Hardware-Aware Automatic Code-Transformation to Support Compilers in Exploiting the Multi-Level Parallel Potential of Modern CPUs

Abstract
Modern compilers offer more and more capabilities to automatically parallelize code-regions if these match certain properties. However, there are several application kernels that, although rather simple transformations would suffice in order to make them match these properties, are either not at all parallelized by state-of-the-art compilers or could at least be improved w.r.t. their performance. This paper proposes a loop-tiling approach focusing on automatic vectorization and multi-core parallelization, with emphasis on a smart cache exploitation. The method is based on polyhedral code transformations that are applied as a pre-compilation step and it is shown to help compilers in generating more efficient parallel code-regions. It automatically adapts to hardware parameters such as the SIMD register width and cache sizes. Further, it takes memory-access patterns into account and is capable to minimize communication among tiles that are to be processed by different cores. An extensive computational study shows significant improvements in the number of instructions vectorized, cache miss rates, and running times for a range of application kernels. The method often outperforms the internal auto-parallelization techniques implemented into gcc and icc.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Optimization; B.3.2 [Memory Architectures]: Design Styles—Cache Memories; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—Parallel processors; SIMD processors

Keywords SIMD, cache, TSS, polyhedral model, tiling, code generation, auto-parallelization, vectorization, loop transformation

1. Introduction and Related Work
Tiling (or blocking) of nested loops is a well-known technique to improve on data locality and, if concurrency in terms of outer loop dimensions is possible, to perform automatic parallelization [31]. A set of rules that define which loops to tile and which tile sizes to create is called a tile size selection (TSS) strategy. TSS models can be roughly divided into static (purely compile-time) and empirical search based (partially or completely runtime) methods. A static TSS strategy uses a pre-designed set of rules to determine tile sizes for the loop nests of a given program, potentially taking the respective target architecture into account. In the empirical approaches, a TSS model is used in order to characterize good tilings and to reduce the search space for candidate tile sizes which are then to be evaluated on the target machine before a final decision is made.

Besides a general automation of TSS, multi-core parallelization and spatial cache optimization are classical targets of static as well as empirical approaches that are frequently combined with loop transformations [8]. However, common TSS approaches usually do not explicitly deal with an effective loop vectorization, even if they consider parallelization in general. Further, they are often restricted to one-level or all-level tilings and concentrate on the tile sizes only, neglecting the determination of a loop that is well-suited for vectorization. On the other hand, methods that deal with auto-vectorization usually provide no integrated approach to TSS. For example, Trifunovic et al. [30] present a framework to choose the best-suited loop for vectorization within a nest, but do not combine it with a TSS model. Similarly, Kong et al. [23] demonstrate how polyhedral transformations can be used to optimize for both, SIMD- and multi-core parallelism. However, again their approach does not provide any TSS. Coleman and McKinley [10] provide a simple mathematical model to compute tile sizes w.r.t. a pre-specified outer target loop. So while this approach provides a TSS, it does neither facilitate the generation of SIMD instructions nor is it capable of computing target loops for the tiling itself. A further research branch considers cache-miss equation analysis or dedicated cost models [1, 5, 8, 13, 17, 27]. Once more, these methods do not explicitly target vectorization and need to be manually equipped with hardware parameters. This is similar on the empirical side [3, 16, 21, 22, 26] where the respective approaches further require runtime information. There are as well approaches that try to combine both static and empirical methodologies. Shirako et al. [28] proposed an analytical model to limit the search space of empirical TSS approaches which is so far however restricted to one-level tilings only. Some of their ideas, especially to leave some loop dimensions unblocked and to tile a vectorizable loop into large blocks, are similar to those that we present. Yuki et al. [32]...
The main contribution of this paper is a fully implemented combination of loop transformations with a fast, automated, hardware-aware, and straightforward static TSS model that is explicitly dedicated to promote both vectorization and multi-core parallelization. It integrates the selection of a well suitable loop for vectorization with a level-one cache oriented tile size computation, and a communication-minimal second-level tiling for multi-core parallelization. The tile size computation also involves an estimation of the number of data elements to be loaded per loop iteration. The correlation between the computed tile sizes and performance is verified experimentally. For many practical application kernels, we obtain a measurably well-suited stream of data through the cache hierarchy and an increased rate of issued SIMD instructions that lead to improved running times and considerable multi-core speedups that we also compare to gcc’s and icc’s auto-parallelizers.

2. Background and Motivation

The exhaustive search space, the desired retargetability to different processor architectures, and the irregular access patterns and data dependencies in several application codes make automated TSS a challenging problem that can however have a significant impact on performance. Our approach emphasizes on the fact that modern CPUs offer multiple levels of parallelism that need to be exploited in order to reach their full speedup potential. However, in the SIMD context, it turns out that not every loop is vectorizable and, in the multithreading context, several tile sizes may be valid but lead to different amounts of necessary communication. Especially for deep loop nests, one may encounter significant performance disparities when tiling particular loops with different tilesizes, as will be discussed in this paper and as is well exemplified, e.g., in [32]. While compilers are evolving and offer the automatic parallelization of particular code-regions if these match certain properties, we will see that the performance of, e.g., gcc or icc can be improved considerably if loop transformations and a smart TSS strategy are carried out prior to compilation. Of course, our method may be internalized into a compiler as well and would then allow for an even more accurate analysis of the input source codes.

To support compilers in vectorizing a loop, it is usually advantageous to first make this loop innermost within its nest such that several blocks of iterations that are to be processed concurrently by one SIMD instruction are arranged contiguously. Consider a standard matrix multiplication code with single precision floating point data. It consists of a single statement in a three times nested loop:

\[
\begin{align*}
&\text{for } (i=0; i<M; i++) \\
&\text{for } (j=0; j<N; j++) \\
&\text{for } (k=0; k<K; k++) \\
&C[i][j] = C[i][j] + \alpha \times A[i][k] \times B[k][j];
\end{align*}
\]

Listing 1: Standard Matrix Multiplication

The innermost loop of this nest, the \(k\)-loop, carries dependencies w.r.t. the read- and write-accesses to \(C[i][j]\) in each of its iterations and is therefore not vectorizable. As opposed to that, both the \(i\)- and the \(j\)-loop are suitable for parallel execution. Interchanging the \(k\)- with the \(j\)-loop leads to a parallelizable innermost and therefore straightforwardly vectorizable \(j\)-loop. A subsequent static tiling of all three loops with a block size \(b\) results in blocks that address \(b \times b\) elements of each of the three matrices. For example, a block size of \(b = 32\) leads to inner blocks that address approximately \(3 \times 32 \times 32 = 3072\) floating point data elements (12kB). Even though such a tiling results in a reasonable high data reuse and fits into almost all L1 caches of today’s x86 processors, it also causes a lot of large jumps in memory after every \(32h\) iteration. Further, we recognized that if \(N \mod b\) is not compliant with (not a multiple of) the SIMD register width, some compilers produce ‘remainder’ tiles that are not vectorized [12]. The mentioned static tiling of all loops tendentially increases the number of such remainder tiles. Nevertheless, many early tiling approaches, and those that do not have specific other optimization targets than data locality or parallelization in general, carry out a static tiling of one or all loops into equally-sized blocks which is unlikely to perform well for varying applications and systems.

Instead, we propose to tile only those loops within a loop-nest that are relevant for the data to be processed in a vectorized manner, and to tile the vectorized loop with a relatively large tile size. While this strategy may or may not improve spatial locality (depending on whether the blocked loop index variable corresponds to one-strides in memory), it reduces the frequency of changes to the index variables of other loops, especially in the case of deeply nested loops. This leads to fewer remainder tiles that have either a small size, a non-SIMD-width compliant size, or both. Further, we may expect that it is beneficial for the successful prefetching of operands, if the innermost loop is executed for a relatively large number of iterations before the control flow leaves it and manipulates the other loops’ index variables. Another core aspect of our SICA (short for SIMD and cache) tiling is to automatically adapt the tile sizes to the given on-core cache levels, SIMD-register widths and the actual code structure using a straightforward polyhedral and analytical model. Our level-one approach blocks a vectorizable loop (in the matrix multiplication example, the \(j\)-loop) while leaving the other loops unblocked. This allows to use a relatively large tile size \(q^{d1}\) that promotes a number of iterations that suffices to be able to profit from prefetchers that recognize the access pattern in the tiled loop. At the same time, we make \(q^{d1}\) small enough such that the data addressed by the resulting blocks fits into the L1 cache. This first-level tiling is then complemented by an appropriate second-level tiling that (a) promotes to preload several complete L1 tiles into the L2 cache and (b) tends to minimize communication necessary between L2 tiles if targeting multi-core processors. Ideally, this strategy creates a pipeline of blocks to be processed in parallel.

The rest of this paper is organized as follows: In Sect. 3, we present our TSS strategy and indicate how it works on a single- as well as on a multi-core processor. We also give a first validation of the correlation between performance and the tile size selection. To point out that our approach has the claimed effects concerning the number of vectorized instructions and cache performance, we carry out performance counter measurements for two selected codes on a single-core in Sect. 4.1. These are followed by performance benchmarks for the same codes on multiple cores in Sect. 4.2 and extended experiments using a large set of benchmark codes in Sect. 4.3. We as well compare our method to the state-of-the-art compiler-internal auto-parallelization techniques implemented into gcc and icc. Finally, we give some insights into their behavior in Sect. 4.4 and an outlook to future work.

3. SICA: SIMD- and cache-aware code transformation

Our SICA transformation and tiling approach is implemented as an extension to PluTo that we shortly address first. After that, we describe how our TSS model works and how it is integrated into the PluTo framework.

3.1 PluTo

PluTo [6, 7] is an academic source-to-source compiler framework that is able to generate tilings based on polyhedral optimization [2,
9, 11, 14, 15, 31]. More precisely, it can compute legal tilings for a given source code (resp. its statements’ dependence polyhedra) through dependence analysis and integer linear programming. For each dimension of a statement’s iteration space, it generates candidates for a so-called tiling hyperplane respecting the underlying dependence structure. The selected hyperplanes then altogether determine the shape of the later tiles and likewise define the transformation of the iteration space. In particular, our approach exploits that PluTo is already able to detect vectorizable loops. In addition, the polyhedral model permits us to estimate the communication overhead of a particular tiling and, by taking these estimates into account and heuristically selecting the best feasible tile-shapes, to minimize it. A more comprehensive description on how valid tiling hyperplanes are generated in PluTo and selected within SICA is given respectively in [6] and [12]. Fig. 2 shows the overall workflow of SICA’s integration into PluTo.

While PluTo involves all core functionalities to perform a tiling, it was not designed to provide elaborate automated TSS strategies. User-specification of tile sizes w.r.t. the input source code is possible but shall not be considered, as this potentially misleading because loops may be interchanged or otherwise manipulated by PluTo. If no tile sizes are provided by the user, PluTo uses a fallback first level tile size of 32 and additionally a second level tile size of 8 for all loops. Such static tiling strategies are widely used as they are relatively easy to implement while providing a reasonable performance for many codes on recent hardware. We would like to remark that we do not consider this as a TSS that we compare against. Rather, incorporating PluTo’s default tiling into our experiments is intended to validate that our adaptive tiling approach indeed yields improvements over a completely static one.

3.2 The SICA TSS Strategy

In general, we assume to be given an input source code that may comprise several loop nests containing a set of statements \( S \) that perform array accesses. Initially, Clan [4] is invoked to analyze the code and to set up its polyhedral representation including functions that describe its array accesses. Then, PluTo considers each statement individually and decides whether a statement’s accesses and surrounding loops are suitable for vectorization or can be made to be suitable by transformations such as loop interchanges. Loop fusions then regroup those statements whose transformed iteration spaces coincide into statement-blocks. Each of the resulting loop nests may now have its own vectorized and therefore tiled loop dimension.

Of central interest is the number \( \mathcal{E} \) of new operands that must be loaded per iteration of each vectorized loop. The way we derive a good estimate on \( \mathcal{E} \) is best described using an illustrative example. Listing 2 displays a twice nested loop with two statements \( E \) and \( S \) that describe its array accesses. Then, PluTo considers each state-ment and surrounding loops to determine if it can be tiled in a way that the resulting loop is vectorized, two new iterations are detected by Clan [4] and are subsequently passed to PluTo. If no tile sizes are provided by the user, PluTo uses a fallback first level tile size of 32 and additionally a second level tile size of 8 for all loops.

Listing 2: Transformed loop nest before tiling

For our tile size estimation, we need to analyze the accesses in the loop nest of the transformed code, so we have to apply PluTo’s transformations also to the access functions obtained from Clan. In general, we consider accesses \( Z \) being the access \( Z \) resulting after applying PluTo’s transformations \( \mathcal{F} \) (cf. Fig. 1). In our example, the accesses are \((i_0, i_1)\) on array \( b \) (twice), and \((i_1, i_1 + 1)\) and \((i_1, i_1 + 2)\) on array \( a \). We then analyze in which way the vectorized loop iterator \( i_v \) is used to access the different arrays’ dimensions in the transformed space. If an access does not refer to \( i_v \), the respective data is constant under \( i_v \)’s iterations and no new operands need to be fetched for it. Conversely, if accesses refer to \( i_v \) in one or more dimensions, new data needs to be loaded. However, we now emphasize on the detection of several re-use patterns in order to improve the quality of our estimation. For example, operands that are accessed multiple times within one iteration or reused in a subsequent iteration should be loaded only once. Similarly, if multiple accesses to the same array dimension differ only in constant terms (say, e.g., \( a[i_1][i_1 + 1] + a[i_1][i_1 + 2] \)), then it is reasonable to assume that only one operand needs to be fetched per iteration, since it can afterwards reside in the cache and be reused in a subsequent iteration. Using these and similar rules, we store the set of different accesses to a particular array \( Arr_j \) as a linked list \( AAC_k[Arr_j] \) (cf. Fig. 1) for each statement \( S_i \in S \) of the statement-block.

For each of the statement-blocks, our code analysis further determines the data-type size \( \mathcal{D} \) (in bytes). Apart from that, the SIMD-register width \( \mathcal{R} \) (in bits) of the system’s processor and the sizes (in KBytes) of the L1 and L2 cache, \( \mathcal{C}_{L1} \) and \( \mathcal{C}_{L2} \) respectively, are detected by CPUID [18] calls. These values permit us to compute, for each of the statement-blocks, how many iterations need to be tiled such that all required operands of an inner iteration ideally fit into the L1 cache at once. If needed, we then alter the corresponding block size to a multiple of the register width so that all iterations can be equally distributed among the SIMD registers.

Putting everything together, the estimated first cache level tile size \( q_{1blt}^L \) for a particular statement-block with \( \mathcal{E} \) new operands to be loaded per iteration of the vectorized loop is:

\[
q_{1blt}^L = \left\lceil \frac{\mathcal{C}_{L1} \times 8192}{\mathcal{R} \times \mathcal{D}} \right\rceil \times \mathcal{R} \div 8 \times \mathcal{D} \quad (1)
\]

A stepwise description of the L1 tile size calculation can be found in [12]. In the future, the process could be made even more accurate by using an exact convex set calculation. The tile quantity for the second level tiling, that is always applied to the outermost loop of the nest in our approach, is then simply obtained from the ratio of the two cache sizes, i.e., \( q_{2}^L = \frac{\mathcal{C}_{L2}}{\mathcal{C}_{L1}} \). This way, \( q_{2}^L \) corresponds to the number of L1-optimized blocks that fit into the L2 cache.

Example Consider again the standard matrix multiplication with single precision floating point data. It has only a single statement \( C[i][j] = C[i][j] + A[i][k] \times B[k][j] \). If the j-loop is vectorized, two new data elements need to be loaded per j-iteration, namely \( C[i][j] \) and \( B[k][j] \). This leads to the following SICA L1 tile size for a system with 32 KByte of L1 cache and 128 Bit SSE registers:

\[
q_{1blt}^L = \left\lceil \frac{32 \times 8192}{128 + 2} \right\rceil \times \frac{128}{8 + 4} = 4096
\]

With 256 KByte of L2 cache, \( q_{2}^L \) evaluates to \( \frac{256}{32} = 8 \).

3.3 Correlation between our TSS strategy and performance

To experimentally validate the claimed correlation between (our) tile sizes and performance, we consider the matrix multiplication as a representative for a relatively simple code structure and the correlation matrix calculation that includes several non-perfectly nested loops and math function calls. For this experiment we chose highly irregular matrix dimensions: \( 189 \times 139233 \) for the matrix multiplication and \( 11923 \times 89 \) for the correlation matrix algorithm. This way many L1 tiles for the vectorized loop dimension can be generated to point out their effects at the best while keeping the overall runtime moderate. Further, the odd dimensions lead to a certain portion of only partially filled ‘remainder’ tiles.
Data: Transformation-Matrices $T_S$ from PluTo and Array-Access-Functions from Clan, $\rho$, $D$, $R$

Result: Cache-specific tile sizes $q_{L1}^i$ for each statement-block and $q_{L2}^i$

begin

/* create list of transformed accesses per array of each statement */
foreach Statement $S_i$

| foreach Array $Arr_j$

| $AAC_S[Arr_j] = \emptyset$;

| foreach Access $Z$ in $S_i$ on Array $Arr_j$

| extract access of a vectorized loop $Z_i$

| transform access $\tilde{Z}_i = \mathcal{F}_S(Z_i)$

| if $\tilde{Z}_i \notin AAC_S[Arr_j]$ /* add it to the list */

| $AAC_S[Arr_j] \leftarrow AAC_S[Arr_j] \cup \tilde{Z}_i$

/* find and model statement-blocks */

/* calculate different relevant access for each block of statements */

foreach Block $Blk$ of statements $S_i$ in the transformed nest

| foreach Array $Arr_j$

| $AAC_{L, S}[Arr_j] = \bigcup_i AAC_S[Arr_j]$

$E_{Blk} = \#\text{Accesses}(Blk)$

$E_{Blk} = \sum_{all Arr_j} |AAC_{L, S}[Arr_j]|$

read $E_{L1}$ and $E_{L2}$ from hardware /* in KB */

$q_{L1}^{elm} = \frac{E_{L1} \cdot 1024}{8 \times \mathcal{F}}$; /* Cache size in elements */

$q_{L2}^{elm} = \frac{E_{L2}}{8 \times \mathcal{F}}$; /* Register size in elements */

foreach Block of statements $Blk$

$q_{L1}^i = \frac{E_{Blk}}{E_{L1}} \cdot q_{L1}^{elm}$

$q_{L2}^i = \frac{E_{Blk}}{E_{L2}} \cdot q_{L2}^{elm}$ /* L2-Tiling */

end

Figure 1: Cache-specific tile size estimation

We introduce a parameter $\rho$ in equation (1) to be able to easily alter the calculated tile size estimates while keeping them to be multiples of the register width without the need to change the size of the input matrices.

$q_{L1}^i = \left[ \rho \cdot \frac{E_{L1} \cdot 8192}{8 \times \mathcal{F}} \right] \cdot \frac{\mathcal{F}}{8 \times \mathcal{F}}$

Fig. 3 shows the resulting running times when scaling $q_{L1}^i$ by $\rho$ in 0.01-steps for a one-level tiling of the two source codes on a single core of the test system described in the previous example.

The runtimes of the resulting 1000 source codes are within the shadowed area and the line is a cubic interpolation of them. For both codes, our estimated L1 tile size (dotted line) is near-optimal.

These results and further measurements [12] support our idea that it is essential to make the tile size a multiple of the register width and to take the cache sizes as well as the data-access patterns into account in the manner described.

Figure 2: SICA transformation process in PluTo

Figure 3: Impact of the tile size on the running time with gcc
3.4 PluTo-SICA on multiple cores

Clearly, when moving from SIMD parallelism to parallel multi-core execution of tiles, the hard constraints do not change: Modifying the iterations’ execution order by the generated tiling and the corresponding transformations (e.g. loop interchange, fusion, skewing), as well as by their potentially parallel execution, must not harm any dependencies. However, when executing tiles on different cores, communication of results may be necessary from one tile to the other (and therefore from one core’s memory hierarchy to the other’s). To achieve a parallel multi-core execution with reduced communication overhead, we combine our SICA tiling strategy with the automatic parallelization mechanisms implemented in PluTo. Communication is minimized heuristically based on the dependence structure (resp. the dependence polyhedra) of the code. Hyperplanes that are valid regarding the dependence structure may induce different amounts of data that need to be communicated between two tiles. Using integer linear programming, the PluTo core iteratively (meaning for one loop dimension after the other) generates hyperplanes that minimize the amount of data communication, or, visually interpreted (cf. Fig. 5), minimize the number of crossings of dependence arrows with tiling hyperplanes [6].

After the tiling hyperplane generation step, a parallel schedule for our cache-optimized tiles is determined that distributes them across the cores by inserting the respective OpenMP-pragmas.

Fig. 6 shows speedup values for the standard matrix multiplication code obtained on our test system (cf. Sect. 4) compiled with gcc and icc.

The results indicate that our approach leads to a near-linear scaling behavior for this code on one socket (up to 8 threads) and a reasonable additional speedup when creating 16 threads to be distributed among all the cores on both CPUs.

4. Benchmarks and Discussion

We carry out experiments in three steps. First, we consider a restricted set of target source codes and input sizes for performance counter measurements, validating the effects of our vectorization-oriented tiling. Then, we extend the set of input sizes in order to evaluate the full (vectorization and multi-core) speedup potential of our method. In the third step, we look at a larger set of target source codes from different domains.

We executed our benchmarks on a dual socket system equipped with two 8-core Intel® Xeon® E5-2660 (2.20 GHz) processors comprising 16 physical cores and AVX, operating on Scientific Linux 6.4 (Carbon) with gcc 4.9.1 and icc 14.0.2. All codes calculate with double precision floating point data and are compiled with optimization level -O3 that includes auto-vectorization.

The experiments are carried out for (a) the original source code, (b) the two-level default tiling by PluTo (see the discussion in Sect. 3.1) (options: --tile --12tile --prevector [--parallel]) and (c) the tiling computed by our SICA extension (additional option: --sica). The performance benchmarks in Sect. 4.2 and 4.3 are additionally applied on the original source code with gcc’s and icc’s internal auto-parallelizers. These were activated by the -ftree-parallelize-loops=16 (advising gcc to create 16 OpenMP threads, one per physical core) and by the -parallel (icc) options. The auto-parallelizer in gcc is based on the integration of the Graphite framework [24] that also uses polyhedral transformation techniques. In comparison to that, icc directly analyzes the dataflow in loops to determine which loops can be safely and efficiently executed in parallel [19].

4.1 Single-Core Performance Counter Evaluation

In this section, we present our results of PAPI [29] performance counter measurements for the matrix multiplication and the correlation matrix algorithms. These counters allow us to investigate the impact of the SICA tiling and to better explain the running times obtained in the other benchmarks. To get depictable results and for similar reasons as already explained in Sect. 3.3, we needed to fix the input matrix sizes and stuck to the same irregular dimensions. Here, the relatively high number of ‘remainder’-tiles for the vectorized loop is suitable to point out the advantages of our adaptive TSS approach over a pure static one.

In particular, we focused on the effects concerning the L2 cache miss rate and the rate of introduced SIMD instructions. The L2 cache, being the last on-core cache level, is crucial for our intended
4.1.1 Matrix multiplication

When using gcc, both, the SICA and PluTo’s default tiling lead to a significantly better performance of the matrix multiplication kernel (cf. Tab. 1). While the original source code cannot be vectorized by gcc at all, the tiled versions enable the issue of SIMD instructions. In case of the SICA tiling, in fact nearly all instructions are vectorized. Tiling only two instead of all loops results in fewer cases where the control flow enters the innermost loop with ‘remainders’ of iterations that cannot be vectorized by gcc. Further, our larger tile size for the vectorized loop corresponds to many predictable accesses to the innermost (here linearly stored) array dimension. Both tiled versions lead to a smaller L2 cache miss rate whereas the reduction is much stronger for the fitted tile sizes calculated by the SICA extension. For icc, its internal optimizations applied to the original source code perform better than when applied to PluTo’s default tiled version. This is especially true for the L2 cache miss rate. It is also marginally better as when applied to the fitted SICA tile sizes. However, with the SICA tiling given as input, icc is able to generate the fastest code since it can again turn nearly every instruction into a SIMD instruction.

4.1.2 Correlation matrix

For the correlation matrix, the source codes from PluTo with and without the SICA tiling both improve the running time w.r.t. to the original version when passed to gcc, as is also shown in Tab. 1. Even more, both tilings lead to a vectorization of almost all of the code’s instructions. However, whereas PluTo’s tiling leads to an increase of L2 cache misses compared to the original source code, the SICA tiling leads to a decrease which explains the better running time. Like in case of the matrix multiplication, PluTo’s default tiling does not lead to a better running time compared to the original source code when compiled with icc. The results regarding the L2 cache miss rate are similar to the gcc-experiment. However, icc is able to fully vectorize the original code and there is nearly no difference in the rate of vectorization using any of the three codes as input.

4.2 Performance benchmarks

To show that our approach works not only for special irregular inputs and also enables compilers to derive additional multi-core speedups, we now vary the matrix sizes \( N = \{2048, 3072, 4096, 5120, 6144, 7168, 8192\} \). Fig. 7 compares the running times obtained for the generated source codes using PluTo’s default and the SICA tiling (both automatically parallelized by inserted OpenMP pragmas) with those achieved by the compilers’ internal auto-parallelizers.

We want to distinguish between speedups gained by a better cache usage, prefetching and the increased rate of vectorization on each core on the one hand, and the additionally achieved multi-core speedup on the other. Tab. 2 depicts the geometric means of the speedups of all input sizes gained by the single-core codes as well as the multi-core codes for both compilers. The additional multi-core speedup is shown upon the arrows.

<table>
<thead>
<tr>
<th>code version</th>
<th>matrix multiplication</th>
<th>correlation matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc</td>
<td>icc</td>
</tr>
<tr>
<td>running time (s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PluTo (def.)</td>
<td>24.97 s</td>
<td>1.99 s</td>
</tr>
<tr>
<td>SICA</td>
<td>3.69 s</td>
<td>3.03 s</td>
</tr>
<tr>
<td>vectorized instructions (%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PluTo (def.)</td>
<td>0.00 %</td>
<td>83.66 %</td>
</tr>
<tr>
<td>SICA</td>
<td>71.53 %</td>
<td>71.28 %</td>
</tr>
<tr>
<td>L2 cache miss rate (%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PluTo (def.)</td>
<td>26.41 %</td>
<td>25.34 %</td>
</tr>
<tr>
<td>SICA</td>
<td>4.78 %</td>
<td>5.75 %</td>
</tr>
</tbody>
</table>

Table 1: PAPI performance counter results on a single-core

<table>
<thead>
<tr>
<th>gcc</th>
<th>matrix multiplication</th>
<th>correlation matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>serial</td>
<td>parallel</td>
</tr>
<tr>
<td>PluTo (def.)</td>
<td>9.74</td>
<td>94.21</td>
</tr>
<tr>
<td>SICA</td>
<td>11.74</td>
<td>124.53</td>
</tr>
<tr>
<td>AUTO</td>
<td>5.01</td>
<td>0.83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>icc</th>
<th>matrix multiplication</th>
<th>correlation matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>serial</td>
<td>parallel</td>
</tr>
<tr>
<td>PluTo (def.)</td>
<td>0.88</td>
<td>7.88</td>
</tr>
<tr>
<td>SICA</td>
<td>1.08</td>
<td>10.58</td>
</tr>
<tr>
<td>AUTO</td>
<td>34.24</td>
<td>6.34</td>
</tr>
</tbody>
</table>

Table 2: Geometric mean of performance benchmarks’ speedups

One can see that, in contrast to PluTo’s default tiling, the SICA tiling leads to a smooth scaling behavior (cf. the close-ups in Fig. 7). It as well reduces the running times across the spectrum of input sizes with average speedups of up to 11.74 on a single-core and resulting multi-core speedups of up to 124.5 (i.e. including a factor of 10.61 based on the parallel execution).

While the auto-parallelization of gcc is able to speed up the matrix multiplication code, it actually slows down the correlation matrix code’s execution. In contrast to that, the icc’s auto-parallelization was successful in both cases with an outstanding speedup for the matrix multiplication code. However, this result stems from the fact that icc’s -O3 -parallel option includes the -opt-matmul flag that enables the automatic detection of matrix multiplication kernels. The compiler then replaces these kernels by an optimized threaded library function instead of performing a normal code-transformation [19]. More insights into the requirements for this replacement and its utilization is given in Sect. 4.4.

As the correlation matrix code consists of multiple deeply nested loop nests whereas not all of them contain a parallelizable as well as a vectorizable loop at the same time, the speedups for this code are accordingly smaller.
### 4.3 PolyBench Benchmarks

Finally, we like to demonstrate how our approach performs for a more diverse set of target source codes. We made experiments for all source codes from PolyBench 3.2 [25] that the PluTo core detected to be vectorizable. The resulting set consists of codes from different fields of application: linear algebra kernels (2mm, 3mm, atax, biqc, dotigen, gemm, gemver, syrk, syr2k, syrk), linear algebra solvers (gramschmidt), datamining (correlation, covariance) and stencils (adi). Details about the applications’ characteristics can be found on the PolyBench homepage [25].

Concerning the input sizes, we chose the predefined ‘standard’ configuration from PolyBench to make our results well comparable to others. In contrast to our benchmarks from Sect. 4.2, the loop-ranges and input sizes of these standard inputs are rather small, e.g., ‘gemver’ derives the matrix product of 1024 × 1024 matrices. As our approach profits from an increase in the ratio of tiles that are successfully prefetched to the L1 cache after recognizing the access pattern, it is impossible to obtain the best possible speedups in this experiment. The inputs are so small that there will be only a single tile w.r.t. the inner loop dimension for many of the application kernels.

For all of them, Fig. 8 and Fig. 9 show the detailed speedups of both generated parallel and two-level tiled versions as well as of the auto-parallelized codes compared to the original ones. Fig. 10 additionally depicts the geometric mean of all the speedups.

It can be observed that the impact of the transformations applied by PluTo with and without the SICA extension is much larger for gcc than for icc. For this reason, the individual and average speedups of both versions are higher when compiled with gcc. To allow for a comparison of the speedups, one has to take the running times of the original source codes into account (cf. Fig. 11). As an example, the original 2mm application runs 4.41 times faster when compiled with icc than its pendant compiled with gcc. Therefore, the 2mm SICA code compiled with icc and obtaining a speedup of 7.9 still runs more than twice as fast in total as the SICA version compiled with gcc with a speedup of 15.4.

The parallel SICA code variants result in remarkable speedups for almost every code-compiler-combination in our benchmarks. On average, it exceeds the performance of the statically tiled parallel code by a factor of almost three with gcc and still almost by a factor of two with icc. Fig. 10 underlines that both, the optimized serial codes as well as the multi-core codes with SICA, lead to better performance than the ones with the static tiling.

Even though the auto-parallelizer within icc works for many of the prospected codes, its average speedup of 2.42 is even smaller than the one obtained with the static tiling generated by PluTo (3.11) and by far smaller than SICA’s average speedup of 5.86 (cf. Fig. 10). Furthermore, two codes are considerably slowed down by applying the auto-parallelizer of icc in its standard setup. However, the auto-parallelizer of gcc was not able to gain any speedups for any of the PolyBench codes at all.

### 4.4 Some insights into the behavior of the auto-parallelizers

Two peculiarities in our results concerning the matrix multiplications raised our attention and asked for even more investigations. In Fig. 8, we see that gcc’s auto-parallelizer could not parallelize any of PolyBench’s codes at all. In particular, it was not able to parallelize the gemm matrix multiplication kernel. Nevertheless, it was able to parallelize PluTo’s matmul code sample from Sect. 4.2. To us, this appeared to be contradictory, since the only difference between the two codes is that, within PolyBench, all kernels are called from functions that pass loop-range parameters and pointers to the arrays’ memory locations, while matmul has the calculations as well as the array declarations plainly within the main function. This outsourcing of the kernels to functions appeared to disable gcc to perform an automatic parallelization of the code. This is supported by the fact that even adding the -floop-parallelize-all flag (making the compiler parallelize every loop it is able to), did not lead to any parallel execution of the gemm kernel with gcc.
Figure 8: PolyBench - speedups of gcc compiled versions to original gcc code

Figure 9: PolyBench - speedups of icc compiled versions to original icc code

Figure 10: PolyBench - overall speedup

Figure 11: PolyBench codes - speedup of icc compared to gcc
When taking a closer look onto icc’s auto-parallelizer performance, surprisingly, there is also no speedup for gemm while other PolyBench kernels could be parallelized. Again, this is interesting facing the fact that icc’s auto-parallelized version of matmul performs outstandingly well. We exemplarily investigate the source of this behavior to give hints for programmers and other researchers.

To make the two codes unexceptionably comparable to each other, we slightly modified them such that both have stack-allocated and statically sized arrays and ‘alpha’ (cf. Fig. 4) set to 1.0. The only remaining difference between the two codes is that gemm calls the matrix multiplication from a function as already indicated above.

The first step is to have a look at the optimization report output by adding the option -opt-report-phase=all. Even though both codes produce the same results and calculate them in exactly the same way, the optimizer is not able to treat them identically.

As already stated in Sect. 4.2, with -O3 -parallel the icc-internal matrix multiplication detection (and replacement) option -opt-matmul is automatically activated. For matmul, it succeeds and the kernel is replaced by a library function call. However, for gemm, it does not. Trying to find the reason for that, we moved the array declarations into gemm’s kernel function. And indeed, the condition that the array declarations and the calculations now take place in the same scope enabled icc to carry out the replacement.

Intel’s compiler detects loops that could potentially be parallelized and applies a heuristic to balance the overhead of creating multiple threads versus the amount of work to be shared among threads. Based on this estimation, it finally decides whether to parallelize a loop or not [20]. Coming back to the original gemm code that could not be replaced, we take at look at the ‘regular’ auto-parallelizer of icc. Its loop-analyses leads to the following results (reported by adding -par-report3):

```
gemm ->
|  gemm ->
|  Loopnest permutation (123) | Loopnest permutation (123)
|  --> (123) | --> (123)
Matmul Report: Loopnest at line: 30 Fusion loop partitions:
| replaced by matmul intrinsic | (loop line numbers)
| ...
```

After the loop-optimizer performed a loop permutation, the parallelizer recognizes that one loop (the original inner one) is not parallelizable while the other two ones are. Nevertheless, it does not parallelize them due to ‘insufficient computational work’. This means, that even though the compiler could parallelize these loops, its internal heuristics decided not to do so. This way, the gemm kernel is not parallelized at all while matmul is even replaced with a highly optimized library function, although both codes calculate the matrix product in exactly the same way. We then forced icc to parallelize gemm by adding the compiler option -par-threshold0. After that, the compiler reports that it successfully parallelized the outer loop. Fig. 12 shows the running times for the differently compiled code versions. So even though icc’s heuristic decided that parallelizing the gemm kernel is not reasonable, doing so results in a speedup as can be seen in Fig. 13 by the bars in the background. Putting the number of LLC misses into relation, it becomes visible that our produced code can partially compensate the advantages of the MKL by the improved cache behavior for the small sizes. However, with increasing matrix sizes which let the LLC come more and more into play, the MKL performs increasingly better. In the future we will try to extend our transformation strategies to additionally exploit the last level cache potential of modern multi-core CPUs.

To summarize, we have seen that icc’s auto-parallelizer is able to generate considerable speedups by automatic parallelization and library calls but also that developers should take care of the compiler’s optimization reports to profit from its full auto-parallelization capabilities. Under certain circumstances, he or she will have to modify the code so that it fulfills the compiler’s requirements in order to get the full potential out of it. In contrast to that, at least for the matrix multiplication example, Sect. 4.2 and 4.3 indicate that source-to-source compilation tools are able to optimize and parallelize both codes based on the detected access patterns without the need for manual adjustment of the original code.

## 5. Outlook

Figure 13 shows a comparison of our generated parallel SICA versions of the matrix multiplication example (compiled with icc) to the extensively tuned and (on our system) best performing linear algebra library Intel® MKL.

For small matrix sizes up to \( N = 3072 \), SICA almost reaches the performance of the MKL. For larger matrices, the MKL can especially profit from its containing LLC (L3 cache) optimization, as can be seen in Fig. 13 by the bars in the backgound. Putting the number of LLC misses into relation, it becomes visible that our produced code can partially compensate the advantages of the MKL by the improved cache behavior for the small sizes. However, with increasing matrix sizes which let the LLC come more and more into play, the MKL performs increasingly better. In the future we will try to extend our transformation strategies to additionally exploit the last level cache potential of modern multi-core CPUs.

We are furthermore working on similar optimization strategies for Intel® MIC.

Besides this, an integration into an end-to-end compiler system would be rewarding since the access to an intermediate representation and its data typing information would extend the applicability of our method to a more general class of source codes and make some parts of our implementation even more accurate.

---

Figure 12: Running time of gemm and matmul

Figure 13: PluTo-SICA generated code and MKL - running time and relative # of LLC misses on icc
Acknowledgments

We thank Uday Bondhugula for his effort put into the Pluto framework. This work was partially funded by the German Ministry for Education and Research (BMBF) under project grant 01S13016A within the ITEA2-Project MACH.

References


