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 ⇒ 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 ≡ a vector of 2×(32 bits), 4×(16 bits), or 8×(8 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];
```

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];
```

Vectorized Code

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

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

Notes

Observe reads and writes into a given location
Example 1

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

Original Code | Parallelized Code
---|---
```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```
```
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 #

Example 1: The Moral of the Story

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>

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++)
```

• Vector instruction is synchronized: All reads before writes in a given instruction
• Read-writes across multiple instructions executing in parallel may not be synchronized

Notes

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>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>

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.
**Example 3**

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

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration i</td>
<td>Read</td>
</tr>
<tr>
<td>Iteration i + k</td>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
<td>Yes</td>
</tr>
<tr>
<td>Write</td>
<td>No</td>
</tr>
<tr>
<td>Write</td>
<td>No</td>
</tr>
<tr>
<td>Read</td>
<td>Does not matter</td>
</tr>
</tbody>
</table>

\[
\text{int } A[N], B[N], i; \quad \text{for } (i=0; i<N; i++) \\
A[i+1] = A[i] + B[i+1];
\]

---

**Example 4**

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

1. This case is not possible
2. Vectorization is a limited granularity parallelization
3. 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

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**
  ![Dependence in the same iteration](attachment:dependence_graph1.png)

- **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**
  ![Dependence due to the outermost loop](attachment:dependence_graph2.png)

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

- Program

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

- Dependence graph

```
S1 δ1
```

- Loop carried flow-dependence
  Neither parallelization not vectorization is possible

Notes:

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 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

<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>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>2.2</td>
<td>2.2</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
</tr>
<tr>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
</tr>
<tr>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td>3.3</td>
<td>3.3</td>
<td>3.3</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Example 4: Dependence

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

Program to swap arrays

Dependence Graph:

Notes
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 */
}
```

Loop independent anti dependence due to A[i]

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
**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 flow dependence due to T

- 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
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++) {
    S_1: C[i] = A[i+2];
    S_2: A[i] = B[i];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S_1: C[i:i+3] = A[i+2:i+5];
    S_2: A[i:i+3] = B[i:i+3];
}
```
Anti Dependence and Vectorization

Write precedes Read lexicographically

```
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];
}
```

True Dependence and Vectorization

Write precedes Read lexicographically

```
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];
}
```
Multiple Dependences and Vectorization

Anti Dependence and True Dependence

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

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

```
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];
}
```

```
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];
}
```

```
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++) {
    \textcolor{red}{L_1: A[i+5] = A[i];}
}
```

```c
int A[N], i, temp;
for (i=0; i<N; i++) {
    \textcolor{red}{S_1: temp = A[i];}
    \textcolor{red}{S_2: A[i+5] = temp;}
}
```

```c
int A[N], T[N], i;
for (i=0; i<N; i+=4) {
    \textcolor{red}{S_1: T[i:i+3] = A[i:i+3];}
    \textcolor{red}{S_2: A[i+5:i+8] = T[i:i+3];}
}
```

```c
int A[N], T[N], i;
for (i=0; i<N; i++) {
    \textcolor{red}{S_1: T[i] = A[i];}
    \textcolor{red}{S_2: 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++) {
    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];
}
```

- 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+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];
}
```

- Rescheduling of statements will not break the cyclic dependence
- The dependence distance from \(S_2\) to \(S_1\) ≥ 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.

Feasibility of Parallelization

**Outer Parallel**

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

Outer Parallel

```
for (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

\[
\text{for } (i=2; i<n; i++) \\
\text{for-all } (j=1 \text{ to } n) \\
A[i][j] = A[i-1][j];
\]
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

- 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.)
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)

\[
\text{for } (i=3; i<15; i=i+3) \\
\quad \text{for } (j=11; j>1; j=j-2) \\
\quad \{ \\
\quad \quad A[i+1][2*j-1] = \ldots \\
\quad \} \\
\}
\]

<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>
<tbody>
<tr>
<td>int a[200];</td>
<td></td>
</tr>
<tr>
<td>int main()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int i;</td>
<td></td>
</tr>
<tr>
<td>for (i=0; i&lt;150; i++)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>a[i] = a[i+1] + 2;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Example 1: Observing Data Dependence
Notes

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>;
<bb 5>:
goto <bb 3>;}
Step 2: Understanding the chain of recurrences

```
<bb 3>:
    # i_3 = PHI <i_3(4), 0(2)>
    i_3 = i_3 + 1;
    D_1955_4 = a[i_3];
    D_1956_5 = D_1955_4 + 2;
    a[i_3] = D_1956_5;
    if (i_3 != 150)
        goto <bb 4>;
    else
        goto <bb 5>;
<bb 4>:
    goto <bb 3>;
```

Notes
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>;
```

Notes

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 1: Observing Data Dependence

Step 3: Observing the data dependence information

\[ \text{iterations that access an element twice in } A: [1 + 1 \times 1] \]
\[ \text{last conflict: 149} \]
\[ \text{iterations that access an element twice in } B: [0 + 1 \times 1] \]
\[ \text{last conflict: 149} \]
Subscript distance: 1

inner loop index: 0
loop nest: (1)
distance \text{vector}: 1
direction \text{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
  - `--fthread-limit=2` `-fdump-tree-parloops=all`
- Additional options for vectorization
  - `--fdump-tree-vect=all` `--msse4` `--fthread-limit=2`
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></td>
</tr>
<tr>
<td>int i;</td>
<td></td>
</tr>
<tr>
<td>for (i=0; i&lt;256; i++) {</td>
<td></td>
</tr>
<tr>
<td>a[i] = b[i];</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;bb 3&gt;:</td>
</tr>
<tr>
<td></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></td>
<td>a[i_11] = D.2836_3;</td>
</tr>
<tr>
<td></td>
<td>i_4 = i_11 + 1;</td>
</tr>
<tr>
<td></td>
<td>if (i_4 != 256)</td>
</tr>
<tr>
<td></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>

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>&lt;bb 3&gt;:</td>
<td>&lt;bb 2&gt;:</td>
</tr>
<tr>
<td># i₁₁ = PHI &lt;i₄(4), 0(2)&gt;</td>
<td>vect_pb.7₁₀ = &amp;b;</td>
</tr>
<tr>
<td>D.2836₃ = b[i₁₁];</td>
<td>vect_pb.1₂₁₅ = &amp;a;</td>
</tr>
<tr>
<td>a[i₁₁] = D.2836₃;</td>
<td></td>
</tr>
<tr>
<td>i₄ = i₁₁ + 1;</td>
<td></td>
</tr>
<tr>
<td>if (i₄ != 256) goto &lt;bb 4&gt;</td>
<td># vect_pb.4₁₆ = PHI &lt;vect_pb.4₁₃,</td>
</tr>
<tr>
<td></td>
<td>vect_pb.7₁₀&gt;</td>
</tr>
<tr>
<td>else</td>
<td>if (i₄ = 256) goto &lt;bb 3&gt;;</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td></td>
</tr>
<tr>
<td>&lt;bb 4&gt;:</td>
<td>vect_var.8₁₄ = MEM[vect_pb.4₁₆];</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td>MEM[vect_pb.9₁₆] = vect_var.8₁₄;</td>
</tr>
<tr>
<td></td>
<td>vect_pb.4₁₃ = vect_pb.4₁₆ + 16;</td>
</tr>
<tr>
<td></td>
<td>vect_pb.9₁₇ = vect_pb.9₁₆ + 16;</td>
</tr>
<tr>
<td></td>
<td>ivtmp.1₃₁₉ = ivtmp.1₃₁₈ + 1;</td>
</tr>
<tr>
<td></td>
<td>if (ivtmp.1₃₁₉ &lt; 64) goto &lt;bb 4&gt;;</td>
</tr>
</tbody>
</table>

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

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 $t_i$ 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 $t_i$

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

- All threads are executed in parallel
Example 2: Observing Vectorization and Parallelization

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

D.1996 = __builtin_omp_get_num_threads ();
D.1998 = __builtin_omp_get_thread_num ();
D.2000 = 255 / D.1997;
D.2002 = D.2001 != 255;
ivtmp.7 = D.2003 * D.1999;
D.2005 = ivtmp.7 + D.2003;
D.2006 = MIN_EXPR <D.2005, 255>;
if (ivtmp.7 >= D.2006)
goto <bb 3>;

Notes

Get the number of threads
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>;
```
Step 5: Examining the thread creation in parallelized control flow graph

D.1996 = __builtin_omp_get_num_threads ();
D.1998 = __builtin_omp_get_thread_num ();
D.2000.10 = 255 / D.1997.6;
ivtmp.7.14 = D.2003.13 * D.1999.8;
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

D.1996 = __builtin_omp_get_num_threads ();
D.1998 = __builtin_omp_get_thread_num ();
D.2000.10 = 255 / D.1997.6;
ivtmp.7.14 = D.2003.13 * D.1999.8;
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 = MINEXPR <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

```
D.1956_3 = b[i_11];
a[i_11] = D.1956_3;
i_4 = i_11 + 1;
if (i_4 != 256)
goto <bb 4>;
else
goto <bb 5>;
if (D.2006_16 > ivtmp.7_19)
goto <bb 5>;
else
goto <bb 3>;
```

Control Flow Graph | Parallel loop body
--- | ---
<bb 3>: | <bb 5>:
# i_11 = PHI <i_4(4), 0(2)> | i.8_21 = (int) ivtmp.7_18;
D.1956_3 = b[i_11]; | D.2010_23 = *b.10_4[i.8_21];
a[i_11] = D.1956_3; | *a.11_5[i.8_21] = D.2010_23;
i_4 = i_11 + 1; | ivtmp.7_19 = ivtmp.7_18 + 1;
if (i_4 != 256) | if (D.2006_16 > ivtmp.7_19)
goto <bb 4>;
else | goto <bb 5>;
goto <bb 5>;
goto <bb 3>;
Example 3: Vectorization but No Parallelization

Step 0: Compiling with

```
-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

Control Flow Graph

\[
\begin{align*}
\text{bb 3:} & \\
& \# i_{12} = \text{PHI} \langle i_{5}(4), 0(2)\rangle \\
& \text{D.2834} = i_{12} + 4; \\
& \text{D.2835} = \text{a}[\text{D.2834}]; \\
& \text{a}[i_{12}] = \text{D.2835}; \\
& i_{5} = i_{12} + 1; \\
& \text{if (i_{5} != 619)} \\
& \text{goto <bb 4>}; \\
& \text{else} \\
& \text{goto <bb 5>}; \\
\text{bb 4:} & \\
& \text{goto <bb 3>};
\end{align*}
\]

Vectorized Control Flow Graph

\[
\begin{align*}
\text{bb 2:} & \\
& \text{vect_pa.10}_{26} = \&\text{a}[4]; \\
& \text{vect_pa.15}_{30} = \&\text{a}; \\
\text{bb 3:} & \\
& \# \text{vect}_{\text{pa.7}}_{27} = \text{PHI} \langle \text{vect}_{\text{pa.7}}_{28}, \\
& \text{vect}_{\text{pa.10}}_{26}\rangle \\
& \# \text{vect}_{\text{pa.12}}_{31} = \text{PHI} \langle \text{vect}_{\text{pa.12}}_{32}, \\
& \text{vect}_{\text{pa.15}}_{30}\rangle \\
& \text{vect}_{\text{var.11}}_{29} = \text{MEM}[\text{vect}_{\text{pa.7}}_{27}]; \\
& \text{MEM}[\text{vect}_{\text{pa.12}}_{31}] = \text{vect}_{\text{var.11}}_{29}; \\
& \text{vect}_{\text{pa.7}}_{28} = \text{vect}_{\text{pa.7}}_{27} + 16; \\
& \text{vect}_{\text{pa.12}}_{32} = \text{vect}_{\text{pa.12}}_{31} + 16; \\
& \text{ivtmp}_{16}_{34} = \text{ivtmp}_{16}_{33} + 1; \\
& \text{if (ivtmp}_{16}_{34} < 154) \\
& \text{goto <bb 4>};
\end{align*}
\]

• 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`

Notes

• 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

\[
\begin{align*}
    a[i+2] &= b[i] + 5 \\
    b[i+1] &= a[i] + 10
\end{align*}
\]
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];
}
```

- 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 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 for Parallelization

**Original Code**

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

**After Interchange**

```c
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++) {
    S_1 : a[i+3] = a[i];
    S_2 : 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>goto &lt;bb 4&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 5&gt;;</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td>&lt;bb 8&gt;:</td>
</tr>
</tbody>
</table>

**Notes**

- 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`

**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

Peeling for Vectorization

Original Code

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

- Cyclic Dependence, dependence distance for *backward* dependence
  \(= 3 < VF\)
- Cannot vectorize

Transformed Code

```
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 for Parallelization

**Original Code**

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

- Dependence on i, cannot be parallelized
- Total number of synchronizations required = \( n \)

**Transformed Code**

```
c[1] = a[0];
for (i=1; i<n-1; i++)
{
  S1: a[i] = b[i];
  S2: c[i+1] = a[i];
}
```

- Outer Loop parallelizable
- Total number of synchronizations required = 1
Part 4

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--;
    }
}
```

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];
}
```

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: niter for prologue loop: 2
peel.c:5: note: Cost model analysis:
  prologue iterations: 2
  epilogue iterations: 1

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

Control Flow Graph

Vectorized Control Flow Graph

<bb 3>:
# i.12 = PHI <i.6, 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)
go to <bb 4>;
else
    goto <bb 5>;
<bb 4>:
go to <bb 3>;

<bb 3>:
# ivtmp.27 = PHI <ivtmp.28, 0(2)>
D.2908_16 = i.7 + 2;
D.2909_17 = b[D.2908_16];
a[D.2908_16] = D.2909_17;
ivtmp.28 = ivtmp.27 + 1;
if (ivtmp.28 < 2)
go to <bb 3>;
else
    goto <bb 5>;

Notes

2 Iterations of Prologue

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

200 Iterations of Vector Code
Reducing Compile Time Misalignment by Peeling

Control Flow Graph

Vectorized Control Flow Graph

```
<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>;
```

```
<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

```
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  Peel Factor = 3  Peel Factor = 2
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

Peel the loop by 3
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];
}

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

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 number of prologue iterations

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

```c
D.2921_16 = (long unsigned int) x_5(b(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.

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

Control Flow Graph

Vectorized Control Flow Graph

<bb 3>:
  # b₁₄ = PHI <b₆, b₄(D)>
  # i₁₅ = PHI <i₇(4), 0(2)>
  D.2907₅ = a[i₁₅];
  *b₁₄ = D.2907₅;
  b₆ = b₁₄ + 4;
  i₇ = i₁₅ + 1;
  if (i₇ != 200)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;

<bb 5>:
  goto <bb 3>;

Notes:

Check for dependence withinVF

Execute vector code if no aliases withinVF
### 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>if (i.27 != 200)</td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>goto &lt;bb 4&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><code>&lt;bb 4&gt;</code>;</td>
<td><code>&lt;bb 4&gt;</code>;</td>
</tr>
<tr>
<td>goto &lt;bb 3&gt;;</td>
<td>goto &lt;bb 3&gt;;</td>
</tr>
</tbody>
</table>

### 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 × 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

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

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
Cost Model of Vectorizer

```c
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`

Notes

```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

Loop not parallelized as number of iterations per thread ≤ 100

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

```
distance_vector:  0  1
direction_vector: =  +
SUCCESS: may be parallelized
```
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>;
```
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.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

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.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

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
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