View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by computer science publication server

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

Dustin Feld and Thomas Soddemann
Fraunhofer SCAI, Schloss Birlinghoven, 53754 Sankt Augustin, Germany
[dustin.feld|thomas.soddemann]@scai.fraunhofer.de

Michael Jünger and Sven Mallach
Institut für Informatik, Universität zu Köln, Weyertal 121, 50931 Köln, Germany
[mjuenger|mallach]@informatik.uni-koeln.de

ABSTRACT

Although Single Instruction Multiple Data (SIMD) units are available in general purpose processors already since the 1990s, state-of-the-art compilers are often still not capable to fully exploit them, i.e., they may miss to achieve the best possible performance.

We present a new hardware-aware and adaptive loop tiling approach that is based on polyhedral transformations and explicitly dedicated to improve on auto-vectorization. It is an extension to the tiling algorithm implemented within the PluTo framework [4, 5]. In its default setting, PluTo uses static tile sizes and is already capable to enable the use of SIMD units but not primarily targeted to optimize it. We experimented with different tile sizes and found a strong relationship between their choice, cache size parameters and performance. Based on this, we designed an adaptive procedure that specifically tiles vectorizable loops with dynamically calculated sizes. The blocking is automatically fitted to the amount of data read in loop iterations, the available SIMD units and the cache sizes. The adaptive parts are built upon straightforward calculations that are experimentally verified and evaluated. Our results show significant improvements in the number of instructions vectorized, cache miss rates and, finally, running times.

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)—*single-instruction-stream, multiple-data-stream processors (SIMD)*

General Terms

Algorithms, Performance, Experimentation

Keywords

SIMD, SSE, AVX, TSS, polyhedral model, polyhedron model, tiling, code generation, automatic parallelization, vectorization, loop optimization, loop transformation

1. INTRODUCTION

Single Instruction Multiple Data (*SIMD*) units offer a speedup-potential brought to a wide range of users. In order to exploit the available concurrency of modern CPUs, one must achieve shared memory parallelism by multithreading and, at the same time, vectorization by effectively applying instructions to multiple data. Both tasks impose a difficult challenge to experienced programmers as well as state-of-the-art compilers. In this paper, we focus on the effective automatic exploitation of SIMD units. We found severe limitations when we made some experiments with the vectorizer of the GNU C Compiler (gcc, version 4.6). To our surprise, even for loops that can be vectorized in a straightforward manner, SSE-instructions were only set if the range specified by the loop bounds was a multiple of the SIMD register width divided by the size of the data type.

Vectorization is difficult since it usually requires an analysis of the dependence structure of the code to be optimized. It demands for the right ordering of instructions and fast accesses to data in order to leverage its full speedup potential. Unfortunately, in many cases even the existence of a legal order of instructions cannot be easily recognized by humans or compiler procedures. Here, polyhedral code optimization is a powerful tool to detect and exploit parallelism in loop structures. It provides a formal characterization of affine loop nests, their iteration spaces, the dependencies between statements and the iteration points in which they occur [2, 7, 9, 12, 13]. It can therefore be used to generate valid transformations of a given source code. Further, *tiling* (or *blocking*) of nested loops is a well-known technique to improve data locality and, if concurrency concerning outer loop dimensions is possible, to perform automatic parallelization [21]. In this manner, transformations such as loop fusion, splitting, skewing, or interchange may enable a coarse-grain (tile-wise) or a fine-grain (loop-internal) concurrency (or both) even where this is not the case for the original source code [2].

There is extensive literature dealing with the optimization of tilings. However, to the best of our knowledge, there is yet no implemented approach that integrates loop transformations which broadly enable automatic vectorization with tilings and an adaptive hardware-aware tile size selection (TSS). Trifunovic et al. [20] analyze the impact of loop transformations on the resulting possibilities to apply auto-vectorization and performance by means of a cost model that can be seamlessly integrated into the polyhedral model. Based on this, they propose a framework to choose the best-suited loop for vectorization within a nest. Unfortunately, it does not comprise a TSS model. As opposed to that, there exist several TSS models which are, however, not explicitly geared towards an improved vectorization. Coleman and McKinley [8] present an iterative technique to calculate cache-fitting tile sizes for all loop dimensions. This is interesting in conjunction with the approach presented

in this paper, especially for cases where there is no vectorizable loop available. This is also true for Sarkar's and Megiddo's [17] approach to generate tile sizes via a memory-oriented cost model of the given loop nest and an analytical model to finally perform the TSS. However, it is restricted to loop nests of depth two or three. Shirako et al. [18] present a method to analytically bound the search space for tile sizes that lead to a good performance. They potentially leave loop dimensions unblocked and propose to tile a vectorizable loop into large blocks. While these general ideas are similar to ours, their approach is merely capable to perform a one-level tiling and based on an empirical search method while ours uses a straightforward polyhedral and analytical basis. Ghosh et al. [11] propose to use cache miss equations [3] for TSS and in order to detect poor cache performance. They show how loop transformations (including tiling) can help to improve on this. Abella et al. [1] use the equations to determine optimal tile sizes by means of a genetic algorithm. However its running time on some inputs is not applicable for a common compilation process.

In this paper, we present a new hardware-oriented TSS approach explicitly targeted to vectorization. It is an adaptive procedure that specifically tiles vectorizable loops with dynamically calculated sizes. We implemented our approach as an extension to PluTo [4, 5] which is an academic source-to-source compiler framework that performs tilings based on polyhedral optimization. The resulting code can be compiled by any *C* compiler. In particular, we use PluTo to obtain valid transformations and tilings of loops as well as to gather information about those loops that can actually be vectorized. However, in contrast to PluTo, we orient the tiling towards an effective use of SIMD units. Instead of partitioning the iteration space with respect to all loop dimensions and into tiles of static size, we restrict the tiling to those loops that are relevant for the data to be processed in a vectorized manner. Further, we dynamically adapt the size of tiles to the SIMD register width and the cache sizes of the underlying hardware. Ideally, our approach leads to a software pipeline of blocks to be processed by the SIMD units. We show for two example source codes that we obtain improved running times by a combination of a measurably well-performing stream of data through the cache hierarchy and an increased rate of issued SIMD instructions.

2. POLYHEDRAL TRANSFORMATIONS, VECTORIZABLE LOOPS AND TILINGS

A loop qualifies for vectorization if it is innermost with respect to its nest and parallelizable, i.e., there are no data dependencies between its iterations.

In the polyhedral model, one considers the iteration points of a loop nest and the dependencies between them using $\mathbb{Z}$-polyhedra [9, 12, 13], as depicted in Fig. 1. In this representation, the index variable of each loop relates to one dimension of the associated polyhedron. A valid transformation corresponds to a change in the order of execution of the iteration points that preserves compliance with the dependencies. This may include the manipulation of loop dimensions (index variables) leading to a deformation of the polyhedron such that computations can be processed in parallel with respect to one or more of the dimensions. By applying integer programming techniques, PluTo is capable to perform transformations such that the necessary communication across the dimensions of the resulting loops is minimized [6]. This is beneficial for parallelization. If communication is not necessary with respect to an outer loop dimension, then a parallelization of the inner loops' iterations is possible. If communication is not necessary with respect to the innermost loop dimension, then the iterations of this loop can be processed in parallel, e.g., by vectorization. Fortunately, PluTo is able to mark loops that qualify for vectorization, possibly by interchanging it to become the innermost one (which we assume from now on to be the case). Furthermore, PluTo can compute a partitioning of the iteration space into tiles. Consider Fig. 1 for an example where a legal (rectangular) tiling is possible only after a transformation of the loop nest. The left tiling is illegal, because iteration points between blocks have reciprocal dependencies. After manipulating the loop dimensions, it is possible to apply the tiling depicted in the right image since it now allows for an order of execution that respects all dependencies. We use the polyhedral representations within PluTo to obtain such legal loop tilings.

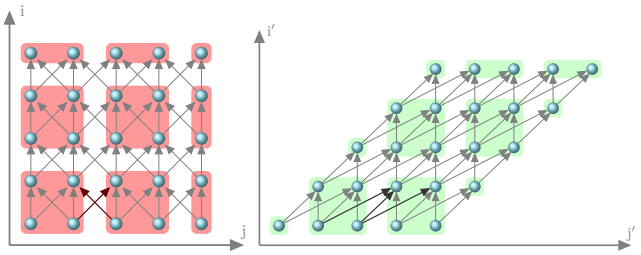

Figure 1: A polyhedral representation of a nested loop with its dependence structure and an invalid tiling (left) as well as a valid one (right).

3. CENTRAL IDEAS

In order to fully exploit the speedup potential of SIMD units, it is advantageous to have access to data that is accordingly prefetched into the available cache hierarchy. How can we achieve this by a smart tiling?

As already stated, a loop to be vectorized must be (made) innermost. Its index variable imposes a constant offset/stride between the memory addresses of data that is to be successively packed into (SIMD) registers while the indices of all other loops remain constant. It can thus be expected 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. Then, the prefetcher should be able to better pre-load future operands while current calculations are served from the cache hierarchy. The prefetching causes cache misses that do not significantly influence the running time but impose cache hits when the operands are really needed. On the other hand, a large number of iterations in the innermost loop may also harm spatial locality in comparison to fewer ones. This is particularly true if the index variable of the innermost loop does not correspond to the minor dimension of all accessed arrays (and therefore not to one-strides in memory).

Clearly, there must be a trade-off between the advantages and disadvantages of a large blocking of the innermost loop. However, we believe that the blocking of (a) *all* loops into

(b) *constantly* sized blocks (like, e.g., the default tiling into blocks of 32 iterations performed by PluTo) is unlikely to perform well for every application and every system. This is especially true for deeply nested loops with many statements within the innermost loop. As an example, if d is the depth of a loop nest, then the innermost statements of a tile using PluTo's default tiling are executed 32^d times and the data accessed by these statements is unlikely to fit into a cache with increasing d. Nonetheless, this fits quite well for 'typical' loop nests with depth two or three and today's usual cache sizes. PluTo allows to also enforce a second-level tiling of the resulting loops into blocks of 8 which already addresses the cache memory hierarchies of modern processors. Alternatively, the user may specify tile sizes manually. We used this fact to elaborate on our ideas and made experiments with different sizes for a specific tiling of one or two loops only.

3.1 Experiments with manual tile sizes

We consider two test cases both of which are written in `C` and have been taken from PluTo's example suite:

- A standard *matrix multiplication* (see Fig. 18)
- A *correlation matrix algorithm* (see Fig. 21)

The environment for the tests and the benchmarks in the subsequent section comprises the following Intel Xeon processor and is running Scientific Linux 6.

CPU:	Intel® Xeon® CPU X5650 (2.67 GHz)
L1 / L2 / L3 cache:	32 KB (data) / 256 KB / 12288 KB
SSE version:	4.2 (128 bit registers)

All runs were performed single-threaded using gcc 4.6 or icc 13.0, both with optimization level `-O3` on single precision floating point data.

3.1.1 Manual one-level tiling

To start, we consider a pure tiling of the vectorized loop only with manually set tile sizes $q^{L1} = 4$, 64 and 256. We compare it to the original source code and PluTo, configured to also generate a one-level tiling of all loops using its default sizes (see Fig. 23).

First, we evaluate the resulting matrix multiplication codes for various choices of (symmetric) matrix sizes $N = M = K$ within a small range for a fine-grain analysis.

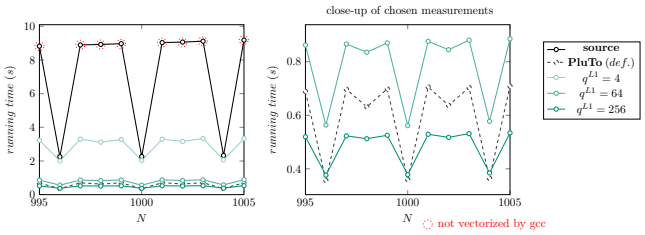

Figure 2: Running times for the one-level-tiled matrix multiplication (fine grain) with gcc

As is depicted in Fig. 2, the performance of the original (source) version highly depends on the size of the matrix. For sizes that are multiples of four, the code generated by gcc performs about four times faster. Application of gcc's verbose mode confirmed the hypothesis that it is only able to apply auto-vectorization in these cases. Likewise, already a very fine-grain tiling with $q^{L1} = 4$ suffices in order to enable auto-vectorization by gcc for any matrix size. Increasing q^{L1} to 256 leads to running times that are comparable to those achieved by the default tiling of PluTo or even faster. The deviant behavior for sizes that are not multiples of four can be explained with the 'remainders' that result from the tiling in these cases. We experimentally determined a perfect correlation between the size of the last tiles and the rate of vectorization, i.e., again gcc appears to not vectorize these remaining loop iterations, especially if their number is odd. We further elaborate on different parameters that influence the sustained performance in Sect. 4.2.

In Table 1, the relative performance from the fine-grained setting is confirmed for a larger interval of register-fitting (multiples of four) and non-register-fitting matrix sizes.

N	256	512	1024	1536	2048
source	0.0096	0.0905	2.3191	7.9956	20.2370
PluTo (def.)	0.0063	0.0536	0.4645	1.4801	3.7902
$q^{L1} = 4$	0.0202	0.1829	2.1804	8.2643	26.1112
$q^{L1} = 64$	0.0071	0.0672	0.6092	2.4307	10.0529
$q^{L1} = 256$	0.0048	0.0454	0.3887	1.6499	5.2734

N	257	513	1025	1537	2049
source	0.0223	0.1956	9.9101	34.0321	82.7667
PluTo (def.)	0.0121	0.1076	0.8839	2.9326	7.5070
$q^{L1} = 4$	0.0444	0.3702	3.6207	14.8025	38.7883
$q^{L1} = 64$	0.0113	0.1081	0.9672	3.7944	14.0045
$q^{L1} = 256$	0.0079	0.0673	0.5736	2.2688	6.4815

Table 1: Running times for the one-level-tiled matrix multiplication (coarse grain) with gcc

We apply the same test cases for the correlation matrix algorithm which has a more complicated code structure. As can be seen in Fig. 3 and Table 2, gcc is not able to vectorize its original version at all. However, again tiling the loops corresponding to the M-matrix-dimension (together with the corresponding code transformation) enables gcc to auto-vectorize the code already when setting q^{L1} to 4. With $q^{L1} = 256$, the running times are almost always faster than with PluTo's default one-level tiling.

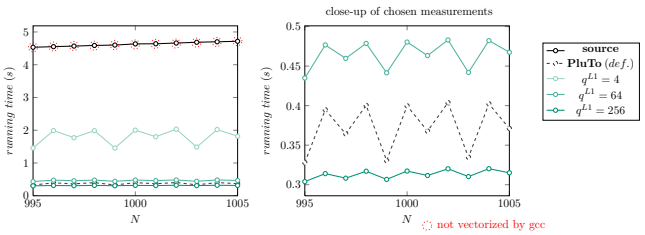

Figure 3: Running times for the one-level-tiled correlation matrix algorithm (fine grain) with gcc

3.1.2 Manual two-level tiling

Now, we consider a two-level tiling (of the vectorized loop and additionally the outermost loop of its nest) again using manually set tile sizes $q^{L1} = 4$, 64 and 256 for the vectorized loop and $q^{L2} = 2$, 4 and 8 for the outermost one. We

N	256	512	1024	1536	2048
source	0.0114	0.2140	4.8512	22.4826	60.9249
PluTo (def.)	0.0078	0.0599	0.4844	1.5089	3.8053
$q^{L1} = 4$	0.0221	0.1844	2.0942	8.5166	21.8286
$q^{L1} = 64$	0.0065	0.0573	0.5204	1.8351	5.7119
$q^{L1} = 256$	0.0054	0.0404	0.3472	1.2619	3.7619

N	257	513	1025	1537	2049
source	0.0115	0.2152	4.9040	22.4878	61.0156
PluTo (def.)	0.0070	0.0533	0.4367	1.3892	3.3829
$q^{L1} = 4$	0.0209	0.1745	1.8981	7.5941	20.9514
$q^{L1} = 64$	0.0065	0.0557	0.5031	1.7586	5.5304
$q^{L1} = 256$	0.0054	0.0403	0.3430	1.2401	3.7118

Table 2: Running times for the one-level-tiled correlation matrix algorithm (coarse grain) with gcc

configured PluTo to also generate a two-level tiling using its default sizes (see Fig. 23). The running times are depicted in Figures 4 and 5 and, for ease of comparison, the one-level tiling running times are shown dotted in the right graphs. As might have been expected, the additional blocking leads to shorter running times in all cases.

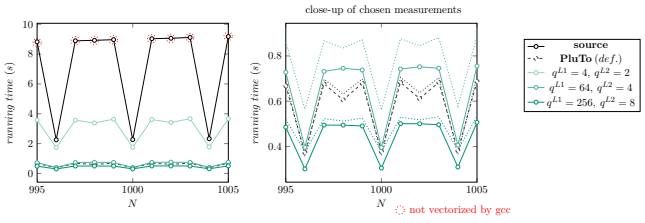

Figure 4: Running times for the two-level tiled matrix multiplication (fine grain) with gcc

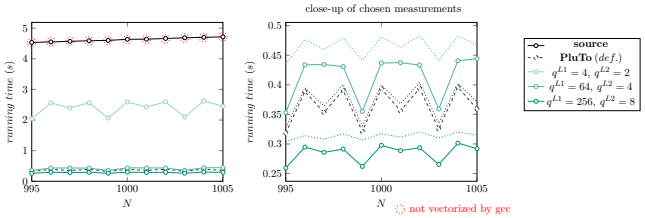

Figure 5: Running times for the two-level tiled correlation matrix algorithm (fine grain) with gcc

3.2 Ideas towards an automatic derivation of tile sizes

The results presented so far appear to support the hypothesis that a tiling of specific loops can be as effective as a tiling of all loops. Furthermore, they tell us that loop ranges should be *SIMD-specific*, i.e., fit to SIMD register widths, in order to leverage the full potential of gcc's automatic vectorizer. Until now, the running times seem to improve with increasing tile sizes. Clearly, the amount of increase that pays off must be limited. Our intuition was that a natural limitation should stem from the cache sizes. Ideally, the block size for the first level tiling should be fitted to the ratio of the size of the L1 cache and the amount of data read in one iteration of the loop to be vectorized. Similarly, the block size for the second level tiling should be fitted to the ratio between the L2 cache size and the L1 cache size. In the best case, this strategy could lead to the situation that all required data for one tile of the vectorized loop fits into the L1 cache and that such blocks of data can be successively pipelined from the L2 cache. With the larger innermost tile size, we should be able to move compulsory cache misses to the prefetcher and, with the *cache-specific* tiling, we should optimize for less capacity cache misses at the same time. Hence, we call the just described combination of both concepts a SIMD- and cache-specific (*SICA*) tiling. For the (optional) second-level tiling, we select the outermost loop of the corresponding nest. This strategy keeps changes to the inner loops as rare as possible which, as a heuristic, should be good for the prefetcher.

We set up a straightforward model to compute all the information needed for experiments to analyze whether this is indeed a promising approach. First of all, we need to know how many new operands must be loaded by each of a loops' iterations. This requires an analysis of the loops' statements, since, e.g., operands (or addresses) that are accessed multiple times should be loaded only once per iteration. Furthermore, constant values as well as operands that do not depend on the vectorized loop should be loaded even only once at all for the entire loop. As already stated in Sect. 2, the innermost loop may, in general, contain several statements which are considered to compose a statement-block. We calculate the total amount of data elements $\mathcal{E}$ to load per iteration for each of these blocks.

$$\text{Elements per Iteration} : \text{ElPeIt} = \mathcal{E} \tag{1}$$

Further, we need the following hardware parameters:

- $\mathcal{C}_{L1}$: The size of the L1 cache (in KBytes)
- $\mathcal{C}_{L2}$: The size of the L2 cache (in KBytes)
- R: The SIMD register width (in Bits)

Using this information, the number of elements of the given type (e.g. `float` or `double`) with size $\mathcal{D}$ (in Bytes) that fit into the L1 cache can be calculated as follows:

$$\text{cache size in elements} : \text{CaSiEl} = \frac{\mathcal{C}_{L1} * 1024}{\mathcal{D}} \tag{2}$$

The number of operands of the given data type with size $\mathcal{D}$ (in Bytes) that can be packed into the SIMD registers is denoted by:

$$\text{elements per register} : \text{ElPeRe} = \frac{\mathcal{R}}{8 * \mathcal{D}} \tag{3}$$

Since there might be other variables that should be cached, one might like to adjust the ratio of the L1 cache size to use to, e.g., only 90% or 80%. We therefore introduce an according parameter ρ with the meaning that $\rho = 1.0$ relates to 100%.

$$\text{ratio of cache to use} : \rho \tag{4}$$

For each block of statements we may now compute how many iterations shall be blocked so that all operands required by this block ideally fit into the L1 cache at once:

$$\text{iterations to block} : \text{ItToBl} = \rho * \frac{\text{CaSiEl}}{\text{ElPeIt}} \tag{5}$$

Finally, we want to make the first-level tile size q^{L1} a multiple of the SIMD-register width. Hence, we compute the greatest multiple of ElPeRe that fits into the L1 cache as follows:

$$q^{L1} = \left\lfloor \frac{\text{ItToBl}}{\text{ElPeRe}} \right\rfloor * \text{ElPeRe} \tag{6}$$

Summing up all calculations into a single formula yields:

$$q^{L1} = \left\lfloor \frac{\rho * \frac{\mathcal{C}_{L1}*1024}{\mathcal{D}} * \frac{1}{\mathcal{E}}}{\frac{\mathcal{R}}{8*\mathcal{D}}} \right\rfloor * \frac{\mathcal{R}}{8*\mathcal{D}} \tag{7}$$

$$= \left\lfloor \rho * \frac{\mathcal{C}_{L1} * 8192}{\mathcal{R} * \mathcal{E}} \right\rfloor * \frac{\mathcal{R}}{8*\mathcal{D}} \tag{8}$$

Now for the second level tiling, we simply calculate the ratio of the two cache sizes.

$$q^{L2} = \frac{\mathcal{C}_{L2}}{\mathcal{C}_{L1}} \tag{9}$$

3.2.1 Example

Consider the standard matrix multiplication for single precision floating point data with only one statement $C[i][\mathbf{j}] = C[i][\mathbf{j}] + \alpha * A[i][k] * B[k][\mathbf{j}]$ and vectorized **j**-loop. Two new data elements need to be loaded per **j**-iteration, namely $C[i][\mathbf{j}]$ and $B[k][\mathbf{j}]$. This leads to the following SICA L1 tile size for our test system with 32 KByte of L1 cache and 128 Bit SSE registers:

$$q^{L1} = \left\lfloor \rho * \frac{32 * 8192}{128 * 2} \right\rfloor * \frac{128}{8 * 4} \overset{\rho=1.0}{=} 4096 \tag{10}$$

Since our system has 256 KByte of L2 cache, the second block size evaluates to $q^{L2} = \frac{256}{32} = 8$.

3.3 SICA extensions to PluTo

We implemented our SICA tiling as an extension of PluTo together with several new parameters and functionalities that cause only neglectable overhead. It comprises adaptive components, like, e.g., procedures to determine hardware parameters by using the `CPUID` [15] instructions (they can be equally manually set via a configuration file) and new routines to calculate the amount of data loaded in one loop iteration. If an innermost loop contains multiple statements (possibly as a consequence of the polyhedral transformations), it is viable to group them into blocks for which the tiling is then performed independently. Unlike in PluTo's original tiling algorithm, every block of statements can be associated with individual tile sizes. This is necessary since different statement blocks (within the same loop nest) may require a different number of operands to be loaded per iteration. Nevertheless, one may still request a globally uniform tiling by adopting the minimal or maximal determined tile size for all statement blocks. As an example, a rectangular SICA tiling of a perfectly nested loop with only one statement S is depicted in Fig. 24.

4. ANALYSIS AND BENCHMARKS

The following experiments can be divided into two parts. First, we deliver a verification of the proposed correlation between the amount of data read within the vectorized loops' iterations, tile sizes, cache sizes and performance. After that, we evaluate the performance of the corresponding adaptive approach concerning running times, cache miss rates, TLB misses and the rate of issued SIMD instructions.

4.1 Verification of the approach

In order to evaluate the impact of the tile sizes only, we fix some (asymmetric) matrix sizes. We keep the matrix dimensions corresponding to non-vectorized loop dimensions small in order to be able to benchmark a large interval of sizes for the vectorized one and to obtain reasonable running times at the same time.

4.1.1 SICA L1 tiling

We again start our experiments with a one-level tiling. For the matrix multiplication, we set $M = 189, N = 139233$ and $K = 189$, since the loop corresponding to the N-dimension is the vectorized one. Then, we vary the tile sizes by successively changing the cache-ratio parameter ρ from 0 to 10 in steps of 0.01 units. This results in 1000 different versions of the code with tile sizes up to 36864. Fig. 6 shows their corresponding running times which are all within the shadowed area while the line is a cubic interpolation of them with some smoothing applied.

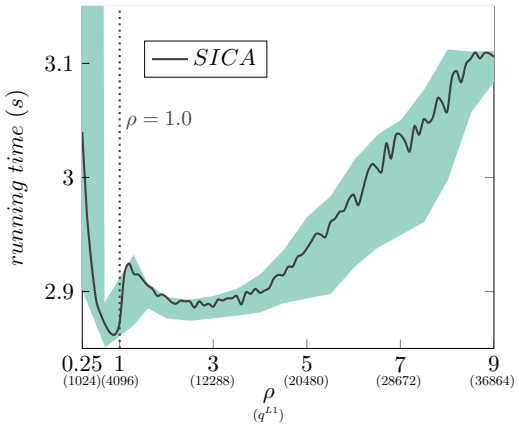

Figure 6: Impact of the tile size on the running time of the matrix multiplication code with gcc

There is a global optimum that corresponds to about 90% of the L1 cache size. This appears to confirm a correlation between performance, tile and cache sizes. Furthermore, we claimed a relationship between the optimum tile sizes and the amount of data that needs to be loaded within the vectorized loops' iterations. To verify this, we additionally measured the running times of codes performing the addition of two or even three matrix multiplications (the corresponding statements in the nested loop are depicted in Fig. 20). For each additional matrix multiplication, there is one additional operand to be loaded per iteration. Regarding our test system, this leads to tile sizes of $q^{L1} = 4096$ for a single matrix multiplication [*matmul1*], $q^{L1} = 2728$ for the sum of two of them [*matmul2*] and $q^{L1} = 2048$ for the sum of three of them [*matmul3*]. Again, Fig. 7, in which we scaled the running times of the different versions to a common ordinate (each individual range in seconds is denoted in brackets), shows that the best tile size is at about 80 to 90% of the theoretical optimum.

As before, the same experiments are repeated for the correlation matrix algorithm. Here, we set $M = 11923$ and $N = 89$ since loops corresponding to the M-dimension are vectorized. Since there are several statements and multiple loop nests (see Fig. 21 for the original code and Fig. 22

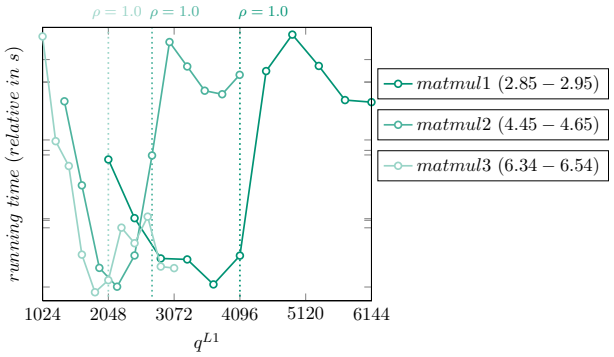

Figure 7: The effect of more data to be loaded in each iteration of the vectorized loop with gcc

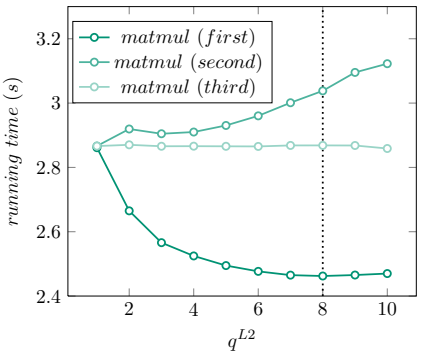

Figure 9: Second level tiling: The impact of the choice of a distinct loop and its tile size (with gcc)

for the SICA version), there is no unique tile size but an individual one for each statement block. This is why it is reasonable to measure the running times only by varying ρ and thereby scaling the tile sizes proportionally.

In Fig. 8, the best tile size turns out to be quite exactly the theoretical optimum. Another interesting observation are the local optima for $\rho = 2.0$ and $\rho = 3.0$, where the necessary data could be loaded from the cache in exactly two or three portions.

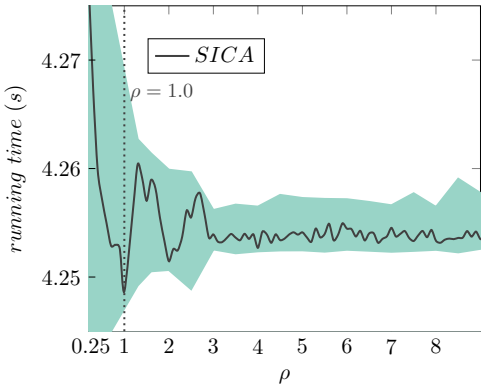

Figure 8: Impact of the tile size on the running time of the correlation matrix algorithm with gcc

4.1.2 SICA L2 tiling

For a two-level tiling, we already theoretically justified to choose the outermost loop from the nest of the vectorizable one. Now, we deliver an experimental justification of this decision using the matrix multiplication example. The resulting running times for each selection of one of the three nested loops for the second-level tiling (while tiling the first level with the calculated q^{L1}) are depicted in Fig. 9. Only when the outermost loop (first) is selected, the running time is improved. Furthermore, the calculated theoretical optimum of $q^{L2} = 8$ (the L2 cache on the test-system is 8 times larger than the L1 cache) leads to the best running times.

4.2 Performance counters

To further investigate the impact of the SICA tiling and to explain the improved running times, we measured PAPI [19] performance counters for the original source code and the two-level tilings by PluTo (default) and our extension with the asymmetric matrix sizes from Sect. 4.1. In particular, we focused on the effects concerning the L2 cache miss rate (as each L2 cache miss is preceded by an L1 cache miss and the L2 cache, being the last on-core level, is crucial for our intended pipeline), the rate of introduced SIMD instructions and the number of L1/L2-TLB misses. Concerning the cache behavior, it is important to consider miss rates instead of absolute numbers of cache misses. This is true since a successful prefetching of the accessed data may lead to an increase of the total number of cache misses and accesses at the same time. However, misses obtained like this do not significantly harm performance but impose cache hits when the operands are really needed. We additionally measured the cache hit rate but did not explicitly picture it. As could be expected, it sums up to 100% with the cache miss rate (besides minor deviation of the counters).

When using gcc, both the SICA and PluTo's default tiling largely improve the performance of the matrix multiplication code (cf. Fig. 10). While the original source code cannot be vectorized by gcc at all, the tiled versions enable the introduction of SIMD instructions. In case of the SICA tiling, in fact nearly all instructions are SIMD instructions. Tiling only two instead of all loop dimensions results in fewer cases where the control flows enters the innermost loop with 'remainders' of iterations that cannot be vectorized by gcc. This effect is in fact intensified by the choice of asymmetric matrix sizes for these experiments. The larger tile size for the vectorized loop (in comparison to PluTo) corresponds to many predictable accesses to the innermost (linearly stored) array dimension. Both tiled versions lead to a reduction of the L2 cache miss rate. This is especially true with the fitted tile sizes calculated by the SICA extension which also leads to a stronger reduction of TLB misses.

Fig. 11 shows that the internal optimizations done within icc 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 even marginally better for the original source than with the fitted SICA tile sizes. However, with the SICA tiling, icc is able to produce the fastest code since it can again turn nearly every instruction into a SIMD instruction. The situation concerning TLB misses is as before with the exception that the code produced by icc on the original source code leads to far less TLB misses than with gcc.

In case of the correlation matrix algorithm and gcc, the

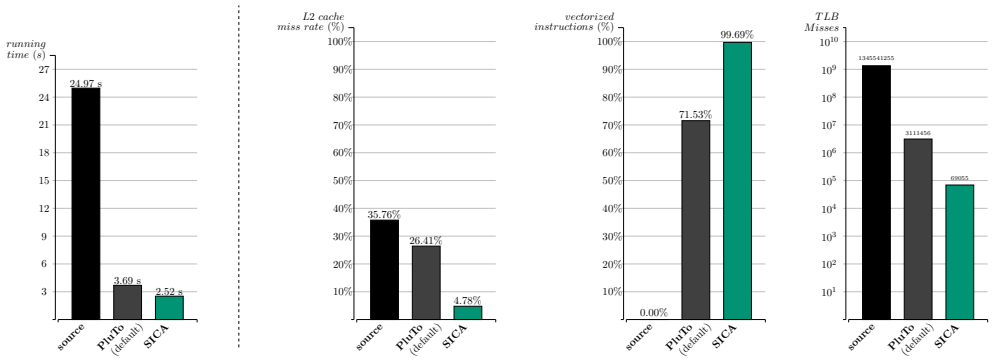

Figure 10: PAPI performance counters for the matrix multiplication code with gcc

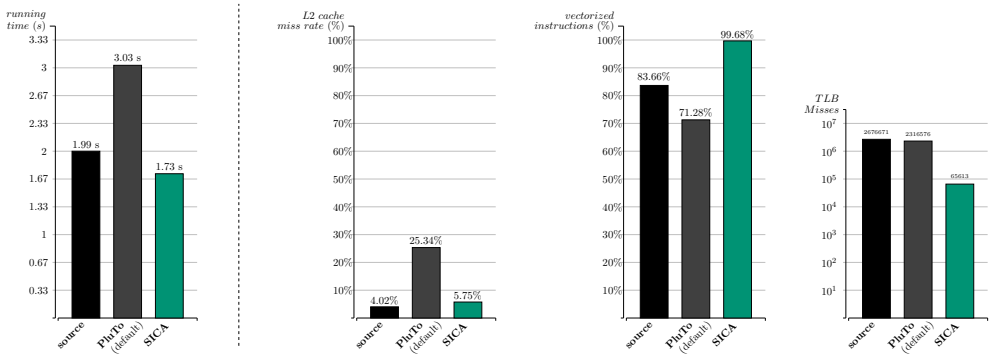

Figure 11: PAPI performance counters for the matrix multiplication code with icc

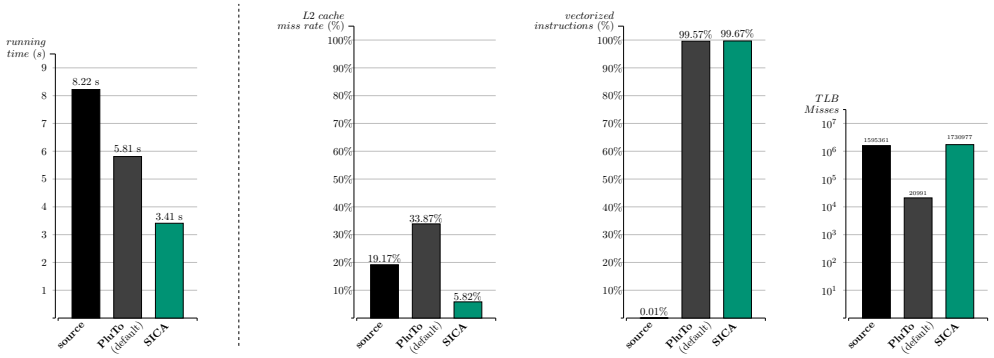

Figure 12: PAPI performance counters for the correlation matrix code with gcc

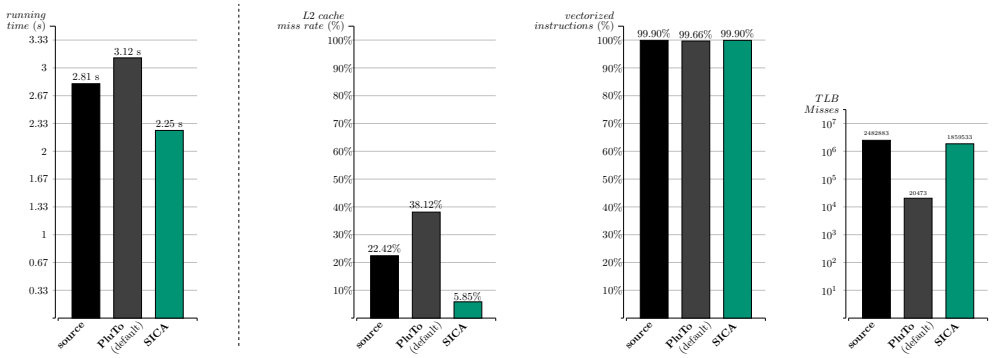

Figure 13: PAPI performance counters for the correlation matrix code with icc

source codes produced by PluTo and with the SICA tiling both improve the running time as is shown in Fig. 12. Even more, both tilings lead to a nearly perfect vectorization of the code. However, whereas the PluTo code 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. As opposed to that and with less impact on the running time, PluTo performs by far best concerning TLB misses while the SICA version cannot even improve on the number of TLB misses that are produced with the original source code.

As with 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. For the L2 cache miss rate and the TLB misses, the results are similar to the case of using gcc. 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.

4.2.1 *Interim summary*

For the considered test cases and compared to the original source code and PluTo's default tiling, the SICA approach always produced the best running times, no matter if used as input for gcc or icc. A big advantage of the SICA tiling is that it typically enables the compilers to vectorize a larger number of instructions. Further, across all benchmarks, it leads to a small L2 cache miss rate which is the best one obtained except for the matrix multiplication with icc. Concerning TLB misses, possible improvements depend on the access pattern of the statements within the vectorized loop.

4.3 Performance benchmarks

Finally, we would like to verify the performance of the SICA two-level tiling for the two application codes across a larger range of symmetric matrix input sizes. We selected the sizes $i \cdot 1024$ for $i \in [2, 8]$. Since these are all multiples of four, they diminish the disadvantages of PluTo's default tiling in the benchmarks before, i.e., there will be no non-vectorizable 'remainders' of loop iterations anymore. Similarly, gcc will always be able to apply its auto-vectorization already to the original source code.

Fig. 14 and 16 show the corresponding running times for gcc and Fig. 15 and 17 those for icc. The output produced by the SICA tiling appears to support both compilers to produce faster code compared to the original source and PluTo's default tiling. For completeness, Table 3 depicts the average speedups obtained for the tested input sizes.

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 3: Average speedups (coarse grain)

5. CONCLUSION, ONGOING WORK AND OUTLOOK

We presented an adaptive hardware-aware tiling approach that has been implemented and evaluated as an extension to

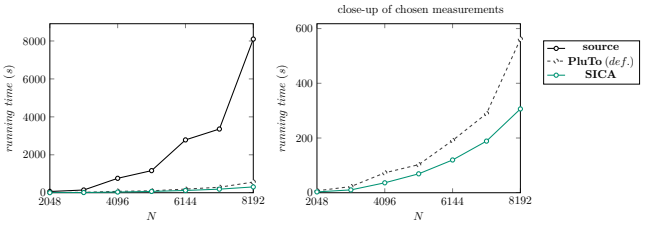

Figure 14: Running times for the two-level tiled matrix multiplication (coarse grain) with gcc

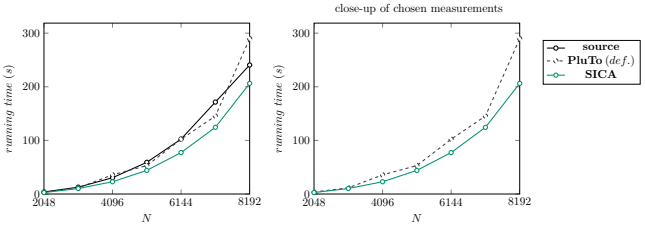

Figure 15: Running times for the two-level tiled matrix multiplication (coarse grain) with icc

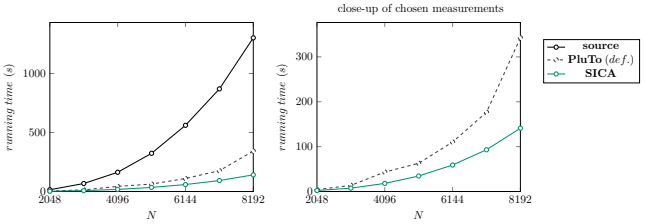

Figure 16: Running times for the two-level tiled correlation matrix (coarse grain) with gcc

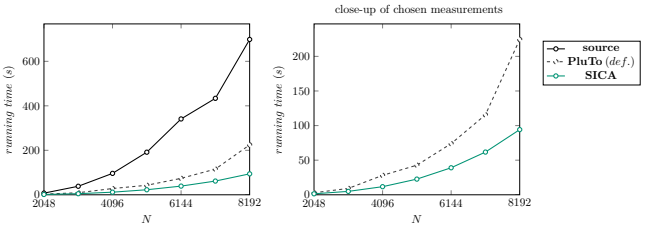

Figure 17: Running times for the two-level tiled correlation matrix (coarse grain) with icc

PluTo. In contrast to PluTo, it does not perform a tiling of all loops in a nest, but specific tilings of distinct loops that are beneficial for an effective vectorization. It is adaptive in that it performs an automatic analysis of the amount of data accessed by a loop's statements and is able to derive information about the underlying hardware such as cache sizes and SIMD register widths. These parameters are used to derive dynamic tile sizes that ideally lead to a software pipeline of blocks to be processed by the SIMD units and to be well prefetched through the cache hierarchy.

We verified and evaluated our approach experimentally on two source codes from PluTo's example suite, namely a standard matrix multiplication and a correlation matrix algorithm. As our results show, the proposed tiling strategy

leads to better running times in comparison to PluTo's default tiling and the original source code when its output is compiled with gcc 4.6 or icc 13.0. An analysis with performance counters brought to light that these results can be mainly explained by an increase of issued SIMD instructions and a strong reduction of the L2 cache miss rate.

In further studies (see [10]), we examined the behavior of our extension on further source codes. They include applications that contain very deep loop nests and where a static all-dimension tiling therefore leads to blocks that are by far too large for today's usual cache sizes. This may considerably harm performance, whereas our extension can handle these cases by its dynamic analysis and the restriction to tile at most two loops.

However, we do not consider a tiling of at most two loops as a globally superior strategy. To the contrary, if the dependence structure of a statement block refers to multiple loop dimensions, a corresponding specific multi-dimensional tiling could be superior in terms of spatial locality and TLB performance. We plan to consider this in future work. Similarly, we would like to manipulate the loop transformations towards an automatic optimization for one-strided memory accesses. This could be potentially achieved by (a) separating the statements of a block according to their access patterns, (b) transforming them one by one targeting one-strided accesses and (c) vectorizing the resulting loop nests. We also plan to experiment with a L3 cache tiling as well as with the combination of our developments with automatic shared-memory parallelization in order to exploit the full concurrency potential of modern multicore processors.

Currently, our developments are ported to the PoCC [16] framework (that includes PluTo) in order to integrate our extensions into Polly and thereby into the LLVM infrastructure [14].

6. ACKNOWLEDGMENTS

Thanks to Uday Bondhugula for the development of the PluTo framework.

7. REFERENCES

[1] J. Abella, A. Gonzalez, J. Llosa, and X. Vera. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *Proc. Int. Parallel Processing Workshop*, pages 568 – 577, 2002.

[2] U. K. Banerjee. *Loop Parallelization*. Loop transformations for restructuring compilers. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[3] D. Bernstein, D. Cohen, and A. Freund. Compiler techniques for data prefetching on the PowerPC. In *Proc. IFIP WG10.3 working Conf. on Parallel Architectures and Compilation Techniques*, PACT '95, pages 19–26, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[4] U. Bondhugula. PluTo: An automatic parallelizer and locality optimizer for multicores.

[5] U. Bondhugula. *Effective Automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Columbus, OH, USA, 2008.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. 2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[7] A. Cohen, M. Sigler, D. Parello, S. Girbal, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM Int. Conf. on Supercomputing (ICS'05)*, pages 151–160, 2005.

[8] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program. volume 21*, pages 313–348, 1992.

[10] D. Feld. Effiziente Vektorisierung durch semi-automatisierte Code-Optimierung im Polyedermodell, *available at http://publica.fraunhofer.de/documents/N-194546.html*, 2011.

[11] S. Ghosh, M. Martonosi, and S. Malik. Automated cache optimizations using CME driven diagnosis. In *Proc. 14th Int. Conf. on Supercomputing*, ICS '00, pages 316–326, New York, NY, USA, 2000. ACM.

[12] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program. volume 34*, pages 261–317, 2006.

[13] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004.

[14] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly - polyhedral optimization in LLVM. In *First Int. Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[15] Intel. Intel processor identification and the CPUID instruction. Technical report, Intel Corporation, 2011.

[16] L.-N. Pouchet. PoCC: the Polyhedral Compiler Collection.

[17] V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *Proc. IEEE Int. Symp. on Performance Analysis of Systems and Software*, ISPASS '00, pages 146–153, Washington, DC, USA, 2000. IEEE Computer Society.

[18] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. Analytical bounds for optimal tile size selection. In *Proc. 21st Int. Conf. on Compiler Construction*, CC'12, pages 101–121, Berlin, Heidelberg, 2012. Springer.

[19] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with PAPI-C. 2009.

[20] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proc. 18th Int. Conf. on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society.

[21] J. Xue. *Loop tiling for parallelism*. Kluwer Int. Series in Engineering and Computer Science. Kluwer Academic, 2000.

APPENDIX

A. STATIC CONTROL PARTS (SCoPs) OF THE CONSIDERED EXAMPLE CODES

```
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];
```

Figure 18: Original standard matrix multiplication

```
for(t1=0;t1<=floor(M-1,8);t1++) {
 for(t2=0;t2<=floor(N-1,3684);t2++) {
  for(t3=0;t3<=K-1;t3++) {
   for(t4=8*t1;t4<=min(M-1,8*t1+7);t4++) {
    {
     lbv=3684*t2; ubv=min(N-1,3684*t2+3683);
     #pragma ivdep
     #pragma vector always
     for(t9=lbv; t9<=ubv; t9++) {
      C[t4][t9]=C[t4][t9]+alpha*A[t4][t3]*B[t3][t9];;
     }
    }
}}}}
```

Figure 19: Matrix multiplication (SICA applied with $\rho = 0.9$)

```
C[i][j]=C[i][j]+A[i][k]*B[k][j];
C[i][j]=C[i][j]+A[i][k]*B[k][j]+D[i][k]*E[k][j];
C[i][j]=C[i][j]+A[i][k]*B[k][j]+D[i][k]*E[k][j]
        +F[i][k]*G[k][j];
```

Figure 20: The different statements in matmul1, matmul2 and matmul3

```
/* 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];
 }
}
```

Figure 21: Original correlation matrix SCoP

```
for(t2=0;t2<=floor(M-1,8);t2++) {
 for(t3=ceil(t2-920,921);t3<=floor(M,7368);t3++) {
  for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),
                             7368*t3+7366);t5++) {
   {
    lbv=max(7368*t3,t5+1); ubv=min(M,7368*t3+7367);
    #pragma ivdep
    #pragma vector always
    for(t10=lbv; t10<=ubv; t10++) {
     symmat[t5][t10]=0.0;;
    }
   }
}}}
for(t2=1;t2<=M-1;t2++) {
 symmat[t2][t2]=1.0;;
}
for(t2=0;t2<=floor(N,8);t2++) {
 for(t3=0;t3<=floor(M,2456);t3++) {
  for(t5=max(1,8*t2);t5<=min(N,8*t2+7);t5++) {
   {
    lbv=max(1,2456*t3); ubv=min(M,2456*t3+2455);
    #pragma ivdep
    #pragma vector always
    for(t10=lbv; t10<=ubv; t10++) {
     data[t5][t10]-=mean[t10];;
     data[t5][t10]/=sqrt(N)*stddev[t10];;
    }
   }
}}}
for(t2=0;t2<=floor(M-1,8);t2++) {
 for(t3=ceil(2*t2-920,921);t3<=floor(M,3684);t3++) {
  for(t4=1;t4<=N;t4++) {
   for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),
                              3684*t3+3682);t5++) {
    {
     lbv=max(3684*t3,t5+1); ubv=min(M,3684*t3+3683);
     #pragma ivdep
     #pragma vector always
     for(t10=lbv; t10<=ubv; t10++) {
      symmat[t5][t10]+=(data[t4][t5]*data[t4][t10]);;
     }
    }
}}}}
for(t2=0;t2<=floor(M-1,8);t2++) {
 for(t3=ceil(2*t2-920,921);t3<=floor(M,3684);t3++) {
  for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),
                             3684*t3+3682);t5++) {
   {
    lbv=max(3684*t3,t5+1); ubv=min(M,3684*t3+3683);
    #pragma ivdep
    #pragma vector always
    for(t10=lbv; t10<=ubv; t10++) {
     symmat[t10][t5]=symmat[t5][t10];;
    }
   }
}}}
```

Figure 22: Correlation matrix SCoP (SICA applied with $\rho = 0.9$)

```
for(i=0; i<M; i++)
 for(j=0; j<N; j++)
  S(i,j);
```

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

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

Figure 23: Perfectly nested loop (left) with one level (middle) and two level (right) traditional tiling by PluTo

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

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

Figure 24: Loop from figure 23 with one level (left) and two level (right) SICA tiling for vectorizable j loop