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

Outline

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

Part 1

Parallelization and Vectorization in GCC using Lambda Framework

Implementation Issues

• Getting loop information (Loop discovery)
• Finding value spaces of induction variables, array subscript functions, and pointer accesses
• Analyzing data dependence
• Performing linear transformations
Essential Abstractions in GCC
GCC Resource Center, IIT Bombay


Loop Information

Loop Tree

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

Loop Tree

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay


Loop Transformation Passes in GCC

NEXT_PASS (pass_tree_loop);

{  struct opt_pass **p = &pass_tree_loop.pass.sub;
  NEXT_PASS (pass_tree_loop_pass_init);
  NEXT_PASS (pass_tree_loop);
  NEXT_PASS (pass_check_data_deps);
  NEXT_PASS (pass_loop_distribution);
  NEXT_PASS (pass_loop_prop);
  NEXT_PASS (pass_graphite);
  {  struct opt_pass **p = &pass_graphite_pass.sub;
    NEXT_PASS (pass_graphite_trans);
    ...
    }
  NEXT_PASS (pass_iv_anon);
  NEXT_PASS (pass_pll_conversion);
  NEXT_PASS (pass_vectorize);
  {  struct opt_pass **p = &pass_vectorize_pass.sub;
    NEXT_PASS (pass_lower_vecssa);
    NEXT_PASS (pass_dce_loop);
    }  }
  NEXT_PASS (pass_iivcanon);
  NEXT_PASS (pass_if_conversion);
  NEXT_PASS (pass_vectorize);
  {  struct opt_pass **p = &pass_vectorize_pass.sub;
    NEXT_PASS (pass_lower_vecssa);
    NEXT_PASS (pass_dce_loop);
    }  }
  NEXT_PASS (pass_predcom);
  NEXT_PASS (pass_complete_unroll);
  NEXT_PASS (pass_parallelize_loops);
  NEXT_PASS (pass_loop_prefetch);
  NEXT_PASS (pass_iv_optimize);
  NEXT_PASS (pass_tree_loop_done);
}

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

Compiling for Emitting Dumps

• Other necessary command line switches
  ▶ -O3 -fdump-tree-all
-03 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

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay


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-vec</td>
<td></td>
</tr>
<tr>
<td>Dump file extension</td>
<td>.vec</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-parallize-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>
Representing Value Spaces of Variables and Expressions

Chain of Recurrences: 3-tuple (Starting Value, modification, stride)

```c
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, 11, +3</td>
</tr>
<tr>
<td>Induction variable j</td>
<td>11, 11, +2</td>
</tr>
<tr>
<td>Index expression i+1</td>
<td>4, 11, +3</td>
</tr>
<tr>
<td>Index expression 2*j-1</td>
<td>21, 11, -4</td>
</tr>
</tbody>
</table>

Advantages of Chain of Recurrences

CR can represent any affine expression

⇒ Accesses through pointers can also be tracked

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

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;
}
```

```c
gcc -fcheck-data-deps -fdump-tree-ckdd-all -O3 -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;
}                                                  |                     |
| gcc -fcheck-data-deps -fdump-tree-ckdd-all -O3 -S datadep.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 5>; |
|                                               | 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>;
<b bb 4>:
    goto <bb 3>;
```

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

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>;
```

Step 3: Understanding Banerjee's test

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

\[
\begin{align*}
    y &= x + 1 \\
    \Rightarrow x - y + 1 &= 0
\end{align*}
\]
- Find min and max of LHS

\[
\begin{align*}
    x - y + 1 &= 0 \\
    \text{Min: } -148 & \quad \text{Max: } +150
\end{align*}
\]
- RHS belongs to $[-148, +150]$ and dependence may exist

Step 4: Observing the data dependence information

- $\text{iterations\_that\_access\_an\_element\_twice\_in\_A: } [1 + 1 \times x_1]$ last_conflict: 149
- $\text{iterations\_that\_access\_an\_element\_twice\_in\_B: } [0 + 1 \times x_1]$ last_conflict: 149
- $\text{Subscript distance: 1}$

inner loop index: 0
loop nest: (1)
distance\_vector: 1
direction\_vector: +

Example 2: Observing Vectorization and Parallelization

Step 0: Compiling the code with `-O3`

```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`
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;;</td>
</tr>
</tbody>
</table>
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>;
```

Get the number of threads

Perform load calculations
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_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

D.1996 = __builtin_omp_get_num_threads ();
D.1998 = __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

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

\begin{align*}
\text{Control Flow Graph} & \quad \text{Parallel loop body} \\
\langle bb 3 \rangle: & \quad \langle bb 5 \rangle: \\
# i_{11} = PHI <i_{4}(4), 0(2)> & \quad i.8_{21} = (\text{int}) \text{ivtmp.7}_{18}; \\
D.1956_{3} = b[i_{11}] & \quad D.2010_{23} = *b.10_{4}[i.8_{21}]; \\
a[i_{11}] = D.1956_{3} & \quad \text{a}_{115}[i.8_{21}] = D.2010_{23}; \\
i_{4} = i_{11} + 1; & \quad \text{ivtmp.7}_{19} = \text{ivtmp.7}_{18} + 1; \\
\text{if (i}_{4} != 256) & \quad \text{if (D.2006}_{16} > \text{ivtmp.7}_{19}) \\
goto <bb 4>; & \quad \text{goto <bb 5>}; \\
\text{else} & \quad \text{else} \\
goto <bb 5>; & \quad \text{goto <bb 3>}; \\
\langle bb 4 \rangle: & \quad \langle bb 3 \rangle: \\
goto <bb 3>; &
\end{align*}
Example 3: Vectorization but No Parallelization

Step 0: Compiling with
-O3 -fdump-tree-vect-all -msse4

```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

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

Example 3: Vectorization but No Parallelization

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

Example 3: Vectorization but No Parallelization

• Step 3: Observing the conclusion about dependence information

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

• Step 4: Observing the final decision about parallelization

    FAILED: data dependencies exist across iterations
Example 4: No Vectorization and No Parallelization

Step 0: Compiling the code with `-O3`

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

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

Example 4: No Vectorization and No Parallelization

Step 1: Observing the final decision about vectorization

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

Step 2: Observing the final decision about parallelization

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

Advanced Issues in Vectorization

Alignment by 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

Peel Factor = 2
### Advanced Issues in Vectorization

#### Alignment by 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

---

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

---

#### Loop Versioning

How do we vectorize a loop that has

- unaligned data references
- undetermined data dependence relation

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

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

---
Advanced Issues in Vectorization

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

When to Vectorize?

Vectorization is profitable when

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

- **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

Problems with Classical Loop Nest Transforms

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

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

Practical optimizing and parallelizing compilers restore to a predefined set of enabling
Problems with Classical Loop Nest Transforms

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

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

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Requirement

GCC requires a rich algebraic representation that

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

Solution: Polyhedral Representation

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

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Solution: Polyhedral Representation

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

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

GRAPHITE

GRAPHITE is the interface for polyhedral representation of GIMPLE

goal: more high level loop optimizations

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

Compilation Workflow
What Code Can be Represented?

The target of polyhedral representation are sequence of loop nests with

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

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

GPOLY

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

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

Building SCoPs

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

```
int a[256][256], b[245], c[145], n;

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

Example : Building SCoPs

Splitting basic blocks:
The statements and parametric affine inequalities can be expressed by:

- **Iteration Domain** (bounds of enclosing loops)

\[
D^S = \{ i \mid D^S \times (i, g, 1)^T \geq 0 \}
\]

for (i=0; i<m; i++)
for (j=5; j<n; j++)
\[
A[2+i][j+1] = \ldots;
\]

\[
\begin{bmatrix}
i & j & m & n & cst \\
1 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & -1
\end{bmatrix} \geq 0
\]

\[
i \leq m - 1
\]
Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- **Iteration Domain** (bounds of enclosing loops)

  \[ D^S = \{ i \mid D^S \times (i,g,1)^T \geq 0 \} \]

  \[
  \begin{bmatrix}
  i & j & m & n & \text{cst} \\
  2 & 0 & 0 & 0 & 0 \\
  \end{bmatrix}
  \]

  \[
  \begin{bmatrix}
  i & j & m & n & \text{cst} \\
  2 & 0 & 0 & 0 & 0 \\
  \end{bmatrix}
  \]

  for (i=0; i<m; i++)

  for (j=5; j<n; j++)

  \[
  A[2*i][j+1] = \ldots;
  \]

  \[
  j \leq n - 1
  \]

- **Data Reference** (a list of access functions)

  \[
  F = \{ (i,a,s) \mid F \times (i,a,s,g,1)^T \geq 0 \} \]

  \[
  \begin{bmatrix}
  i & j & m & n & \text{cst} \\
  1 & 0 & 0 & 0 & 0 \\
  \end{bmatrix}
  \]

  for (i=1; i<m; i++)

  for (j=5; j<n; j++)

  \[
  A[2*i][j+1] = \ldots;
  \]

  \[
  j \leq n - 1
  \]
Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

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

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

sequence \([s_1, s_2]\):
\[ S[s_1] = t, \quad S[s_2] = t + 1 \]

loop \([loop_1\ s\ end_1]\): \(i\) indexes loop_1 iterations
\[ S[loop_1] = t, \quad S[s] = (t, i_1, 0) \]

Scattering Function
\[ \theta_{S1}(i,j)^T = (0,i,0,j,0)^T \]

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Polyhedral Dependence Analysis in GRAPHITE

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

Legality of Transformations

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

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

Memory location A[0][1] is read at pdr_0 when j = 0 and later written at pdr_1 when j = 1
Dependence : Write after Read

Legality of Transformations

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

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

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

Extra Compilation flag: -floop-parallelize-all

Loop Transforms in GRAPHITE

Loop transforms implemented in GRAPHITE:
- loop interchange
- loop blocking and loop stripmining
- loop flattening

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

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

Loop Interchange in GRAPHITE

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

Strides of $i = 255 + 255 = 510$
Strides of $j = 1 + 1 = 2$

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

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

outermost loop has the largest stride
Loop Interchange in GRAPHITE

Original Code

```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 times synchronization executed = n

After Interchange

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

Outer Loop - parallelizable
Total number of times synchronization executed = 1

Is this loop interchange profitable in GRAPHITE?

Loop Regeneration

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

Original Program

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

Loop generated by CLooG

```c
for (i=0; i<250; i++) {
    for (j=0; j<min(k+2,199); j++) {
        if (j < k+3) S1;
    }
}
```

Merge conditional code with loop bounds if possible

GRAPHITE Conclusions

Advantages of GRAPHITE

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

Future Scope

- Making instancewise dependence analysis algorithmically cheaper
- Automating the search most profitable transform composition sequence
- Developing efficient cost models
- Exploring scalability issues
Parallelization and Vectorization in GCC: Conclusions

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

Thank You!