LORE: A Loop Repository for the Evaluation of Compilers

Zhi Chen∗, Zhangxiaowen Gong†, Justin Josef Szaday†, David C. Wong‡, David Padua†, Alexandru Nicolau∗, Alexander V Veidenbaum∗, Neftali Watkinson∗, Zehra Sura§, Saeed Maleki¶, Josep Torrellas†, Gerald DeJong†

‡Intel Corporation, §IBM Research, ¶Microsoft Research,
∗University of California, Irvine, †University of Illinois at Urbana-Champaign
Background

• Loop transformation
  • Better locality, less computation, and parallelism/vectorization
  • Advantage
    • Programmers can focus on preparation of the readable and correct algorithm
  • Challenges
    • Will compilers generate efficient code?
    • Is it necessary to manually transform code?
  • Much room for advances in compiler technology
Motivation

• Data are required for measurements
  • Different classes of code and target machines
  • Compilers across successive generations
  • Solo/compound transformations

• This data is not widely available

• Loop repository (https://vectorization.computer)
  • Experimentation and comparison of results
  • Effect of transformations
  • Execution time of different transformation sequences
System infrastructure
System Modules

C applications

Loop extractor

Loop identification and extraction

Source level instrumentation

Input dataset capturing

Dependence analysis

Loop transformations

Loop mutator

Loop Repository

Performance measurement

Characteristic extraction

Website interfaces

Other tools

Similarity analysis

Loop clustering

Loop clusterer
System Modules

- C applications
- C applications
- Dependence analysis
  - Loop transformations
  - Loop mutator
- Loop identification and extraction
- Source level instrumentation
- Input dataset capturing
- Loop clusterer
- Loop clusterer
- Similarity analysis
- Loop clustering
- Loop Repository
- Loop extractor
- Performance measurement
  - Characteristic extraction
  - Website interfaces
  - Other tools
System Modules

C applications

Dependence analysis
Loop transformations
Loop mutator

Loop identification and extraction
Source level instrumentation
Input dataset capturing
Loop extractor

Loop Repository
Loop mutator

Performance measurement
Characteristic extraction
Website interfaces
Other tools

Similarity analysis
Loop clustering
Loop clusterer
System Modules

C applications

Loop extractor

Loop identification and extraction

Source level instrumentation

Input dataset capturing

Dependence analysis

Loop transformations

Loop mutator

Loop Repository

Performance measurement

Characteristic extraction

Website interfaces

Other tools

Similarity analysis

Loop clustering

Loop clusterer
System Modules

- C applications
- Source level instrumentation
- Input dataset capturing
- Loop extractor
- Loop identification and extraction
- Loop Repository
- Loop transformations
- Loop mutator
- Dependence analysis
- Similarity analysis
- Loop clustering
- Loop clusterer
- Performance measurement
- Characteristic extraction
- Website interfaces
- Other tools
System Modules
System Module: Extractor

• Goal
  • Build standalone codelets with qualified loops
  • Replayable with the original data

• Based on ROSE source-to-source compiler
  • Identify interesting nodes by traversing AST
  • Insert source level instrumentation
System Module: Extractor

// Code before the loop
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
    }
}

// Code after the loop
System Module: Extractor

```c
// Code before the loop
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
    }
}

// Code after the loop
```
Collect values for $N$, $a$, $u_1$, $v_1$, $u_2$, $v_2$
Save them to the disk

```c
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
    }
}
```

// Code before the loop

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
    }
}
```

// Code after the loop

Data copying
Collect values for N, a, u1, v1, u2, v2
Save them to the disk

Code before the loop
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
    }
}

Code after the loop

Data copying
Codelet driver

- Restore execution environment
- Initialize variables with collected data
- Invoke the loop
- Collect measurements
  - *Performance counter values and execution cycles*
Codelet: Dynamic Features

• Categories
  • memory traffic
  • memory hits
  • line fill buffer (LFB) occupancy
  • TLB related events
  • resource stalls
  • Prefetching
  • Instruction mix (PIN tool)
    • Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    • # scalar and vector loads, # scalar and vector stores
Codelet: Dynamic Features

• Categories
  • memory traffic
  • memory hits
  • line fill buffer (LFB) occupancy
  • TLB related events
  • resource stalls
  • Prefetching
  • Instruction mix (PIN tool)
    • Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    • # scalar and vector loads, # scalar and vector stores

From performance counters
Codelet: Dynamic Features

• Categories
  • memory traffic
  • memory hits
  • line fill buffer (LFB) occupancy
  • TLB related events
  • resource stalls
  • Prefetching
  • Instruction mix (PIN tool)
    • Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    • # scalar and vector loads, # scalar and vector stores

Usage of features:
Codelet: Dynamic Features

• Categories
  • memory traffic
  • memory hits
  • line fill buffer (LFB) occupancy
  • TLB related events
  • resource stalls
  • Prefetching
  • Instruction mix (PIN tool)
    • Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    • # scalar and vector loads, # scalar and vector stores

Usage of features:
• Understand performance results
Codelet: Dynamic Features

• Categories
  • memory traffic
  • memory hits
  • line fill buffer (LFB) occupancy
  • TLB related events
  • resource stalls
  • Prefetching
  • Instruction mix (PIN tool)
    • Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    • # scalar and vector loads, # scalar and vector stores

Usage of features:
• Understand performance results
• Determine performance effect produced by our mutator
System Module: Mutator

• Goal: to search in a loop transformation space for performance improvement

• Sequence of source-to-source transformations, including:
  • Interchange
  • Tiling
  • Unrolling
  • Unroll-and-jam
  • Distribution

• Validated by dependence analysis (if necessary)
System Module: Clusterer

• Goals: reduce redundant loops and identify loop patterns

• For each codelet, we collect:
  • Data for 28 different static features
  • Dependence graph

• Uses a simple, hierarchical clustering algorithm

• Distance metric is weighted
  • Favors the dependence graph when it is available

• loop nest level
• lower/upper bounds and stride
• # statements
• # memory operations
• # floating-point/integer operations
• # constant
• # branches
• # basic blocks (BB) and edges in the loop’s CFG
The Repository: LORE

• Statistics (as of the writing of the paper)
  • 25 benchmark suites
    • Audio/Video codecs, deep learning, SPEC, real-world applications from GitHub, etc.
  • ~2500 codelets, ~90,000 mutated codelet versions
  • 40 performance counters
  • Instruction mix
  • 28 static features
  • 3 compilers each with 2 versions
    • 5 vectorization settings on the same HW configuration
  • These numbers keep growing
Platform

• Software

<table>
<thead>
<tr>
<th>Parameters</th>
<th>GCC</th>
<th>ICC</th>
<th>Clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Model</td>
<td>Intel Xeon E5-1630 v3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 cache</td>
<td>4.8.5</td>
<td>15.0.6</td>
<td>3.6.2</td>
</tr>
<tr>
<td>L2 cache</td>
<td>6.2.0</td>
<td>17.0.1</td>
<td>4.0.0</td>
</tr>
</tbody>
</table>

• Hardware

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Model</td>
<td>Intel Xeon E5-1630 v3</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256KB</td>
</tr>
<tr>
<td>L3 cache</td>
<td>10MB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Ubuntu 16.04</td>
</tr>
</tbody>
</table>
## Compiler flags

<table>
<thead>
<tr>
<th>GCC 6.2.0</th>
<th>Baseline</th>
<th>-03 -ftree-vectorize-ftree-vectorize-fsafe-math-funsafe-loop-optimizations-ftree-loop-if-convert-stores-march=native-mtune=native</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>[Baseline Flags] -f vect-cost-model=unlimited -mno-avx -mno-avx2</td>
<td></td>
</tr>
<tr>
<td>AVX</td>
<td>[Baseline Flags] -f vect-cost-model=unlimited -mno-avx2</td>
<td></td>
</tr>
<tr>
<td>AVX2</td>
<td>[Baseline Flags] -f vect-cost-model=unlimited</td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>-03 -fno-tree-vectorize -f safe-math -funsafe-loop-optimizations -march=native -mtune=native</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clang 4.0.0</th>
<th>Baseline</th>
<th>-03 -f vectorize-fslp-vectorize-fslp-vectorize-aggressive -ffast-math -march=native -mtune=native</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>[Baseline Flags] -mno-avx -mno-avx2</td>
<td></td>
</tr>
<tr>
<td>AVX</td>
<td>[Baseline Flags] -mno-avx2</td>
<td></td>
</tr>
<tr>
<td>AVX2</td>
<td>[Baseline Flags]</td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>-03 -f no-vectorize -f no-slp-vectorize -ffast-math -march=native -mtune=native</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ICC 16.0.3</th>
<th>Baseline</th>
<th>-03 -restrict-xHost -ipo</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>-03 -restrict -ipo -vec-threshold0 -msse4.2</td>
<td></td>
</tr>
<tr>
<td>AVX</td>
<td>-03 -restrict -ipo -vec-threshold0 -mavx</td>
<td></td>
</tr>
<tr>
<td>AVX2</td>
<td>[Baseline Flags] -vec-threshold0</td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>[Baseline Flags] -no-vec</td>
<td></td>
</tr>
</tbody>
</table>
Website (https://vectorization.computer)

- Publicly accessible
- Users can
  - View/download source/assembly
  - Discuss loops
- Visualize results
- Run queries
Benchmark: NPB 2.3-OpenACC-C
Application: BT

Baseline source code

```c
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
extern int i;
extern int j;
extern int k;
extern int m;
extern double u[(64 + 1) / 2 * 2 + 1][64 / 2 * 2 + 1][64 / 2 * 2 + 1];
extern double rhs[64 / 2 * 2 + 1][64 / 2 * 2 + 1][64 / 2 * 2 + 1];

void loop()
{
    int _arr_sub_0 = grid_points[0];
    int _arr_sub_1 = grid_points[1];
    int _arr_sub_2 = grid_points[2];
    int _i_3 = 1;
    int _j_4 = 1;
    int _k_5 = k;
    int _m_6 = m;

    #pragma scop
    for (_i_3 = 1; _i_3 <= _arr_sub_0 - 1; _i_3 += 1) {
        for (_j_4 = 1; _j_4 <= _arr_sub_1 - 1; _j_4 += 1) {
            for (_k_5 = 1; _k_5 <= _arr_sub_2 - 1; _k_5 += 1) {
                u[_i_3][_j_4][_k_5][_m_6] = u[_i_3][_j_4][_k_5][_m_6] + rhs[_i_3][_j_4][k][m_6];
            }
        }
    }
}
```

Static features of processed baseline

- Nest levels: 4
- Statements: 13
- Memory accesses: 3
- Floating point binary operations: 5
- Integer binary operations: 18
- Integer unary operations: 0
- Assignments: 9
- Constants: 15
- Constant 0s: 1
- Constant 1s: 13
- Binary operations with integer constants: 15
- Basic blocks: 14
- Basic blocks with 1 successor: 10
- Basic blocks with 2 successors: 4
- Basic blocks with 3 or more successors: 0
- Basic blocks with 1 predecessor: 9
- Basic blocks with 2 predecessors: 4
### Hardware Dynamic Counters

<table>
<thead>
<tr>
<th>Metric</th>
<th>Reference</th>
<th>Scalar</th>
<th>SSE</th>
<th>AVX</th>
<th>AVX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>INST_RETIRED_ANY</td>
<td>21325</td>
<td>21325</td>
<td>19778</td>
<td>18673</td>
<td>16213</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_THREAD</td>
<td>12229</td>
<td>11508</td>
<td>8817</td>
<td>8285</td>
<td>7791</td>
</tr>
<tr>
<td>MEM_LOAD_UOPS_RETIRED_L1_HIT_PS</td>
<td>8685</td>
<td>8674</td>
<td>7053</td>
<td>7072</td>
<td>6862</td>
</tr>
<tr>
<td>L1D_PEND_MISS_PENDING</td>
<td>790</td>
<td>677</td>
<td>586</td>
<td>592</td>
<td>601</td>
</tr>
<tr>
<td>L1D_PEND_MISS_PENDING_CYCLES</td>
<td>730</td>
<td>531</td>
<td>458</td>
<td>514</td>
<td>495</td>
</tr>
<tr>
<td>RESOURCESTALLS_ANY</td>
<td>946</td>
<td>1043</td>
<td>39</td>
<td>178</td>
<td>153</td>
</tr>
<tr>
<td>RESOURCESTALLS_MEM_RS</td>
<td>1008</td>
<td>1010</td>
<td>27</td>
<td>169</td>
<td>183</td>
</tr>
<tr>
<td>RESOURCESTALLS_SB</td>
<td>940</td>
<td>1019</td>
<td>18</td>
<td>141</td>
<td>39</td>
</tr>
<tr>
<td>L1D_REPLACEMENT</td>
<td>158</td>
<td>163</td>
<td>104</td>
<td>102</td>
<td>113</td>
</tr>
<tr>
<td>RESOURCESTALLS_RS</td>
<td>6</td>
<td>4</td>
<td>27</td>
<td>85</td>
<td>136</td>
</tr>
</tbody>
</table>

Showing 1 to 10 of 40 entries

### Dynamic Instruction Counts

<table>
<thead>
<tr>
<th>Metric</th>
<th>Reference</th>
<th>Scalar</th>
<th>SSE</th>
<th>AVX</th>
<th>AVX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_cnt</td>
<td>20960</td>
<td>20960</td>
<td>19433</td>
<td>18308</td>
<td>15848</td>
</tr>
<tr>
<td>fp_arith_inst_cnt</td>
<td>5000</td>
<td>5000</td>
<td>6000</td>
<td>6000</td>
<td>3000</td>
</tr>
<tr>
<td>fp_sse_inst_cnt</td>
<td>0</td>
<td>0</td>
<td>4000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vector_inst_exclude_mem_cnt</td>
<td>0</td>
<td>0</td>
<td>4000</td>
<td>4000</td>
<td>2000</td>
</tr>
<tr>
<td>scalar_id_cnt</td>
<td>8697</td>
<td>8697</td>
<td>3702</td>
<td>3702</td>
<td>3702</td>
</tr>
<tr>
<td>benchmark</td>
<td>application</td>
<td>file</td>
<td>line</td>
<td>n_mutations</td>
<td>transformations</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>------------------</td>
<td>------</td>
<td>-------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>ALPBench</td>
<td>MPGencl</td>
<td>predict.c</td>
<td>511</td>
<td>4</td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>ALPBench</td>
<td>Ray_trace</td>
<td>imageno.c</td>
<td>167</td>
<td>4</td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>ALPBench</td>
<td>csuFacIdEval_5.1</td>
<td>csuCommonMatrix.c</td>
<td>435</td>
<td>4</td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>ASC-lm</td>
<td>amsmk</td>
<td>csr_matvec.c</td>
<td>124</td>
<td>4</td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>cortexsuite</td>
<td>svd3</td>
<td>svd.c</td>
<td>293</td>
<td>4</td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>Fhournstones</td>
<td>default</td>
<td>c4.c</td>
<td>131</td>
<td></td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>FreeBench</td>
<td>neural</td>
<td>neural.c</td>
<td>711</td>
<td>16</td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>FreeBench</td>
<td>pffft</td>
<td>pffft.c</td>
<td>653</td>
<td>4</td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>FreeBench</td>
<td>pffft</td>
<td>pffft.c</td>
<td>558</td>
<td>16</td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>GitApps</td>
<td>gap</td>
<td>trans.c</td>
<td>520</td>
<td></td>
<td>interchange tiling unrolling distribution</td>
</tr>
<tr>
<td>GitApps</td>
<td>mozjpeg</td>
<td>jchuff.c</td>
<td>1009</td>
<td>4</td>
<td>interchange tiling unrolling distribution</td>
</tr>
</tbody>
</table>
Sample usage
Comparison of vectorizers

<table>
<thead>
<tr>
<th></th>
<th>GCC 6.2.0</th>
<th>ICC 17.0.1</th>
<th>Clang 4.0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td># of loops included</td>
<td></td>
<td>959</td>
<td></td>
</tr>
<tr>
<td># (%) of loops originally having effective vectorization</td>
<td>273 (28.5%)</td>
<td>395 (41.2%)</td>
<td>216 (22.5%)</td>
</tr>
<tr>
<td># (%) of additional loops vectorized effectively and beneficial against the original baseline after mutation</td>
<td>77 (8.0%)</td>
<td>63 (6.6%)</td>
<td>98 (10.2%)</td>
</tr>
<tr>
<td>Subtotal</td>
<td>350 (36.5%)</td>
<td>458 (47.8%)</td>
<td>314 (32.7%)</td>
</tr>
</tbody>
</table>

- Effective/beneficial means at least 1.15x speedup
Comparison of vectorizers

<table>
<thead>
<tr>
<th></th>
<th>GCC 6.2.0</th>
<th>ICC 17.0.1</th>
<th>Clang 4.0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td># of loops included</td>
<td>959</td>
<td></td>
<td></td>
</tr>
<tr>
<td># (% of loops originally</td>
<td>273 (28.5%)</td>
<td>395 (41.2%)</td>
<td>216 (22.5%)</td>
</tr>
<tr>
<td>having effective vectoriz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># (% of additional loops</td>
<td>77 (8.0%)</td>
<td>63 (6.6%)</td>
<td>98 (10.2%)</td>
</tr>
<tr>
<td>vectorized effectively</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and beneficial against</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the original baseline</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>after mutation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtotal</td>
<td>350 (36.5%)</td>
<td>458 (47.8%)</td>
<td>314 (32.7%)</td>
</tr>
</tbody>
</table>

• Effective/beneficial means at least 1.15x speedup
Comparison of vectorizers

(a) GCC

(b) ICC

(c) Clang
Comparison of vectorizers

(a) GCC

(b) ICC

(c) Clang
Comparison of vectorizers

(a) GCC
(b) ICC
(c) Clang
Comparison of vectorizers

(a) GCC

(b) ICC

(c) Clang
GCC 6.2.0 VS GCC 4.8.5

beneficial mutations (speedup $\geq 1.15$)

unfavorable mutations (slowdown $> 1.15$)
GCC 6.2.0 VS GCC 4.8.5

Reasons for slowdown:

beneficial mutations (speedup >= 1.15)

unfavorable mutations (slowdown > 1.15)
GCC 6.2.0 VS GCC 4.8.5

Reasons for slowdown:
• Compilers can optimize the loop well by themselves

beneficial mutations (speedup $\geq 1.15$)

unfavorable mutations (slowdown $> 1.15$)
GCC 6.2.0 VS GCC 4.8.5

Reasons for slowdown:
• Compilers can optimize the loop well by themselves
• The number of transformations is inadequate

beneficial mutations (speedup >= 1.15)  unfavorable mutations (slowdown > 1.15)
Conclusion

• Developed a loop repository which consists of a series of tools to extract, mutate, and classify loops
• Provided a broad range of characteristics for each codelet
• The repository could help the community to identify the strength and weakness of a compiler
• Sample usage showed that there is still room for compilers to improve
Future work

• Incorporate more loop transformations
• Measure the importance of loops during extraction
• Evaluate more compilers and more platforms