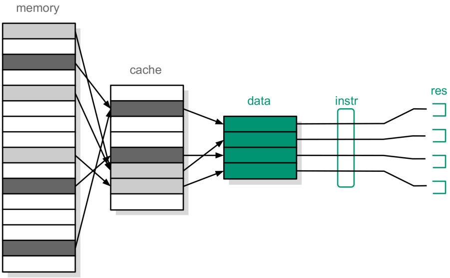
Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Code-Transformation

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

Fraunhofer SCAI & University of Cologne



Topics

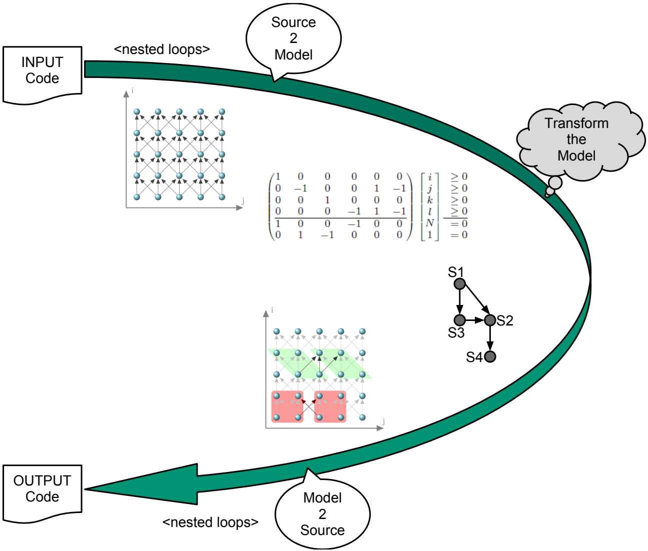
- automatic **vectorizers** are implemented in recent compiler frameworks
- but the archieved **performance** due to this potential varies regarding
  - the compilers **transformation** potential
  - the ability to handle ‘any’ **parameter** constallations
  - the memory **access patterns** and
  - the 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 archieve fast access to the streamed data.


INPUT
Code
<nested loops>
Source
2
Model
Transform
the
Model
S1
S2
S3
S4
OUTPUT
Code
<nested loops>
Model
2
Source

- 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 **archieve** 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

```
for(i=0; i<M; i++)
 for(j=0; j<N; j++)
  for(k=0; k<K; k++)
```

⇒

```
for(ii=0; ii≤floor(M-1,32); ii++)
 for(jj=0; jj≤floor(N-1,32); jj++)
  for(kk=0; kk≤floor(K-1,32); kk++)
   for(i=32*ii; i≤min(M-1,32*ii+31); i++)
    for(j=32*jj; j≤min(N-1,32*jj+31); j++)
     for(k=32*kk; k≤min(K-1,32*kk+31); k++)
```

PluTo-SICA L1 tiling

vectorizable

```
for(i=0; i<M; i++)
 for (j=0; j<N; j++)
  for(k=0; k<K; k++)
```

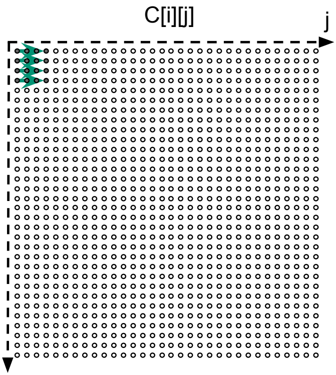
⇒

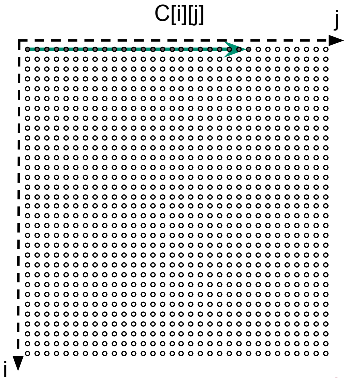
```
for(i=0; i<M; i++)
 for (jj=0; jj≤floor(N-1, q^L1); jj++)
  for(k=0; k<K; k++)
   for (j=q^L1*jj; j≤min(N-1, q^L1*jj+(q^L1 - 1)); j++)
```

Comparison of the approaches

accesses of the vectorized loop

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





PluTo (default) vs. PluTo-SICA

PluTo (default) L1 + L2 tiling

```
for(i=0; i<M; i++)
 for(j=0; j<N; j++)
  for(k=0; k<K; k++)
```

⇒

```
for(iii=0; iii≤floor(M-1,256); iii++)
 for(jjj=0; jjj≤floor(N-1,256); jjj++)
  for(kkk=0; kkk≤floor(K-1,256); kkk++)
   for(ii=8*iii; ii≤min(floor(M-1,32),8*iii+7); ii++)
    for(jj=8*jjj; jj≤min(floor(N-1,32),8*jjj+7); jj++)
     for(kk=8*kkk; kk≤min(floor(K-1,32),8*kkk+7); kk++)
      for(i=32*ii; i≤min(M-1,32*ii+31); i++)
       for(j=32*jj; j≤min(N-1,32*jj+31); j++)
        for(k=32*kk; k≤min(K-1,32*kk+31); k++)
```

PluTo-SICA L1 + L2 tiling

```
for(i=0; i<M; i++)
 for (j=0; j<N; j++)
  for(k=0; k<K; k++)
```

vectorizable ⇒

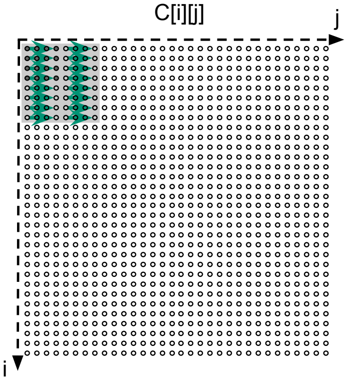
```
for(ii=0; ii≤floor(M-1,q^L2); ii++)
 for (jj=0; jj≤floor(N-1,q^L1); jj++)
  for(k=0; k<K; k++)
   for(i=q^L2*ii; j≤min(N-1,q^L2*ii+(q^L2 - 1)); i++)
    for (j=q^L1*jj; j≤min(N-1,q^L1*jj+(q^L1 - 1)); j++)
```

Comparison of the approaches

accesses of the vectorized loop

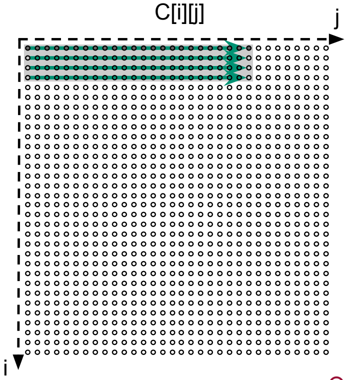
MEMORY SPACE

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

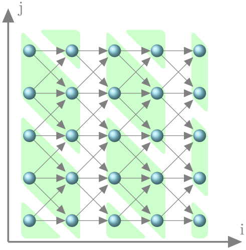
Issue

Support the compiler to perform calculations through the SIMD registers.

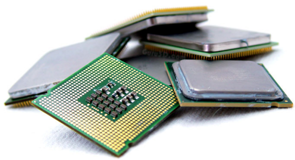
Issue

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

MODELL



HARDWARE



Goals of the SICA approach

Issue

Support the compiler to perform calculations through the SIMD registers.

Issue

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

Solution

register fitting *tile sizes*

Solution

cache adapted *tile sizes*

$\rightarrow$ **parameter-independent** vectorization

$\rightarrow$ **good load** through cache hierarchie

Derivation of tile sizes

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

CaSiEl	**Ca**che **Si**ze in **El**ements	automatically or manually
ElPeIt	**El**ements **Pe**r **It**eration	automatically detected

Derivation of tile sizes

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

ρ	Ratio of cache to use [%]	default or manually
CaSiEl	**Ca**che **Si**ze in **El**ements	automatically or manually
ElPeIt	**El**ements **Pe**r **It**eration	automatically detected

Derivation of tile sizes

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

ρ	Ratio of cache to use [%]	default or manually
CaSiEl	**Ca**che **Si**ze in **El**ements	automatically or manually
ElPeIt	**El**ements **Pe**r **It**eration	automatically detected
ElPeRe	**El**ements **Pe**r **Re**gister	automatically or manually

Derivation of tile sizes

- L1-Tile-Size : $q^{L1} = \left\lfloor \rho * \frac{\text{CaSiEl}}{\text{ElPeIt} * \text{ElPeRe}} \right\rfloor * \text{ElPeRe}$
- L2-Tile-Size : $q^{L2} = \frac{\mathcal{C}_{L2}}{\mathcal{C}_{L1}}$

ρ	Ratio of cache to use [%]	default or manually
CaSiEl	**Ca**che **Si**ze in **El**ements	automatically or manually
ElPeIt	**El**ements **Pe**r **It**eration	automatically detected
ElPeRe	**El**ements **Pe**r **Re**gister	automatically or manually
$\mathcal{C}_{L1}$	L1-Cache size [KByte]	automatically or manually
$\mathcal{C}_{L2}$	L2-Cache size [KByte]	automatically or manually

For nested loops with more than one block of statements, PluTo-SICA can assign an adapted size to each of those!

- *Processor:* Intel® Xeon® CPU X5650 @ 2.67GHz
- *Backend-Compiler:* *gcc 4.6* and *icc 13.0*
- *Compiler opt level:* -O3
- *L1-Cache:* 32 KByte (data)
- *L2-Cache:* 256 KByte
- *SSE-Version:* 4.2

Testcodes - SCoPs

matrix multiplication

```
for(i=0; i<M; i++)
 for(j=0; j<N; j++)
  for(k=0; k<K; k++)
   C[i][j] = C[i][j] + alpha*A[i][k]
       *B[k][j];
```

correlation matrix algorithm

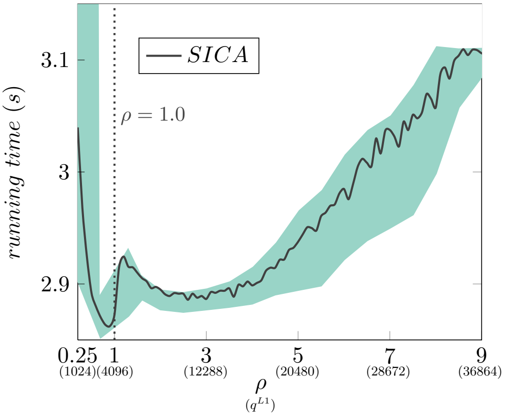
```
/*Center and reduce the column vectors.*/
for(i = 1; i <= n; i++)
 for(j = 1; j <= m; j++)
 {
  data2[i][j] -= mean[j];
  data2[i][j] /= sqrt(n) * stddev[j];
 }
/*Calculate the m*m correlation matrix.*/
for(j1 = 1; j1 <= m-1; j1++)
 {
 symmat[j1][j1] = 1.0;
 for (j2 = j1+1; j2 <= m; j2++)
  {
  symmat[j1][j2] = 0.0;
  for (i = 1; i <= n; i++)
   symmat[j1][j2] += ( data2[i][j1] *
       data2[i][j2]);
  symmat[j2][j1] = symmat[j1][j2];
 }
}
```

Fraunhofer SCAI

Institut für Informatik Universität zu Köln

...L1-Cache tiling (*gcc*)

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



- *j*-loop is vectorized
- $\rho = 1.0$

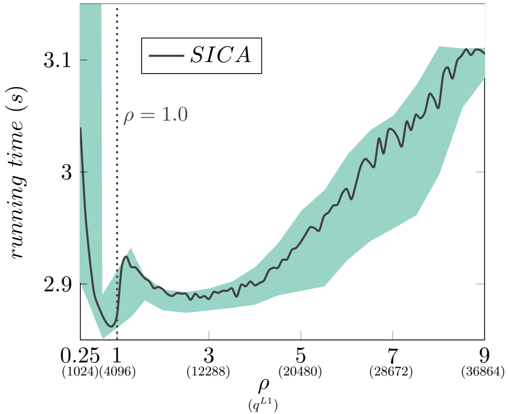
$C[i][\mathbf{j}] = C[i][\mathbf{j}] + alpha * A[i][k] * B[k][\mathbf{j}];$

$$q^{L1} = \left\lfloor \frac{\rho * \text{CaSiEl}}{\text{ElPelt} * \text{ElPeRe}} \right\rfloor * \text{ElPeRe}$$

$$= \left\lfloor \frac{1.0 * 8192}{2 * 4} \right\rfloor * 4 = \mathbf{4096}$$

...L1-Cache tiling (*gcc*)

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



- *j*-loop is vectorized
- $\rho = 0.9$

$C[i][\mathbf{j}] = C[i][\mathbf{j}] + alpha * A[i][k] * B[k][\mathbf{j}];$

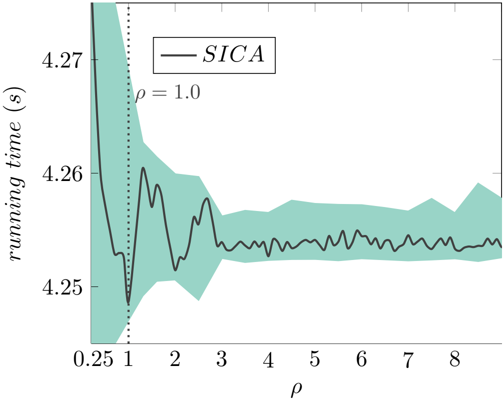
$$q^{L1} = \left\lfloor \frac{\rho * \text{CaSiEl}}{\text{ElPeIt} * \text{ElPeRe}} \right\rfloor * \text{ElPeRe}$$

$$= \left\lfloor \frac{0.9 * 8192}{2 * 4} \right\rfloor * 4 = \mathbf{3684}$$

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

...L1+L2-Cache tiling (*gcc*)

- 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 possbile** ($\rightarrow$ 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 (*ElPelt*)

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 ($M, K = 189$ $N = 139233$)

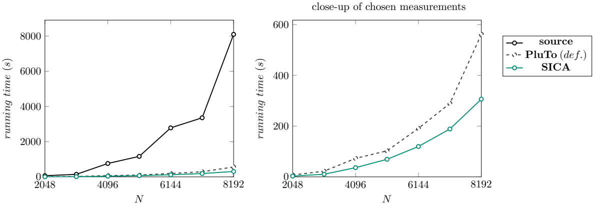
- **L2 cache miss rate**
  - *source* 04.02%, *PluTo (def.)* 25.34%, *SICA* 5.75%
- **rate of vectorization**
  - *source* 83.66%, *PluTo (def.)* 71.28%, *SICA* 99.68%

correlation matrix algorithm ($M = 11923$ $N = 89$)

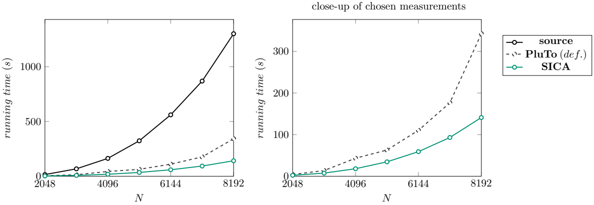
- **L2 cache miss rate**
  - *source* 22.42%, *PluTo (def.)* 38.12%, *SICA* 5.85%
- **rate of vectorization**
  - *source* 99.90%, *PluTo (def.)* 99.66%, *SICA* 99.90%

...performance benchmarks (*gcc*)

matrix multiplication ($M = N = K$)

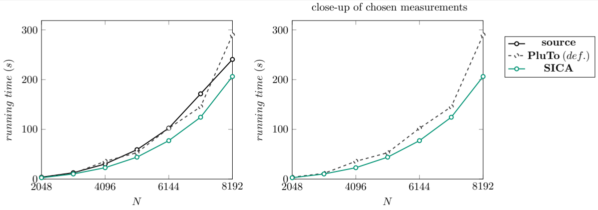


correlation matrix algorithm ($M = N$)

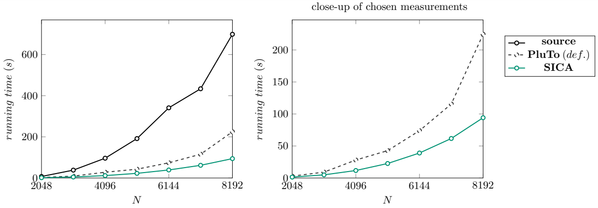


...performance benchmarks (*icc*)

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 archieved speedups:

gcc	matrix multiplication	correlation matrix
PluTo (def.)	11.14	4.47
SICA	20.05	8.89
icc	matrix multiplication	correlation matrix
PluTo (def.)	1.01	3.73
SICA	1.31	7.54

Table: Average speedups (coarse grain)

- PluTo-SICA
  - performs a **hardware-related** code optimization
  - specifically targeted to **vectorization**
  - enables a **parameter independent** vectorization by *gcc*
  - determines (near) optimal **tile sizes**
  - and archieves **performance improvement** for both *gcc* and *icc*

  - *icc* mainly profits from the **improved cache behavior**
  - *gcc* additionally profits from strongly **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

$\Rightarrow$ **The SICA approach can greatly support recent compilers at vectorization**

verified for several scientific codes (partially from polybench)

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

run on 12 cores

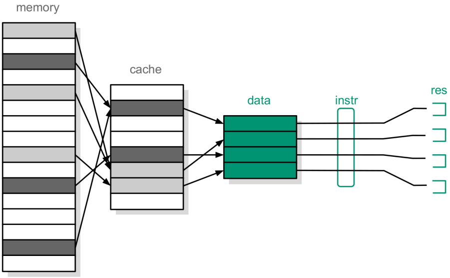
gcc	serial		parallel
PluTo (def.)	11.52	→ 1.38 →	**15.94**
SICA	20.61	→ 10.58 →	**217.94**

icc	serial		parallel
PluTo (def.)	1.01	→ 7.11 →	**7.18**
SICA	1.30	→ 10.17 →	**13.27**

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?