Workshop on Essential Abstractions in GCC

Introduction to Parallelization and Vectorization

GCC Resource Center
(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

3 July 2011

Outline

- Transformation for parallel and vector execution
- Data dependence
The Scope of this Tutorial

- What this tutorial does not address
  - Algorithms used for parallelization and vectorization
  - Code or data structures of the parallelization and vectorization pass of GCC
  - Machine level issues related to parallelization and vectorization
- What this tutorial addresses

Basics of Discovering Parallelism using GCC

Part 1

Transformations for Parallel and Vector Execution
### A Taxonomy of Parallel Computation

<table>
<thead>
<tr>
<th></th>
<th>Single Program</th>
<th>Multiple Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Data</strong></td>
<td>SISD</td>
<td>MPSD</td>
</tr>
<tr>
<td><strong>Multiple Data</strong></td>
<td>SPMD</td>
<td>MPMD</td>
</tr>
</tbody>
</table>

### Notes

Redundant computation for validation of intermediate steps
### A Taxonomy of Parallel Computation

<table>
<thead>
<tr>
<th></th>
<th>Single Program</th>
<th>Multiple Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Instruction</td>
<td>Single Data</td>
<td>SISD</td>
</tr>
<tr>
<td>Multiple Instructions</td>
<td>Multiple Data</td>
<td>SIMD, MISD, MPSD</td>
</tr>
<tr>
<td>Single Data</td>
<td>Single Instruction</td>
<td>SISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>Multiple Instruction</td>
<td>SIMD, MIMD, MPMD</td>
</tr>
</tbody>
</table>

**Redundant computation for validation of intermediate steps**

**Transformations performed by a compiler**
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 $2 \times (32 \text{ bits}), 4 \times (16 \text{ bits}), \text{ or } 8 \times (8 \text{ bits})$

**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];
```

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 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

Vectorized Code

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

Notes
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];

Parallelized Code
int A[N], B[N], i;
foreach (i=1; i<N; )
    A[i] = A[i] + B[i-1];

Notes

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>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>

Notes
Example 2

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

Original Code

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

Notes

- 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

Table: 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>Yes</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>No</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td>No</td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td>Does not matter</td>
</tr>
</tbody>
</table>

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

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.
Example 3

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

int A[N], B[N];
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</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>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]

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

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
- Blank line to increase height
- Blank line to increase height
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>$S_i \delta S_j$</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 prohibits parallel execution of $S_i$ and $S_j$

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

- No loop carried dependence
  Both vectorization and parallelization are possible

Notes

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

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

Notes

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
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 not 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;
  }
```

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

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:

[Diagram of dependence graph with nodes S1, S2, S3, and arrows indicating 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

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

```
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]

Notes
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 flow dependence due to T

Program to swap arrays

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

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 $T$
Draw the dependence graph for the following program (Earlier program modified to swap 2-dimensional arrays)

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

**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 lexicographically precedes Write

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

Anti Dependence and Vectorization

Write lexicographically precedes Read

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

Write lexicographically precedes Read

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

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

Conjunction of Dependences and Vectorization

Anti Dependence and True Dependence

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

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

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

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

**True Dependence and Anti Dependence**

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

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

```
int A[N], B[N], i;
for (i=0; i<N; i=i+4) {
    B[i+2:i+5] = A[i+1:i+4];
    A[i:i+3] = B[i:i+3];
}
```

**Notes**

Rescheduling of statements will not break the cyclic dependency - cannot vectorize
Last but not the least ...