Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Code-Transformation

D. Feld, T. Sodemann, M. Jünger, S. Mallach

Fraunhofer SCAI & University of Cologne
## Topics

1. Introduction
2. Polyhedral Model
3. PluTo-SICA
4. Benchmarks
5. Summary and future work
• automatic **vectorizers** are implemented in recent compiler frameworks
• but the archived **performance** due to this potential varies regarding
  ▶ the compilers **transformation** potential
  ▶ the ability to handle ‘any’ **parameter** constellations
  ▶ the memory **access patterns** and
  ▶ the consequent **access time** to the memory
• these issues depend strongly on
  ▶ the **structure** of the prospected code and on
  ▶ the **characteristics** of the targeted hardware
• for vectorization e.g. within x86 CPUs with SSE or AVX we:

### Issue
**Support the compiler** to perform calculations through the SIMD registers.

### Issue
Force an **extensive cache usage** to archive fast access to the streamed data.
The model

The model is a polyhedral model that facilitates SIMD-code generation. It involves the transformation of source code into a model, which is then transformed further to generate output code.

For detailed steps and mathematical formulations, refer to the source code and models provided by Feld, Soddemann, Jünger, and Mallach (SICA).
Transformation for vectorization

- to **facilitate** vectorization within a loop nest, it may have to be transformed
  - iterations to be vectorized must be **parallelizable** (independent)
  - iterations to be vectorized must become **innermost** within the nest

- to **archive** those properties
  - loops my have to be skewed
  - loops my have to be interchanged
  - ...

- to **exploit** the full potential of vectorization, one further has to
  - optimize for data-locality
  - take the hardware into account for the transformation
"Blocking (or Tiling) is a well-known optimization technique for improving the effectiveness of memory hierarchies. Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks, so that data loaded into the faster levels of the memory hierarchy are reused."

(Lam, Rothberg and Wolf 1991)
Automatic parallelization tool based on the polyhedral model

- to perform high-level transformations such as
  - loop-nest optimization and
  - parallelization

on affine loop nests

- transforms C programs from source to source

- for coarse-grained parallelism

- and data locality

  - PluTo does not contain ‘real’ tile size selection strategies
  - but a default strategy that tiles every loop in a static size (32 for one level and 8 for the second)
  - fits well for recent CPUs and ‘common’ scientific codes

- as well as basic vectorization support
• applying PluTo’s **transformations** to support vectorization

**SIMD- and cache-spezific (SICA) tiling**

• performing a tiling of **specific loops**
  ▶ the vectorized loop
  ▶ and (optionally) the outermost loop

• that is particularly **related to the transformations**

• adapting the tile sizes to the **underlying hardware**

• **automatically** read out the hardware parameters

→ **support the compiler** at vectorization and
→ **optimize** the resulting performance
PluTo (default) vs. PluTo-SICA

**PluTo (default) L1 tiling**

1. for(ii=0; ii≤floor(M-1,32); ii++)
2. for(jj=0; jj≤floor(N-1,32); jj++)
3. for(kk=0; kk≤floor(K-1,32); kk++)
4. for(i=32*ii; i≤min(M-1,32*ii+31); i++)
5. for(j=32*jj; j≤min(N-1,32*jj+31); j++)
6. for(k=32*kk; k≤min(K-1,32*kk+31); k++)

**PluTo-SICA L1 tiling**

1. for(i=0; i<M; i++)
2. for(jj=0; jj≤floor(N-1,qL1); jj++)
3. for(k=0; k<K; k++)
4. for(j=qL1*jj; j≤min(N-1,qL1*jj+(qL1 - 1)); j++)

---

Feld, Soddemann, Jünger, Mallach (SICA)

Facilitate SIMD-Code-Generation

January 21, 2013
Comparison of the approaches

accesses of the vectorized loop

MEMORY SPACE
PluTo (default) L1 tiling

MEMORY SPACE
PluTo-SICA L1 tiling
PluTo (default) vs. PluTo-SICA

**PluTo (default) L1 + L2 tiling**

1. for(iii = 0; iii ≤ floor(M - 1, 256); iii++)
2. for(jjj = 0; jjj ≤ floor(N - 1, 256); jjj++)
3. for(kkk = 0; kkk ≤ floor(K - 1, 256); kkk++)
4. for(ii = 8 * iii; ii ≤ min(floor(M - 1, 32), 8 * iii + 7); ii++)
5. for(jj = 8 * jjj; jj ≤ min(floor(N - 1, 32), 8 * jjj + 7); jj++)
6. for(kk = 8 * kkk; kk ≤ min(floor(K - 1, 32), 8 * kkk + 7); kk++)
7. for(i = 32 * ii; i ≤ min(M - 1, 32 * ii + 31); i++)
8. for(j = 32 * jj; j ≤ min(N - 1, 32 * jj + 31); j++)
9. for(k = 32 * kk; k ≤ min(K - 1, 32 * kk + 31); k++)

**PluTo-SICA L1 + L2 tiling**

1. for(i = 0; i < M; i++)
2. for(j = 0; j < N; j++)
3. for(k = 0; k < K; k++)

4. for(i = q^{L2} * ii; j ≤ min(N - 1, q^{L2} * ii + (q^{L2} - 1)); i++)
5. for(j = q^{L1} * jj; j ≤ min(N - 1, q^{L1} * jj + (q^{L1} - 1)); j++)
Comparison of the approaches

accesses of the vectorized loop

MEMORY SPACE
PluTo (default) L1+L2 tiling

MEMORY SPACE
PluTo-SICA L1+L2 tiling
Goals of the SICA approach

- **Adjusting** the tile sizes $q^{L1}$ and $q^{L2}$ to the cache
- therefor it has to be determined, how much data has to be loaded per iteration of the vectorized loop
  → analysis of the array access functions
  → mechanism to detect different array accesses
- how many different data elements have to be loaded for one resulting block of the vectorized loop

generate a pipeline of blocks that can be
- loaded through the cache hierarchy
- by successful prefetching

combined with an extensive vectorization
Goals of the SICA approach

**Issue**

**Support the compiler** to perform calculations through the SIMD registers.

**Issue**

Force an **extensive cache usage** to achieve fast access to the streamed data.

MODELL

HARDWARE
Goals of the SICA approach

**Issue**

Support the compiler to perform calculations through the SIMD registers.

**Solution**

register fitting tile sizes

→ parameter-independent vectorization

**Issue**

Force an extensive cache usage to achieve fast access to the streamed data.

**Solution**

cache adapted tile sizes

→ good load through cache hierarchy
SIMD- and Cache-specific Tiling

**Derivation of tile sizes**

- **L1-Tile-Size**: $q^{L1} \approx \frac{\text{CaSiEl}}{\text{ElPelt}}$

  - CaSiEl: **Cache Size in Elements**
  - ElPelt: **Elements Per Iteration**

  Automatically or manually detected
Derivation of tile sizes

- L1-Tile-Size: $q^{L1} \approx \rho \times \frac{\text{CaSiEl}}{\text{ElPelt}}$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho$</td>
<td>Ratio of cache to use [%]</td>
</tr>
<tr>
<td>CaSiEl</td>
<td>Cache Size in Elements</td>
</tr>
<tr>
<td>ElPelt</td>
<td>Elements Per Iteration</td>
</tr>
</tbody>
</table>

default or manually automatically or manually automatically detected
### Derivation of tile sizes

- **L1-Tile-Size**: 
  \[ q^{L1} = \left\lfloor \rho * \frac{\text{CaSiEl}}{\text{ElPeIt} * \text{ElPeRe}} \right\rfloor * \text{ElPeRe} \]

- \( \rho \): Ratio of cache to use [%]
- CaSiEl: Cache Size in Elements
- ElPeIt: Elements Per Iteration
- ElPeRe: Elements Per Register

- Default or manually automatically or manually automatically detected
- Default or manually automatically or manually automatically detected
Derivation of tile sizes

- **L1-Tile-Size**: \( q^{L1} = \left\lfloor \rho \times \frac{\text{CaSiEl}}{\text{ElPeIt} \times \text{ElPeRe}} \right\rfloor \times \text{ElPeRe} \)

- **L2-Tile-Size**: \( q^{L2} = \frac{C_{L2}}{C_{L1}} \)

\( \rho \) \hspace{1cm} \text{Ratio of cache to use [%]} \hspace{1cm} \text{default or manually}

\text{CaSiEl} \hspace{1cm} \text{Cache Size in Elements} \hspace{1cm} \text{automatically or manually}

\text{ElPeIt} \hspace{1cm} \text{Elements Per Iteration} \hspace{1cm} \text{automatically detected}

\text{ElPeRe} \hspace{1cm} \text{Elements Per Register} \hspace{1cm} \text{automatically or manually}

\( C_{L1} \hspace{1cm} \text{L1-Cache size [KByte]} \hspace{1cm} \text{automatically or manually}

\( C_{L2} \hspace{1cm} \text{L2-Cache size [KByte]} \hspace{1cm} \text{automatically or manually}

**For nested loops with more than one block of statements, PluTo-SICA can assign an adapted size to each of those!**
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor:</strong></td>
<td>Intel® Xeon® CPU X5650 @ 2.67GHz</td>
</tr>
<tr>
<td><strong>Backend-Compiler:</strong></td>
<td>gcc 4.6 and icc 13.0</td>
</tr>
<tr>
<td><strong>Compiler opt level:</strong></td>
<td>-O3</td>
</tr>
<tr>
<td><strong>L1-Cache:</strong></td>
<td>32 KByte (data)</td>
</tr>
<tr>
<td><strong>L2-Cache:</strong></td>
<td>256 KByte</td>
</tr>
<tr>
<td><strong>SSE-Version:</strong></td>
<td>4.2</td>
</tr>
</tbody>
</table>
Testcodes - SCoPs

**matrix multiplication**

1. for(i=0; i<M; i++)
2. for(j=0; j<N; j++)
3. for(k=0; k<K; k++)

**correlation matrix algorithm**

1. /*Center and reduce the column vectors.*/
2. for(i = 1; i <= n; i++)
3. for(j = 1; j <= m; j++)
4. {
5.   data2[i][j] -= mean[j];
6.   data2[i][j] /= sqrt(n) * stddev[j];
7. }
8. /*Calculate the m*m correlation matrix.*/
9. for(j1 = 1; j1 <= m-1; j1++)
10. {
11.   symmat[j1][j1] = 1.0;
12.   for(j2 = j1+1; j2 <= m; j2++)
13.     {
14.       symmat[j1][j2] = 0.0;
15.       for(i = 1; i <= n; i++)
16.         symmat[j1][j2] += ( data2[i][j1] * data2[i][j2] );
17.       symmat[j2][j1] = symmat[j1][j2];
18.     }
19.   }

...L1-Cache tiling (gcc)

matrix multiplication \( (M, K = 189, N = 139233) \)

- \( j \)-loop is vectorized
- \( \rho = 1.0 \)

\[
C[i][j] = C[i][j] + alpha \times A[i][k] \times B[k][j];
\]

\[
q^{L_1} = \left[ \frac{\rho \times CaSiEl}{ElPelt \times ElPeRe} \right] \times ElPeRe
\]

\[
= \left[ \frac{1.0 \times 8192}{2 \times 4} \right] \times 4 = 4096
\]
...L1-Cache tiling (gcc)

matrix multiplication \((M, K = 189, N = 139233)\)

\[
C[i][j] = C[i][j] + \alpha * A[i][k] * B[k][j];
\]

- \(j\)-loop is vectorized
- \(\rho = 0.9\)

\[
q^{L1} = \left\lfloor \frac{\rho \cdot \text{CaSiEl}}{\text{EIPelt} \cdot \text{EIPeRe}} \right\rfloor \cdot \text{EIPeRe}
\]

\[
= \left\lfloor \frac{0.9 \cdot 8192}{2 \cdot 4} \right\rfloor \cdot 4 = 3684
\]

OPTIMUM
...L1-Cache tiling \((gcc)\)

**correlation matrix algorithm** \((M = 11923 \ N = 89)\)

- there are 6 statements in the SCoP
- the tile sizes for the statements vary \((\rho = 1.0)\)
  - \(q^{L1}(S1, S2) = 2728\)
  - \(q^{L1}(S3) = 8192\)
  - \(q^{L1}(S4, S5) = 4096\)
  - \(q^{L1}(S3) = 1\)
    - (because of different access functions)

**OPTIMUM**
Choosing the outermost loop for the second level tiling leads to performance improvement (any other one does not). This choice keeps changes to the inner loops as rare as possible (→ good for the prefetcher).

The estimated optimal value \( q^{L2} = \frac{256}{32} = 8 \) for the second level corresponds perfectly to our empirical evaluation. We verified our approach for \( q^{L1} \) and \( q^{L2} \) by analysing and measuring:

- More codes
- Combined with several Pluto transformations and
- Different amounts of data to be loaded (ElPeIt)

The relation between hardware, access structure and (near) optimal tile size in Pluto-SICA was confirmed.
...performance counter measurements *(gcc)*

**matrix multiplication**  \((M, K = 189 \ N = 139233)\)

- **L2 cache miss rate**
  - source 35.76%, *PluTo (def.*) 26.41%, *SICA* 4.78%

- **rate of vectorization**
  - source 0.00%, *PluTo (def.*) 71.53%, *SICA* 99.69%

**correlation matrix algorithm**  \((M = 11923 \ N = 89)\)

- **L2 cache miss rate**
  - source 19.17%, *PluTo (def.*) 33.87%, *SICA* 5.82%

- **rate of vectorization**
  - source 0.00%, *PluTo (def.*) 99.57%, *SICA* 99.67%
...performance counter measurements (**icc**)

**matrix multiplication**  \((M, K = 189 \ N = 139233)\)

- **L2 cache miss rate**
  - source 04.02%, **PluTo** (def.) 25.34%, **SICA** 5.75%

- **rate of vectorization**
  - source 83.66%, **PluTo** (def.) 71.28%, **SICA** 99.68%

**correlation matrix algorithm**  \((M = 11923 \ N = 89)\)

- **L2 cache miss rate**
  - source 22.42%, **PluTo** (def.) 38.12%, **SICA** 5.85%

- **rate of vectorization**
  - source 99.90%, **PluTo** (def.) 99.66%, **SICA** 99.90%
...performance benchmarks (*gcc*)

**matrix multiplication** \((M = N = K)\)

**correlation matrix algorithm** \((M = N)\)
...performance benchmarks \((icc)\)

**matrix multiplication** \((M = N = K)\)

**correlation matrix algorithm** \((M = N)\)
Average Speedups

- PluTo-SICA detects (near) optimal values for the tile sizes
- The (already well imposed) performance of PluTo is significantly improved by our extension (for vectorizable codes)
- Averagely archived speedups:

<table>
<thead>
<tr>
<th></th>
<th>gcc</th>
<th>matrix multiplication</th>
<th>correlation matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>PluTo (def.)</td>
<td>11.14</td>
<td>4.47</td>
<td></td>
</tr>
<tr>
<td>SICA</td>
<td>20.05</td>
<td>8.89</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>icc</th>
<th>matrix multiplication</th>
<th>correlation matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>PluTo (def.)</td>
<td>1.01</td>
<td>3.73</td>
<td></td>
</tr>
<tr>
<td>SICA</td>
<td>1.31</td>
<td>7.54</td>
<td></td>
</tr>
</tbody>
</table>

Table: Average speedups (coarse grain)
PluTo-SICA
- performs a \textit{hardware-related} code optimization
- specifically targeted to \textit{vectorization}
- enables a \textit{parameter independent} vectorization by \textit{gcc}
- determines (near) optimal \textit{tile sizes}
- and achieves \textit{performance improvement} for both \textit{gcc} and \textit{icc}

- \textit{icc} mainly profits from the \textit{improved cache behavior}
- \textit{gcc} additionally profits from strongly \textit{increased rates of vectorization}

Further studies showed, that
- drawbacks of static all-dimensional tiling rises for deeply nested loops
- whereas our extension can handle those cases

\Rightarrow \textbf{The SICA approach can greatly support recent compilers at vectorization}

verified for several scientific codes (partially from polybench)
Future work

- examine the behavior of our extension on further codes (part. done)
- examine the performance of our approach combined with automatic parallelization
  - already achieved very promising results
  - tiled matrix multiplication

<table>
<thead>
<tr>
<th></th>
<th>gcc serial</th>
<th>gcc parallel</th>
<th>icc serial</th>
<th>icc parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>PluTo (def.)</td>
<td>11.52</td>
<td>15.94</td>
<td>1.01</td>
<td>7.18</td>
</tr>
<tr>
<td>SICA</td>
<td>20.61</td>
<td>217.94</td>
<td>1.30</td>
<td>13.27</td>
</tr>
</tbody>
</table>

Table: Average speedups (coarse grain)

- development of hardware related tile size selection strategies for non-vectorizable codes
- combination of our approach with optimizations for 1-strided accesses
- internally we are porting our developments to PoCC to use them in LLVM
Thank you for your attention!

Questions?