29 = PHI <i_35(9), tmp.10_42(7)>
  # ivtmp.3_31 = PHI <ivtmp.3_36(9),
    tmp.11_43(7)>
  D.2908_32 = i_29 + 2;
  D.2909_33 = b[D.2908_32];
  a[D.2908_32] = D.2909_33;
  i_35 = i_29 + 1;
  ivtmp.3_36 = ivtmp.3_31 - 1;
  if (ivtmp.3_36 != 0)
    goto <bb 8>;
```

1 Iteration of Epilogue
Cost Model for Peeling

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

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

Cost Model for Peeling

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

Peel Factor = 3

Cost Model for Peeling

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


Peel Factor = 3
Cost Model for Peeling

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


Peel Factor = 2
Cost Model for Peeling

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


Maximize alignment with minimal peel factor
Cost Model for Peeling

```c
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
Reducing Run Time Misalignment by Versioning

```c
int a[256], b[256];
int main (int x, int y)
{
    int i;
    for (i=0; i<200; i++)
        a[i+y] = b[i+x];
}
```
Reducing Run Time Misalignment by Versioning

```c
int a[256], b[256];
int main (int x, int y)
{
    int i;
    for (i=0; i<200; i++)
        a[i+y] = b[i+x];
}
```

version.c:5: note: Unknown alignment for access: b
version.c:5: note: Unknown alignment for access: a
Reducing Run Time Misalignment by Versioning

```c
D.2921_16 = (long unsigned int) x_5(D);
base_off_6_17 = D.2921_16 * 4;
vect_pb_7_18 = &b + base_off_6_17;
D.2924_19 = (long unsigned int) vect_pb_7_18;
D.2925_20 = D.2924_19 & 15;
D.2926_21 = D.2925_20 >> 2;
D.2927_22 = -D.2926_21;
D.2928_23 = (unsigned int) D.2927_22;
prolog_loop_niters_8_24 = D.2928_23 & 3;
D.2932_37 = prolog_loop_niters_8_24 == 0;
if (D.2932_37 != 0)
    goto <bb 6>;
else
    goto <bb 3>;
```
Reducing Run Time Misalignment by Versioning

D.2921_16 = (long unsigned int) x_5(D);
base_off.6_17 = D.2921_16 * 4;
vect_pb.7_18 = &b + base_off.6_17;
D.2924_19 = (long unsigned int) vect_pb.7_18;
D.2925_20 = D.2924_19 & 15;
D.2926_21 = D.2925_20 >> 2;
D.2927_22 = -D.2926_21;
D.2928_23 = (unsigned int) D.2927_22;
prolog_loop_niters.8_24 = D.2928_23 & 3;
D.2932_37 = prolog_loop_niters.8_24 == 0;
if (D.2932_37 != 0)
    goto <bb 6>;
else
    goto <bb 3>;

Compute address misalignment as ‘addr & (vectype_size -1)’
Reducing Run Time Misalignment by Versioning

\[
\begin{align*}
D.2921_{16} &= (\text{long unsigned int}) x_5(D); \\
\text{base\_off.}6_{17} &= D.2921_{16} \times 4; \\
\text{vect\_pb.}7_{18} &= \& b + \text{base\_off.}6_{17}; \\
D.2924_{19} &= (\text{long unsigned int}) \text{vect\_pb.}7_{18}; \\
D.2925_{20} &= D.2924_{19} \& 15; \\
D.2926_{21} &= D.2925_{20} \gg 2; \\
D.2927_{22} &= -D.2926_{21}; \\
D.2928_{23} &= (\text{unsigned int}) D.2927_{22}; \\
\text{prolog\_loop\_niters.}8_{24} &= D.2928_{23} \& 3; \\
D.2932_{37} &= \text{prolog\_loop\_niters.}8_{24} == 0; \\
\text{if (D.2932_{37} != 0)} \\
    \quad \text{goto <bb 6>}; \\
\text{else} \\
    \quad \text{goto <bb 3>};
\end{align*}
\]

Compute number of prologue iterations
Reducing Run Time Misalignment by Versioning

```
D.2921_16 = (long unsigned int) x_5(D);
base_off.6_17 = D.2921_16 * 4;
vect_pb.7_18 = &b + base_off.6_17;
D.2924_19 = (long unsigned int) vect_pb.7_18;
D.2925_20 = D.2924_19 & 15;
D.2926_21 = D.2925_20 >> 2;
D.2927_22 = -D.2926_21;
D.2928_23 = (unsigned int) D.2927_22;
prolog_loop_niters.8_24 = D.2928_23 & 3;
D.2932_37 = prolog_loop_niters.8_24 == 0;
if (D.2932_37 != 0)
    goto <bb 6>;
else
    goto <bb 3>;
```

If accesses can be aligned, go to vectorized code
Reducing Run Time Misalignment by Versioning

D.2921_16 = (long unsigned int) x_5(D);
base_off.6_17 = D.2921_16 * 4;
vect_pb.7_18 = &b + base_off.6_17;
D.2924_19 = (long unsigned int) vect_pb.7_18;
D.2925_20 = D.2924_19 & 15;
D.2926_21 = D.2925_20 >> 2;
D.2927_22 = -D.2926_21;
D.2928_23 = (unsigned int) D.2927_22;
prolog_loop_niters.8_24 = D.2928_23 & 3;
D.2932_37 = prolog_loop_niters.8_24 == 0;
if (D.2932_37 != 0)
    goto <bb 6>;
else
    goto <bb 3>;

Else go to sequential code
int a[256];
int main (int *b)
{
    int i;
    for (i=0; i<200; i++)
        *b++ = a[i];
}
Versioning for Undetermined Aliases

```c
int a[256];
int main (int *b)
{
    int i;
    for (i=0; i<200; i++)
        *b++ = a[i];
}
```

version.c:5: note: misalign = 0 bytes of ref a[i_15]
version.c:5: note: can’t force alignment of ref: *b_14
version.c:5: note: versioning for alias required: can’t determine dependence between a[i_15] and *b_14
version.c:5: note: create runtime check for data references a[i_15] and *b_14
## Versioning for Undetermined Aliases

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bb 3&gt;:</td>
<td>&lt;bb 2&gt;:</td>
</tr>
<tr>
<td># b_14 = PHI &lt;b_6, b_4(D)&gt;</td>
<td>vect_pa.6_12 = &amp;a;</td>
</tr>
<tr>
<td># i_15 = PHI &lt;i_7(4), 0(2)&gt;</td>
<td>vect_p.9_11 = b_4(D);</td>
</tr>
<tr>
<td>D.2907_5 = a[i_15];</td>
<td>D.2919_13 = vect_pa.6_12 + 16;</td>
</tr>
<tr>
<td>*b_14 = D.2907_5;</td>
<td>D.2920_8 = D.2919_13 &lt; vect_p.9_11;</td>
</tr>
<tr>
<td>b_6 = b_14 + 4;</td>
<td>D.2921_17 = vect_p.9_11 + 16;</td>
</tr>
<tr>
<td>i_7 = i_15 + 1;</td>
<td>D.2922_18 = D.2921_17 &lt; vect_pa.6_12;</td>
</tr>
<tr>
<td>if (i_7 != 200)</td>
<td>D.2923_19 = D.2920_8</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>if (D.2923_19 != 0)</td>
</tr>
<tr>
<td>else</td>
<td>goto &lt;bb 3&gt;;</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>else</td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td>goto &lt;bb 6&gt;;</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td></td>
</tr>
</tbody>
</table>

Check for dependence within VF
### Versioning for Undetermined Aliases

#### Control Flow Graph

```c
<bb 3>:
    # b_14 = PHI <b_6, b_4(D)>
    # i_15 = PHI <i_7(4), 0(2)>
    D.2907_5 = a[i_15];
    *b_14 = D.2907_5;
    b_6 = b_14 + 4;
    i_7 = i_15 + 1;
    if (i_7 != 200)
        goto <bb 4>;
    else
        goto <bb 5>;
<bb 4>:
    goto <bb 3>;
```

#### Vectorized Control Flow Graph

```c
<bb 3>:
    #vect_pa.10_30 = PHI <vect_pa.10_31, vect_pa.13_29>
    #vect_p.15_34 = PHI <vect_p.15_35, vect_p.18_33>
    #ivtmp.19_36 = PHI <ivtmp.19_37, 0>
    vect_var_.14_32 = MEM[vect_pa.10_30]
    MEM[vect_p.15_34] = vect_var_.14_32;
    vect_pa.10_31 = vect_pa.10_30 + 16;
    vect_p.15_35 = vect_p.15_34 + 16;
    ivtmp.19_37 = ivtmp.19_36 + 1;
    if (ivtmp.19_37 < 50)
        goto <bb 3>;
    else
        goto <bb 9>;
```

Execute vector code if no aliases within VF
### Versioning for Undetermined Aliases

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Vectorized Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;bb 3&gt;</code>:</td>
<td><code>&lt;bb 6&gt;</code>:</td>
</tr>
<tr>
<td># b_14 = PHI &lt;b_6, b_4(D)&gt;</td>
<td># b_20 = PHI &lt;b_4(D)(6), b_26(8)&gt;</td>
</tr>
<tr>
<td># i_15 = PHI &lt;i_7(4), 0(2)&gt;</td>
<td># i_21 = PHI &lt;0(6), i_27(8)&gt;</td>
</tr>
<tr>
<td>D.2907_5 = a[i_15];</td>
<td>D.2907_24 = a[i_21];</td>
</tr>
<tr>
<td>*b_14 = D.2907_5;</td>
<td>*b_20 = D.2907_24;</td>
</tr>
<tr>
<td>b_6 = b_14 + 4;</td>
<td>b_26 = b_20 + 4;</td>
</tr>
<tr>
<td>i_7 = i_15 + 1;</td>
<td>i_27 = i_21 + 1;</td>
</tr>
<tr>
<td>if (i_7 != 200)</td>
<td>ivtmp.3_23 = ivtmp.3_28 - 1;</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>if (ivtmp.3_28 != 0)</td>
</tr>
<tr>
<td>else</td>
<td>goto &lt;bb 6&gt;;</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>else</td>
</tr>
<tr>
<td><code>&lt;bb 4&gt;</code>:</td>
<td>goto &lt;bb 9&gt;;</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td></td>
</tr>
</tbody>
</table>

Execute scalar code if aliases are within VF
Profitability of Vectorization

```c
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<50; i++)
        a[i] = b[i*4];
}
```
Profitability of Vectorization

```c
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<50; i++)
        a[i] = b[i*4];
}
```

```
vec.c:5: note: cost model: the vector iteration cost = 10 divided by the scalar iteration cost = 2 is greater or equal to the vectorization factor = 4.

vec.c:5: note: not vectorized: vectorization not profitable.
```
Profitability of Vectorization

```c
short int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<50; i++)
        a[i] = b[i*4];
}
```

Vectorization Factor = 8
VF x scalar iteration cost > vector iteration cost
Profitability of Vectorization

```c
short int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<50; i++)
        a[i] = b[i*4];
}
```

Vectorization Factor = 8
VF x scalar iteration cost > vector iteration cost

`vec.c:5: note: LOOP VECTORIZED.`
`vec.c:2: note: vectorized 1 loops in function.`
Cost Model of Vectorizer

Vectorization is profitable when

\[ SIC \times nites + SOC > VIC \times \left( \frac{nites - PL\_ITERS - EP\_ITERS}{VF} \right) + VOC \]

\( SIC \) = 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
int main (int *a, int *b) 
{
    int i, n;
    for (i=0; i<n; i++)
        *a++ = *b--;
}
int main (int *a, int *b)
{
    int i, n;
    for (i=0; i<n; i++)
        *a++ = *b--;
}

vec.c:4: note: versioning for alias required: can’t
determine dependence between *b_19 and *a_18

vec.c:4: note: Cost model analysis:
  Vector inside of loop cost: 4
  Vector outside of loop cost: 14
  Scalar iteration cost: 2
  Scalar outside cost: 1
  prologue iterations: 0
  epilogue iterations: 2
  Calculated minimum iters for profitability: 12
Cost Model of Vectorizer

```c
int main (int * restrict a, int * restrict b) {
    int i, n;
    for (i=0; i<n; i++)
        *a++ = *b--;
}
```
Cost Model of Vectorizer

```c
int main (int * restrict a, int * restrict b) {
    int i, n;
    for (i=0; i<n; i++)
        *a++ = *b--;
}
```

`vec.c:4: note: Cost model analysis:
  Vector inside of loop cost: 3
  Vector outside of loop cost: 16
  Scalar iteration cost: 2
  Scalar outside cost: 7
  prologue iterations: 2
  epilogue iterations: 2
  Calculated minimum iters for profitability: 5`
int a[500];
int main()
{
    int i;
    for (i=0; i<350; i++)
        a[i] = a[i] + 2;
}

Compile with:
gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4
Cost Model of Parallelizer

```c
int a[500];
int main ()
{
    int i;
    for (i=0; i<350; i++)
        a[i] = a[i] + 2;
}
```

Compile with:
```
gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4
```

Loop not parallelized as number of iterations per thread $\leq 100$
int a[500];
int main ()
{
    int i;
    for (i=0; i<350; i++)
        a[i] = a[i] + 2;
}

Compile with:
gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=3
Cost Model of Parallelizer

```c
int a[500];
int main ()
{
    int i;
    for (i=0; i<350; i++)
        a[i] = a[i] + 2;
}
```

Compile with:
```
gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=3
```

SUCCESS: may be parallelized
Cost Model of Parallelizer

Inner Parallelism

```c
int i, j;
for (i=0; i<450; i++)
    for (j=0; j<420; j++)
        a[i][j] = a[i-1][j];
```

Compile with:
```
 gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4
```
Cost Model of Parallelizer

Inner Parallelism

```c
int i, j;
for (i=0; i<450; i++)
    for (j=0; j<420; j++)
        a[i][j] = a[i-1][j];
```

Compile with:
gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4

distance_vector:   1  0
direction_vector:   +  =
FAILED: data dependencies exist across iterations
Cost Model of Parallelizer

Outer Parallelism

```c
int i, j;
for (j=0; j<420; j++)
    for (i=0; i<450; i++)
        a[i][j] = a[i-1][j];
```

Compile with:
```
gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4
```
Cost Model of Parallelizer

Outer Parallelism

```c
int i, j;
for (j=0; j<420; j++)
    for (i=0; i<450; i++)
        a[i][j] = a[i-1][j];
```

Compile with:
```
gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4
```

distance_vector: 0 1
direction_vector: = +
SUCCESS: may be parallelized
Cost Model of Parallelizer

D.2000_5 = __builtin_omp_get_num_threads ()
D.2001_6 = (unsigned int) D.2000_5
D.2002_7 = __builtin_omp_get_thread_num ()
D.2003_8 = (unsigned int) D.2002_7
D.2004_9 = 419 / D.2001_6
D.2006_11 = D.2005_10 != 419
ivtmp.7_13 = D.2007_12 * D.2003_8
D.2009_14 = ivtmp.7_13 + D.2007_12
D.2010_15 = MIN_EXPR <D.2009_14, 419>
if (ivtmp.7_13 >= D.2010_15)
goto <bb 3>;
Cost Model of Parallelizer

```c
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
    goto <bb 3>;
```

Get the number of threads
Cost Model of Parallelizer

```c
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
    goto <bb 3>;
```

Get thread identity
Cost Model of Parallelizer

```c
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
    goto <bb 3>;
```

Perform load calculations
Cost Model of Parallelizer

D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
    goto <bb 3>;

Assign start iteration to the chosen thread
Cost Model of Parallelizer

D.2000_5 = __builtin_omp_get_num_threads();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
    goto <bb 3>;

Assign end iteration to the chosen thread
Cost Model of Parallelizer

D.2000_5 = __builtin_omp_get_num_threads();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
  goto <bb 3>;

Start execution of iterations of the chosen thread
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