When Polyhedral Transformations Meet SIMD Code Generation

Kong, Veras, Stock, Frachetti, Puchet, Sadayappan

PLDI ‘13
Objectives for Performance on multi-cores

- Data locality
- Fine-grain and coarse grain parallelism
- Most current optimization frameworks fail at achieving one of these three objectives.
SIMD Vectorization is important?

- Vector instruction width is increasing; 256 bits in AVX and 512 bits in AVX-512.

- Vectorizing compilers face significant challenges in utilizing the machine peak.

- Transformations require auxiliary code transformations like loop skewing.
Approach

- Address data locality by generating tiled code.
- Maximize parallelism in the innermost loop for SIMD vectorization.
- Back-end optimizer and code generator
  - Optimization of register load/store operations
  - Target-specific code generation
Properties of a vectorizable loop

- there is no loop carried dependence.
- all array references and stride-0 or stride-1
- unaligned loads/stores are avoided whenever possible.
Vectorizable Codelet Extraction

- Each vectorizable inner-loop is a vectorizable codelet.

- Exploit the data reuse potential by increasing the number of computations in the codelet.

- This is done through unroll-and-jam.
Vectorizable codelet example

Consider the following example loop nest:

```c
for (i = lbi; i < ubi; ++i)
    for (j = lbj; j < ubj; ++j) {
        R:  A[i-1][j] = B[i-1][j];
        S:  B[i][j] = C[i]*A[i-1][j];
    }
```

After unroll-and-jam:

```c
for (i = lbi; i < ubi; i += 2)
    for (j = lbj; j < ubj; ++j) {
        R:  A[i-1][j] = B[i-1][j];
        S:  B[i][j] = C[i]*A[i-1][j];
        R:  A[i][j] = B[i][j];
        S:  B[i+1][j] = C[i+1]*A[i][j];
    }
```
Vectorizable codelet example (contd.)

After SIMD vectorization:

```
for (i = lbi; i < ubi; i += 2)
  // Prolog: peel for alignment.
  lbj2 = & (A[i-1][j]) % V
  for (j = lbj; j < lbj + lbj2; ++j) {
    A[i-1][j] = B[i-1][j];
    B[i][j] = C[i] * A[i-1][j];
  }
  // Body: codelet (abstract vectorization)
  ubj2 = (lbj2 - ubj) % V
  for (j = lbj2; j < ubj - ubj2; j += V) {
    vstore(A[i-1][j], vload(B[i-1][j]));
    vstore(B[i][j], vmul(vsplat(C[i]),
                     vload(A[i-1][j])));
    vstore(A[i][j], vload(B[i][j]));
    vstore(B[i+1][j], vmul(vsplat(C[i+1]),
                         vload(A[i][j])));
  }
```
Polyhedral Framework for Codelet Extraction

- Internally represents loops and their data dependence as a collection of parametric polyhedra.
  - Each statement has an iteration domain.
  - Each memory access is defined by an access function.
  - Data dependences are represented using dependence polyhedra.
  - Program transformation using a scheduling function.
Iteration Domain

- When a statement is enclosed by one or more loops all iterations are captured in the iteration domain.
- Iteration domain of R in the previous example is

\[ D_R = \{(i, j) \in \mathbb{Z}^2 \mid lbi \leq i < ubi \land lbj \leq j < ubj\} \]
Access function and data dependences

- For each point in the iterations domain access function returns the coordinate of the array.
- Given to statements R and S a dependence polyhedron contains all pairs of dependence instances between R and S.
Scheduling function

- Function that maps each point in the iteration domain to vector of time-stamps.
- This is used to reorder the points in the iteration domain.
- The loops iterations will be scheduled lexicographic order of their associated time-stamp.
Optimization scheduling algorithm

- The scheduling algorithm works in two steps
  - find a schedule that maximizes the parallelism.
  - Minimizing the number of unaligned loads and stores on the transformed program.
Minimizing dependence distance

- Designed to find good data locality.
- Find a minimization function that bounds the dependence distance.

\[ u_k \cdot \bar{n} + w_k \geq \Theta^S(\bar{x}_S) - \Theta^R(\bar{x}_R) \quad \langle \bar{x}_R, \bar{x}_S \rangle \in D_{R,S} \]

\[ u_k \in \mathbb{N}^p, w_k \in \mathbb{N} \]
Maximizing fine-grain parallelism

- Requires that no dependences in the generated code are carried by the innermost loop.
- Minimizing the number of dependences in the innermost loop + minimizing dependence distance.
Minimizing unaligned loads/stores

- Perform a combinations of statement retiming and additional loop skewing.
- “Shift” statements such that all array elements in the vectorizable loop become aligned.
- Skewing the loop by a positive factor so that the array index variable is a factor of vector length.
Finally code generation!

- The codelet generator takes as input a LISP-like IR.
- The IR conveys information about
  - alignments
  - loop-trip count properties
- Performs optimizations like CSE, array scalarization and strength reduction.
Implementation framework

- The framework was implemented using the PolyOpt/C polyhedral compiler.

- SIMD codelet generator is implemented in the SPIRAL system’s back-end infrastructure.
Experimental Setup

● 5 typical image processing and physical simulation applications.

● 3 linear algebra benchmarks.

● Experiments were performed on Intel Sandybridge using either SSE or AVX vector instruction set.
## Time breakdown for the kernels

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SIMD ISA</th>
<th>ATC (sec)</th>
<th>AFT (sec)</th>
<th>FP (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi-2d</td>
<td>SSE</td>
<td>0.040</td>
<td>0.046</td>
<td>0.068</td>
</tr>
<tr>
<td>jacobi-3d</td>
<td>SSE</td>
<td>0.458</td>
<td>0.514</td>
<td>1.113</td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>AVX</td>
<td>0.036</td>
<td>0.054</td>
<td>0.077</td>
</tr>
<tr>
<td>jacobi-3d</td>
<td>AVX</td>
<td>0.188</td>
<td>0.354</td>
<td>1.062</td>
</tr>
<tr>
<td>laplacian-2d</td>
<td>SSE</td>
<td>0.046</td>
<td>0.056</td>
<td>0.071</td>
</tr>
<tr>
<td>laplacian-3d</td>
<td>SSE</td>
<td>0.482</td>
<td>0.538</td>
<td>1.131</td>
</tr>
<tr>
<td>laplacian-2d</td>
<td>AVX</td>
<td>0.031</td>
<td>0.042</td>
<td>0.061</td>
</tr>
<tr>
<td>laplacian-3d</td>
<td>AVX</td>
<td>0.197</td>
<td>0.373</td>
<td>0.993</td>
</tr>
<tr>
<td>poisson-2d</td>
<td>SSE</td>
<td>0.051</td>
<td>0.064</td>
<td>0.090</td>
</tr>
<tr>
<td>poisson-2d</td>
<td>AVX</td>
<td>0.029</td>
<td>0.049</td>
<td>0.075</td>
</tr>
<tr>
<td>correlation</td>
<td>SSE</td>
<td>0.840</td>
<td>0.915</td>
<td>1.179</td>
</tr>
<tr>
<td>correlation</td>
<td>AVX</td>
<td>0.752</td>
<td>0.917</td>
<td>1.139</td>
</tr>
<tr>
<td>covariance</td>
<td>SSE</td>
<td>0.846</td>
<td>0.922</td>
<td>1.159</td>
</tr>
<tr>
<td>covariance</td>
<td>AVX</td>
<td>0.733</td>
<td>0.883</td>
<td>1.121</td>
</tr>
<tr>
<td>doitgen</td>
<td>SSE</td>
<td>0.191</td>
<td>0.353</td>
<td>0.865</td>
</tr>
<tr>
<td>doitgen</td>
<td>AVX</td>
<td>0.158</td>
<td>0.308</td>
<td>0.834</td>
</tr>
</tbody>
</table>
Observations

● Peeling is the main reason for loss of performance from ATC to AFT, slowdown between 1.1x and 2x.
● Partial tiles can cause a slowdown of 1.3x to 3x over AFT.
● Overall slowdown of peeling and partial tile execution can be 1.4x to 6x.
Experimental evaluation

- For each Vector ISA and data type precision this implementation is compared against ICC and PTile, a parallel parametric tiling software.

- Measured in GFLOP/s
## Performance against other frameworks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PB Size</th>
<th>seq</th>
<th>par</th>
<th>ICC</th>
<th>PTile</th>
<th>Prevect</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi-2d</td>
<td>$20 \times 2000^2$</td>
<td>2.96</td>
<td>3.71</td>
<td>6.24</td>
<td>17.15</td>
<td>13.22</td>
<td>25.39</td>
</tr>
<tr>
<td>laplacian-2d</td>
<td>$20 \times 2000^2$</td>
<td>4.30</td>
<td>4.44</td>
<td>6.29</td>
<td>18.7</td>
<td>17.85</td>
<td>24.86</td>
</tr>
<tr>
<td>poisson-2d</td>
<td>$20 \times 2000^2$</td>
<td>4.23</td>
<td>6.68</td>
<td>17.47</td>
<td>19.56</td>
<td>14.75</td>
<td>31.77</td>
</tr>
<tr>
<td>jacobi-3d</td>
<td>$20 \times 256^3$</td>
<td>4.21</td>
<td>4.70</td>
<td>3.82</td>
<td>5.38</td>
<td>5.04</td>
<td>3.84</td>
</tr>
<tr>
<td>laplacian-3d</td>
<td>$20 \times 256^3$</td>
<td>4.80</td>
<td>5.44</td>
<td>4.39</td>
<td>6.13</td>
<td>5.67</td>
<td>6.99</td>
</tr>
<tr>
<td>correlation</td>
<td>$2000^3$</td>
<td>0.94</td>
<td>0.81</td>
<td>26.21</td>
<td>18.78</td>
<td>19.89</td>
<td>33.35</td>
</tr>
<tr>
<td>covariance</td>
<td>$2000^3$</td>
<td>0.97</td>
<td>0.99</td>
<td>26.53</td>
<td>18.70</td>
<td>20.08</td>
<td>34.85</td>
</tr>
<tr>
<td>doitgen</td>
<td>$256^4$</td>
<td>8.63</td>
<td>8.56</td>
<td>14.72</td>
<td>31.35</td>
<td>30.76</td>
<td>41.63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PB Size</th>
<th>seq</th>
<th>par</th>
<th>ICC</th>
<th>PTile</th>
<th>Prevect</th>
<th>SIMD</th>
<th>seq</th>
<th>par</th>
<th>ICC</th>
<th>PTile</th>
<th>Prevect</th>
<th>SIMD</th>
<th>seq</th>
<th>par</th>
<th>ICC</th>
<th>PTile</th>
<th>Prevect</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlation</td>
<td>$2000^3$</td>
<td>0.64</td>
<td>0.63</td>
<td>14.38</td>
<td>9.56</td>
<td>9.66</td>
<td>21.28</td>
<td>26.29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>covariance</td>
<td>$2000^3$</td>
<td>0.65</td>
<td>2.16</td>
<td>13.52</td>
<td>9.66</td>
<td>9.70</td>
<td>21.88</td>
<td>27.23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Performance data in GFLOP/s.
Observations

- Upto 50x improvement over ICC and Ptile.
- Despite being in to form that satisfies the vectorizability criteria, C code generated by the other frameworks does not lead to effective SIMD parallelism.
Conclusion

- Viable scheme to achieve data locality and parallelism for compute intensive programs.
- Isolated optimizations that can be performed by high-level loop transformation engine from that can be implemented by SIMD code generation.
- Used polyhedral compilation to formalize constraints for vectorizable codelets targeting parallelism data reuse and alignment.