Programming of the Pocket calculators for continued education (18 hours, 1 credit)

Instructor: Prof. Dr. Ing. habil. Gert-Michael Leue, Gerhard Scholz GmbH

The course is designed for university lecturers and instructors who want to

- program pocket calculators for additional classes.
- improve professional skills in programming pocket calculators.
- learn about the latest technological developments in this area.

The course will cover the following topics:

1. Introduction to Pocket Calculators
2. Programming Basics
3. Advanced Programming Techniques
4. Real-World Applications
5. Future Trends

The instructor will provide hands-on training and real-life examples to help participants develop their skills.

Date: [insert date]
Time: [insert time]
Location: [insert location]

Target Audience: University lecturers and instructors

Cost: [insert cost]

Enrollment: [insert enrollment information]
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 $\Rightarrow$ 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

Vectorized Code

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

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];
```

Parallelized Code

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

<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>

**Example 2**

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

```
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

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

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ 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>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td>Yes</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>Does not matter</td>
<td></td>
</tr>
</tbody>
</table>

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

Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ 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>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>Yes</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>Does not matter</td>
<td></td>
</tr>
</tbody>
</table>

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

Example 4

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ 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 \uparrow 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

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](image)

- 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](image)

- 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 not vectorization is possible

Iteration Vectors and Index Vectors: Example 1

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

<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>0.0</td>
</tr>
<tr>
<td>0.1</td>
<td>1.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>0.2</td>
<td>1.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>0.3</td>
<td>1.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.1</td>
<td>3.1</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>1.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td>1.3</td>
<td>3.3</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>2.0</td>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2.1</td>
<td>3.1</td>
<td>4.1</td>
<td>4.1</td>
</tr>
<tr>
<td>2.2</td>
<td>3.2</td>
<td>4.2</td>
<td>4.2</td>
</tr>
<tr>
<td>2.3</td>
<td>3.3</td>
<td>4.3</td>
<td>4.3</td>
</tr>
</tbody>
</table>

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

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Iteration Vectors and Index Vectors: Example 2

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

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

<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>0.0</td>
</tr>
<tr>
<td>0.1</td>
<td>1.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>0.2</td>
<td>1.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>0.3</td>
<td>1.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.1</td>
<td>3.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1.2</td>
<td>3.1</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>1.3</td>
<td>3.2</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>2.0</td>
<td>3.1</td>
<td>4.1</td>
<td>4.1</td>
</tr>
<tr>
<td>2.1</td>
<td>3.2</td>
<td>4.2</td>
<td>4.2</td>
</tr>
<tr>
<td>2.2</td>
<td>3.3</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>2.3</td>
<td>3.4</td>
<td>5.1</td>
<td>5.1</td>
</tr>
<tr>
<td>3.0</td>
<td>4.0</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>3.1</td>
<td>4.1</td>
<td>6.1</td>
<td>6.1</td>
</tr>
<tr>
<td>3.2</td>
<td>4.2</td>
<td>6.2</td>
<td>6.2</td>
</tr>
<tr>
<td>3.3</td>
<td>4.3</td>
<td>6.3</td>
<td>6.3</td>
</tr>
</tbody>
</table>

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

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

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

Conclusion: No dependence
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 $A[i]$

```
S1 -----> S2
\   /   \   / \\
  \ /     \ /   \ /  \\
   \        S3 ---->
```

Loop independent flow dependence due to $T$

```
S1 -----> S2
\   /   \   / \\
  \ /     \ /   \ /  \\
   \        S3 ---->
```

Loop carried anti 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

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++) {
    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];
}
```

```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];
    S1: A[i] = B[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++) {
    S1: A[i+2] = C[i];
    S2: B[i] = A[i];
    S1: A[i+2] = C[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++) {
    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];
}
```

```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];
}
```
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++) {
    S1: A[i+5] = A[i];
}
```

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

```c
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++) {
    S1: B[i+2] = A[i];
    S2: A[i+1] = B[i+1];
}
```

**Cyclic Anti Dependence**

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

**Cyclic Anti Dependence**

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

- 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.

Feasibility of Parallelization

Outer Parallel

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

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

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

Part 2

The Lambda Framework

- 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.)
Loop Transformation Passes in GCC: Our Focus

<table>
<thead>
<tr>
<th>Data Dependence</th>
<th>Pass variable name</th>
<th>pass_check_data_deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabling switch</td>
<td>-fcheck-data-deps</td>
<td></td>
</tr>
<tr>
<td>Dump switch</td>
<td>-fdump-tree-ckdd</td>
<td></td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.ckdd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop Distribution</th>
<th>Pass variable name</th>
<th>pass_loop_distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabling switch</td>
<td>-ftree-loop-distribution</td>
<td></td>
</tr>
<tr>
<td>Dump switch</td>
<td>-fdump-tree-idist</td>
<td></td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.idist</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vectorization</th>
<th>Pass variable name</th>
<th>pass_vectorize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabling switch</td>
<td>-ftree-vectorize</td>
<td></td>
</tr>
<tr>
<td>Dump switch</td>
<td>-fdump-tree-vect</td>
<td></td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.vect</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallelization</th>
<th>Pass variable name</th>
<th>pass_parallelize_loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabling switch</td>
<td>-ftree-parallelize-loops=n</td>
<td></td>
</tr>
<tr>
<td>Dump switch</td>
<td>-fdump-tree-parloops</td>
<td></td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.parloops</td>
<td></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] = ...
    }
}
```

Example 1: Observing Data Dependence

Step 0: Compiling

```
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]; int main()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int i; 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>

**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>;
  goto <bb 3>;
```

Essential Abstractions in GCC

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

Step 2: Understanding the chain of recurrences

```c
<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>;
```

```
# i_3 = PHI i_3(4), 0(2)
base_address: &a
offset from base address: 0
constant offset from base address: 0
aligned to: 128
(chrec = {0, +, 1}_a)
```

```c
<bb 5>:
  goto <bb 3>;
```

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

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]; int main() { int i; for (i=0; i&lt;256; i++) { a[i] = b[i]; } return 0; }</td>
<td>&lt;bb 3&gt;: # _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;; &lt;bb 4&gt;: 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.

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

Step 4: Understanding the strategy of parallel execution

- Create threads t_i 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 t_i:
  for (j=start_for_thread_i; j<=end_for_thread_i; j++) {
    /* execute the loop body to be parallelized */
  }
- All threads are executed in parallel
Example 2: Observing Vectorization and Parallelization

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

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.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

```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

```
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

D.1996 = __builtin_omp_get_num_threads ();
D.1998 = __builtin_omp_get_thread_num ();
D.2000 = 255 / D.1996;
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>;

Assign start iteration to the chosen thread

Start execution of iterations of the chosen thread

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

Control Flow Graph

Parallel loop body

<i>&lt;&lt;{bb 3};&gt;&lt;</i>

# i = PHI <i>{a[4]}, 0(2>&gt;
D.1956 = b[i];
a[i] = D.1956;
if (i != 256)
goto &lt;bb 4&gt;:
else
    goto &lt;bb 5&gt;:

<i>&lt;&lt;{bb 5};&gt;&lt;</i>

i.8 = (int) ivtmp.7;
D.2010 = *b.10 + 4;
D.2010 > ivtmp.7 + 1;
goto &lt;bb 5&gt;:
else
    goto &lt;bb 3&gt;:

if (ivtmp.7 >= D.2006)
goto &lt;bb 3&gt;;
Example 3: Vectorization but No Parallelization

Step 0: Compiling with

```
-02 -fdump-tree-vect-all -msse4 -ftree-vectorize
```

```
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.
```

Step 2: Examining vectorization

```
Control Flow Graph

Vectorized Control Flow Graph

<bb 3>:
    # i.12 = PHI <i.5(4), 0(2)>
    D.2834.3 = i.12 + 4;
    D.2835.4 = a[D.2834.3];
    a[i.12] = D.2835.4;
    i.5 = i.12 + 1;
    if (i.5 != 619)
        goto <bb 4>;
    else
        goto <bb 5>;
<bb 4>:
    goto <bb 3>;
```

```
<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>;
```

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`

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`

Step 3: Understanding the dependences that prohibit vectorization and parallelization

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

δ₁

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 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++) {
    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_i = i,13 + 3;</td>
<td>D.2692_i = i,11 + 3;</td>
</tr>
<tr>
<td>D.2693_a = a[i,13];</td>
<td>D.2693_a = a[i,11];</td>
</tr>
<tr>
<td>a[D.2692_a] = D.2693_a;</td>
<td>a[D.2692_i] = D.2693_i;</td>
</tr>
<tr>
<td>D.2694_b = c[i,13];</td>
<td>D.2694_b = c[i,11];</td>
</tr>
<tr>
<td>b[i,13] = D.2694_b;</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></td>
</tr>
<tr>
<td>goto &lt;bb 4&gt;;</td>
<td>else</td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td>goto &lt;bb 3&gt;;</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>D.2694_b = c[i,13];</td>
<td>D.2694_b = c[i,11];</td>
</tr>
<tr>
<td>b[i,13] = D.2694_b;</td>
<td>if (i,6) != 230</td>
</tr>
<tr>
<td>i,6 = i,13 + 1;</td>
<td>goto &lt;bb 8&gt;;</td>
</tr>
<tr>
<td>if (i,6) != 230</td>
<td></td>
</tr>
<tr>
<td>goto &lt;bb 5&gt;;</td>
<td></td>
</tr>
</tbody>
</table>

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

Original Code
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
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

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++)
  S1: 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++)
{
  \textcolor{red}{S_1}: \ a[i] = b[i];
  \textcolor{blue}{S_2}: \ 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++) {
  \textcolor{red}{S_1}: \ a[i] = b[i];
  \textcolor{blue}{S_2}: \ c[i+1] = a[i];
}
```

- Outer Loop parallelizable

Total number of synchronizations required = \( 1 \)

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.28364]
peel.c:5: note: misalign = 8 bytes of ref a[D.28364]

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
---|---
<br>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;
if (i,6 != 203)
goto <bb 4>;
else
  goto <bb 5>;
<br>4>:  goto <bb 3>;

2 Iterations of Prologue

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Cost Model for Peeling

```
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
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
```

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

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

```c
D.2921_16 = (long unsigned int) x_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_27 = prolog_loop_niters.8_24 == 0;
if (D.2932_27 != 0)
  goto <bb 6>;
else
  goto <bb 3>;
```

Compute number of prologue iterations

Else go to sequential code

If accesses can be aligned, go to vectorized code

Versioning for Undetermined Aliases

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

### Versioning for Undetermined Aliases

**Control Flow Graph**

```
<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>;
```

**Control Flow Graph**

```
<bb 2>:
  vect_pa.6_12 = &a;
  vect_p.9_11 = b_4(D);
  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>;
```

### Check for dependence within VF

**Versioning for Undetermined Aliases**

```
<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>;
```

**Versioning for Undetermined Aliases**

```
<bb 6>:
  # b_20 = PHI <b_6, b_4(D)>
  # i_21 = PHI <0(6), i_27(8)>
  ivtmp.3_23 = PHI <200, ivtmp.3_28>
  D.2907_24 = a[i_21];
  *b_20 = D.2907_24;
  b_26 = b_20 + 4;
  i_27 = i_21 + 1;
  ivtmp.3_28 = ivtmp.3_23 - 1;
  if (ivtmp.3_28 != 0)
    goto <bb 6>;
  else
    goto <bb 9>;
```

### Excute vector code if no aliases within VF

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

**Profitability of Vectorization**

```
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

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

Cost Model of Vectorizer

```
SIC * nitors + SOC > VIC * (nitors - PL_ITERS - EP_ITERS) / VF + 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

```
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

```
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
```
**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 ≤ 100

---

**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
```

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

D.2005 = __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

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Get thread identity

Perform load calculations
Cost Model of Parallelizer

Assign start iteration to the chosen thread

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