Detecting SIMDization Opportunities through Static/Dynamic Dependence Analysis

Olivier Aumage, Denis Barthou, Christopher Haine, Tamara Meunier

Detecting SIMDization Opportunities through Static/Dynamic Dependence Analysis

Olivier Aumage, Denis Barthou, Christopher Haine, and Tamara Meunier

Inria – University of Bordeaux, France
firstname.lastname@labri.fr

Abstract. Using SIMD instructions is essential in modern processor architecture for high performance computing. Compilers automatic vectorization shows limited efficiency in general, due to conservative dependence analysis, complex control flow or indexing. This paper presents a technique to detect SIMDization opportunities, complementing in a more detailed way compiler optimization reports. The method is based on static and dynamic dependence analysis, able to analyze codes not vectorized by a compiler. This method generates user-hints to help vectorize applications. We show on TSVC benchmark the benefits of this approach.

Keywords: SIMDization, performance tuning, dependence analysis

1 Introduction

For modern multi-core architectures, the Single Instruction, Multiple Data (SIMD) instructions are essential in order to reach high levels of performance. With the increase of vector width – up to 16 floats for Intel Xeon Phi SIMD vectors for instance – SIMD instructions are real performance multipliers. Several options are given to the application developer in order to vectorize a code: explicit vectorization through assembly instructions, intrinsics, GCC vector extensions or other language extensions (such as Intel SPMD Program compiler for instance [20]) or implicit vectorization through the automatic vectorizer of the compiler.

Explicit vectorization, however, is complex and time consuming, assembly and intrinsics-based approaches also are not portable and GCC extensions only offer a limited subset of arithmetic operations. Consequently, the vectorization effort for most applications is delegated to the compiler, which may not entirely succeed or even completely fail to meet the programmer expectations, depending on the code structure and complexity. Indeed, an over-conservative dependence analysis, an incomplete static information concerning the control-flow or a strided data layout are among the main reasons why the compiler may not generate vector code, even though the code actually is vectorizable. Therefore, determining whether the code is vectorizable, independently of any compiler limitations, as well as pinpointing the issues that may hinder vectorization and the transformations required to enable it are critical capabilities for the developer.

This paper proposes a new approach for detecting vectorization opportunities for application innermost loops. This approach is based on a twofold static and dynamic

dependence analysis of the binary code, and generates vectorization hints for the user. By combining a static and dynamic dependence analysis, our method identifies the limiting factors for vectorization, such as misaligned data, non contiguous data together with opportunities to vectorize, obtained through rescheduling, loop transformations, reduction rewriting and vectorizable idiom rewriting. The runtime analysis is essential in capturing dependences in presence of complex control flow or structure indexing. Our technique provides feed-back to the user, pinpointing at the origin line in the source (using debugging DWARF information). This approach is complementary to what compiler optimization reports can provide, bringing a more detailed analysis of vectorization opportunities, in particular of the opportunities missed by the compiler. We show the benefits of the hints generated through a study of the extended TSVC benchmarks[14] (Note: The extended TSVC suite of 151 loops is itself based on the original TSVC suite [3] of 135 loops). While our method primarily targets SIMDization efforts, it would be applicable to more general vector computing hardware.

Section 2 first presents the implementation background we use, Section 3 describes the hybrid dependence analysis, using both static and dynamic analysis. Section 4 uses this analysis to perform SIMDization analysis and generate hints. Finally, we show in Section 5 the results obtained on TSVC benchmarks.

2 MAQAO Background

The analyses proposed in this paper are implemented within MAQAO and use the compression scheme in MAQAO for memory traces. We briefly recall their features used in the following.

MAQAO is a performance tuning tool [2], that analyzes the binary code of applications, written in C/C++ or Fortran. MAQAO builds the control flow graph and the call graph of the code. It proposes an API to instrument statically a binary code and generate a new binary. This instrumentation is able to capture any value in the code, and in particular can be used to trace memory accesses, count loop iterations, capture function parameters. Compared to PIN [12], a tool with similar functionalities, MAQAO performs only static analysis of binaries and static rewriting (from binary to binary). PIN, on the contrary, dynamically rewrites binary codes while they execute, and performs analysis on the fly. As the static analysis of MAQAO is offline, as well the instrumentation process, the overall cost for analyzing a binary with MAQAO is much smaller than with PIN.

MAQAO captures memory streams by instrumenting all instructions that access memory. For each instruction instrumented, the flow of addresses captured is compressed on-the-fly using a lossless algorithm, NLR, designed by Ketterlin and Clauss[8]: The successive values captured by the instrumentation are represented by a program with loops and expressions. The expressions describe memory addresses and depend on the surrounding loop counters. Figure 1 shows an example of traces for a simple code, assuming elements of the arrays 4-byte long floats. Expressions can only depend linearly on the loop counters, and loop bounds only depend linearly on surrounding loop counters. The memory addresses described form union of polytopes. Hence the

method not only captures the memory workingset, it also captures a schedule for the accesses.

```
for(i=1; i<100; i++)
  C[i] = C[i - 1] + B[2 * i];
```

(a) Code example

```
for i0 = 0 to 98
   write 0x2ba1a3bd442c + 4 * i0
```

(b) Compressed trace for `C[i]`

```
for i0 = 0 to 98
   read 0x2ba1a3bd4428 + 4 * i0
```

(c) Compressed trace for `C[i-1]`

```
for i0 = 0 to 98
   read 0x2ba1a4000000 + 8 * i0
```

(d) Compressed trace for `B[2 * i]`

Fig. 1. Example of trace compression using NLR. For the code in (a), one compressed trace per memory access is produced, (b), (c) and (d).

Three important features for these traces are used in this paper: (i) Regular strides are captured. This is important for SIMD optimization, since this will decide whether data layout restructuration is needed or not. In Figure 1.(b), the stride is 4, meaning data of array C is contiguously accessed in this write. For the array B, the stride of 8 shows that one float out of 2 is read, data is not contiguously accessed. (ii) Regular streams are fully traced, in a compact form. This enables the computation of dependence distances. (iii) For multi-dimensional data, memory expressions provide spatial locality information through the ordering of the strides. This can be used in order to propose loop restructuring hints.

For irregular patterns, new loops are created, possibly leading to a trace with no compression if no regularity is found.

3 Hybrid Static/Dynamic Dependence Graph

The dependence analysis we propose is a combination of a static dependence analysis, for registers, and dynamic dependence analysis for memory dependences. The static dependence analysis on registers is already implemented in MAQAO and corresponds to an SSA analysis. Memory dependences are obtained by tracing with MAQAO all memory accesses within the innermost loops, and then computing dependence distances.

Static, Register-Based Dependence Graph. The dependence graph on registers is resulting from an SSA (static single assignment form[4]) analysis, proposed by MAQAO. Besides, MAQAO handles special cases for dependences on x86:

- `xor` instructions, applied twice to the same register, set the value of this register to 0. While the operation is reading a register, this is not considered as a read access.
- SIMD instructions can operate only on the lower or higher part of a SIMD register. Operations that operate on different parts of a register are not considered in dependence.

In addition to the existing analysis, we tag all dependences where a register is read for an address computation. The graph is then partitioned according to these edges

(cutting the graph through these edges), usually in two parts: Instructions preceding these edges are address computation instructions (such as index computation, update of address registers), while instructions after these edges are actual computation (memory accesses, floating point operations, ...) for which SIMDization may be applied. When an indirection occurs, the dependence graph has a path with two tagged edges and can therefore be partitioned into three or more subgraphs. The partition of instructions following all tagged edges is said to be the *computational part of the graph*, while the other instructions are part of the *address computation part* of the graph. In the rare cases where it is not possible to cut the graph following tagged edges, we assume there is no computational part.

Dynamic Dependence Graph. Dynamic dependences are essential to capture what the compiler may have missed, concerning the control flow or the way data structures are indexed. The dynamic dependence graph is built from the memory trace for each read and write instructions in innermost loops.

Algorithm 1 describes how dependence distances are computed. *w.trace* denotes the trace captured for an instruction *w*. For each couple of read and write accesses in a loop, we first perform an interval test, based on their trace (line 2), and then compute a *dependence distance*. The dependence distance is defined as the number of loop iterations between two instructions accessing the same memory location. When two traces have the same loop structure, the substraction between the traces (line 4) substracts the expressions that are at the same position in the trace. If the result is not the same constant value for all substractions, then $*$ is returned, otherwise the constant value is returned. The special $*$ dependence distance notation between two instructions denotes the fact that their dependence distance is not constant during the execution of the program. Note that only uniform dependences are captured this way but as far as SIMDization is concerned, this captures all vectorization opportunities that do not require non-local code transformation.

Algorithm 1: Dynamic Dependence computation for an innermost loop *L*

```
1  for w, a write and r, a read in L do
2      if w.trace ∩ r.trace ≠ ∅ then
3          if loops of w.trace = loops of r.trace then
4              return r.trace − w.trace;
5          else
6              return ∗;
7          end
8      else
9          return 0 ;
10     end
11 end
```

For the example in Figure 1.b and 1.c, both traces have the same structure, the same strides, and the difference between the read and the write addresses is an offset of -4. Then we evaluate how this difference can be compensated by a variation in the loop indices (here $i0$). We find a unique solution within the loop bounds, 1 and this shows that the dependence distance between the write and the read is 1. In general, finding the vector of iteration counters that compensate for the offset between the read and the write leads to a multi-dimensional dependence vector. Only read after write dependences are evaluated, and the sequential order of the assembly code is used to compare relative

```
for (int i = 1; i < LEN2; i++) {
  for (int j = 1; j < LEN2; j++) {
   bb[j][i] = bb[j][i-1] + cc[j][i];
}
```

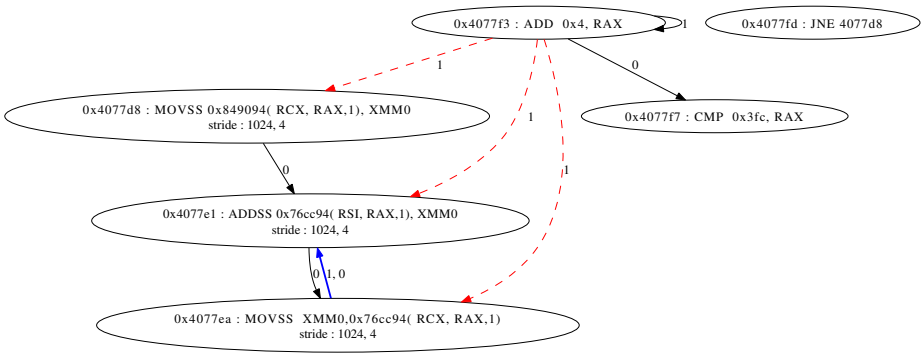

Fig. 2. Code and dependence graph for one loop of function s2233 in TSVC benchmark.

positions for reads and writes. Note that all distances for register dependences correspond in this case to innermost loop carried dependences. Figure 2 presents the s2233 function from TSVC and its dependence graph, combining both the static and dynamic graphs. The nodes each represent an assembly instruction, along with its strides when it is a memory access. The dashed edges in red represent dependences for registers used in address computation. Cutting the graph along these edges separates the computational part (left nodes) from the address computation and control part. The bold blue edge, labelled with $1,0$ represents the memory dependence corresponding to `bb`, directly computed from the trace. The strides denoted on the edges have two values: 1024 and 4. The first one corresponds to the stride for the innermost loop, j, and the second to the i loop. This shows that here, none of the accesses have good spatial locality.

4 SIMDization Analysis

The SIMDization analysis is in two steps: First, we determine whether the code has a parallelism compatible with a SIMDization, independently of any data layout or control limitations (such as large stride). Then, we refine the analysis to detect special cases and accordingly guide the programmer towards enabling and improving the vectorizability of the code.

Vectorizable Dependence Graph. The dependence graph (static and dynamic) is first partitioned according to address computation edges as described previously. The graph is said vectorizable if one of the three conditions applies to the computational part of the graph:

- There is no cycle.
- There is a cycle, with a cumulative weight greater than the width of SIMD vectors.

– There is a cycle, with a cumulative weight smaller than the width of SIMD vectors, and the instructions of the cycle all are of one of the following types: `add`, `mul`, `max`, `min`. The cycle corresponds to a reduction.

A code with a vectorizable dependence graph may require transformations in order to be SIMDizable. This is detailed in the following section.

Code Transformation Hints. We propose to identify a number of transformations required for the SIMDization of the code, depending on the dependence graph, on the stride expressions, and on the control flow graph.

Data alignment: When the graph is vectorizable without cycle, misaligned data is detected by comparing the starting address of all memory streams with the width of SIMD vectors. In the simpler case, the user can either change memory allocation of heap-allocated data structures, or use pragmas for aligning stack-allocated data. When for instance $A[i]$ and $A[i+1]$ occur in the same code, one of the two accesses is misaligned. This would require shuffle instructions, or unaligned loads/stores whenever they exist.

Rescheduling: When the graph is vectorizable without cycle, there may still exist some dependences with non-null distance. Due to the fact that the analysis is performed on assembly code, this may require some modifications at the source code such as some rescheduling of loads/stores and computations, splitting some larger instructions. A template of the vector code is generated by our tool (an example is given in the following section), giving a correct instruction schedule after SIMDization.

Loop transformations: Loop interchange is proposed when all accesses within the loop have a large innermost loop stride, and another loop counter in the expression corresponding to the same outer loop has a stride 4 (for floats and ints). Interchanging these two loops would result in better locality and enable SIMDization.

Loop reversal: Traces with negative stride expressions lead to this hint. Note that if other reads for instance have a positive stride, the reversal is not beneficial any more.

Data reshaping required: This is a fallback hint for large innermost loop strides, and for codes with indirection (detected on the static dependence graph). On the Sandybridge and Xeon Phi architectures, instructions for loading or storing non-consecutive elements into/from SIMD vector have been added to the ISA (GATHER on Xeon Phi and Sandybridge and SCATTER on Xeon Phi). Use of these instructions, through assembly code or intrinsics, is an alternative to data reshaping.

Versioning required: The static analysis on the code may lead to a conclusion different from the trace analysis. For instance, the trace may find a regular stride for a memory stream whereas statically, this stream results from an indirection, or depends on a parameter. Similarly, the dynamic control flow (the real path taken) may be a subset of a more complex static control flow. In these cases, the trace may have captured only one behavior of the code, for a particular input. The vectorization may only be possible in this case through the versioning of the loop, depending on the values of the array, of a parameter.

Idiom Recognition: When the code is vectorizable, with 0 dependence distances or with reductions, the dependence graph can match a predefined dependence graph representing a well known computation. The shape of the dependence graph and the

```
for (int i = 0; i < LEN - 1; i++)
  xx[i+1] = array[i] + a[i];
```

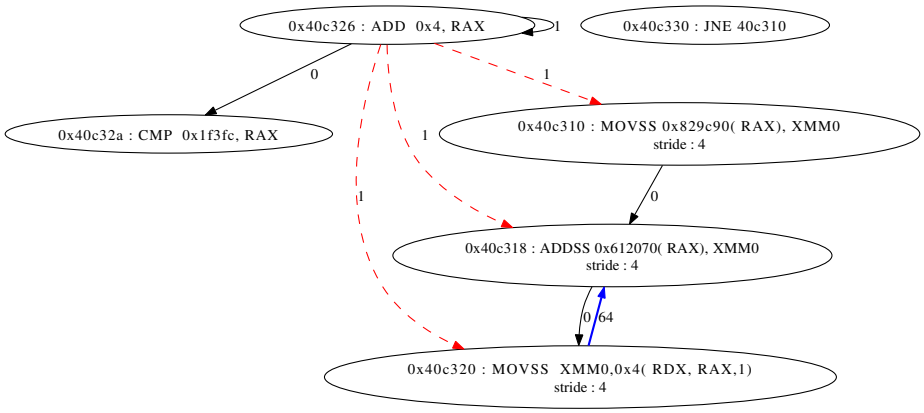

Fig. 3. Code and dependence graph for the loop in function s424 of TSVC benchmark. `array` is aliased with `xx` with an offset of 63 floats.

instructions themselves are matched with the predefined graphs. In this case, the user can call a library function instead of trying to vectorize the actual code.

The predefined functions considered are so far: dot product, daxpy, copy, sparse copy (copy with an indirection either in the load or in the store), but more complex functions can be added with ease.

5 Tests on TSVC Benchmark

The TSVC benchmark has 151 codes with small loops, illustrating different vectorization difficulties. We first present the output generated by our method on one example and then show aggregated results for all TSVC benchmarks.

5.1 Output example

Figure 3 presents a code with an alias between two arrays. This kind of alias can hinder automatic vectorization and only dynamic dependence analysis or possibly interprocedural alias and points-to analysis are able to cope with such situation. Here, the dynamic dependence shows that the dependence distance between the write of `x[i+1]` and the read of `array[i]` is 64 iterations. Thus vectorization is possible as long as vector width is < 64. The output generated by our method is the following:

```
Loop at lines 4443-4444 of tsc.c:
  vectorizable
  contiguous data
  code template:
     load (i:i+4)           line 4444
     load (i:i+4) and add   line 4444
     store (i+64:i+68)      line 4444
```

The source line and name of the file are provided by MAQAO and extracted from debug information. User-friendly names are associated for most frequent instructions found in the computational part of TSVC. The dependence between the store and load is represented by the dependence vector on the indices.

5.2 SIMDization Opportunities

Table 5.2 presents the overall hints generated by the analysis on all TSVC benchmarks. Out of the 151 functions to analyze, 123 are detected as SIMDizable, with 0, 1 or more hints provided by MAQAO. If we focus on the codes that are said vectorizable and are not vectorized by GCC, for 23 of them, speed ups greater than 2 are obtained through hand vectorization, compared to GCC generated codes.

Tool	**Maqao**	**GCC**	**ICC**
Detected vectorizable cases	123	46	104
Corresponding MAQAO hint			
- Reduction	30	15	24
- Idiom	8	3	7
- Data alignment issue	11	4	4
- Code restructuration	53	6	39
- Loop interchange or data tranpose	9	4	7
- Rescheduling	9	1	1
- Control	23	0	17

Table 1. MAQAO vs GCC and Intel ICC compilers

6 Related Works

With the advent of short vector SIMD instructions in modern processors, SIMDizability and automated SIMDization have been the topic of several research efforts. Indeed, making use of these SIMD units has early been recognized as key for performance [11]. Compilers such as from Intel, IBM or PGI, as well as GNU GCC [18, 17, 16, 22] have received much auto-vectorization effort to enable and extend their capabilities [9, 9, 5, 1]. Scout [10] has been designed to vectorize at a higher level, by translating scalar statements to vectorized statements using SIMD intrinsics. However, Malecki *et al.* showed in their SIMDization tests [14] on the TSVC suite that many potentially vectorizable constructs are left unaddressed by state of the art compilers, either because some known theoretical techniques have not yet been implemented, because no known theoretical techniques exist for codes with complex structure, or because the compiler must act conservatively as a consequence to a lack of available information at compile-time [19].

The purpose of tools based on dynamic dependency graph analysis [13] such as our MAQAO approach is therefore to explore SIMDizability from an entirely different

point of view. As such, MAQAO is a complementary analyser with respect to vectorizing compilers, for application programmers to investigate where and how their code could be restructured to enable or improve vectorization by the compiler. Holewinski *et al.* propose a similar approach [7], which is the closest from our own work, to the best of our knowledge so far. However, their implementation does not preserve structural information about the application such as iterated memory references inside loop nests. This limits the amount of details that can be reported to the programmer, and this also limits the richness of the information that could be injected back inside a vectorizing compiler to improve its output. Our approach instead preserves such key information, which distinguishes from prior efforts. An hybrid compile-time/run-time approach is proposed by Nuzman *et al.* [15] using a two-step vectorization scheme. The compile-time step performs expensive analysis operations. The run-time, specialization step is performed by an embedded just-in-time compiler. This approach enables some degree of adaptiveness to the hardware at run-time. However, the run-time step does not alter the vectorizability status using dynamic dependency information. Adaptiveness is also explored by Park *et al.* [6] by guiding the application of optimizations in a predictive manner through a machine-learning approach, and by Tournavitis *et al.* [21] in a technique associating profile-driven auto-parallelization together with machine-learning.

7 Conclusion and Future Work

We have presented in this paper a new method to help users vectorize their code. The approach generates hints identifying what are the vectorization opportunities, and what are the bottlenecks in the code preventing SIMDization. The technique we propose is unique with respect to other efforts on the topic in that it uses a static dependence analysis on the binary code, at the register level, enriched with a dynamic dependence analysis for memory accesses. The combination of both provides a detailed picture of vectorization and possible transformations necessary for SIMDization. We have shown on TSVC benchmark the accuracy of our analysis.

As future work we plan to expand the range and quality of our analysis by detecting other loop transformations automatically (such as loop distribution or reroll) for SIMDization, and also to test this approach to larger, varied application codes.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann (2002)
2. Barthou, D., Rubial, A.C., Jalby, W., Koliai, S., Valensi, C.: Performance tuning of x86 OpenMP codes with MAQAO. In: Tools for High Performance Computing. Springer Berlin Heidelberg (2010)
3. Callahan, D., Dongarra, J., Levine, D.: Vectorizing compilers: a test suite and results. In: Conference on Supercomputing (1988)
4. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (1991)

5. Eichenberger, A.E., Wu, P., O'Brien, K.: Vectorization for SIMD architectures with alignment constraints. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (2004)
6. Eunjung Park, L.N.P., Cavazos, J., Cohen, A., Sadayappan, P.: Predictive modeling in a polyhedral optimization space. In: ACM/IEEE Intl. Conf. on Code Generation and Optimization (2011)
7. Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L.N., Rountev, A., Sadayappan, P.: Dynamic trace-based analysis of vectorization potential of applications. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (2012)
8. Ketterlin, A., Clauss, P.: Prediction and trace compression of data access addresses through nested loop recognition. In: ACM/IEEE Intl. Conf. on Code Generation and Optimization. pp. 94–103. ACM, New York, NY, USA (2008)
9. Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.N., Sadayappan, P.: When polyhedral transformations meet SIMD code generation. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (2013)
10. Krzikalla, O., Feldhoff, K., Muller-Pfefferkorn, R., Nagel, W.E.: Scout: a source-to-source transformator for SIMD-optimizations. In: Workshop on Productivity and Performance (2011)
11. Larsen, S., Amarasinghe, S.: Exploiting superword level parallelism with multimedia instruction sets. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (2000)
12. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (2005)
13. Mak, J., Mycroft, A.: Limits of parallelism using dynamic dependency graphs. In: International Workshop on Dynamic Analysis (2009)
14. Maleki, S., Gao, Y., Garzarn, M.J., Wong, T., Padua, D.A.: An evaluation of vectorization compilers. In: International Conference on Parallel Architectures and Compilation Techniques (PACT) (2011)
15. Nuzman, D., Dyshel, S., Rohou, E., Rosen, I., Williams, K., Yuste, D., Cohen, A., Zaks, A.: Vapor SIMD: Auto-vectorize once, run everywhere. In: ACM/IEEE Intl. Conf. on Code Generation and Optimization (2011)
16. Nuzman, D., Henderson, R.: Multi-platform auto-vectorization. In: ACM/IEEE Intl. Conf. on Code Generation and Optimization (2006)
17. Nuzman, D., Rosen, I., Zaks, A.: Auto-vectorization of interleaved data for SIMD. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (2006)
18. Nuzman, D., Zaks, A.: Autovectorization in GCC-two years later. In: Proceedings of the GCC Developers Summit (2006)
19. Petersen, P.M., Padua, D.A.: Static and dynamic evaluation of data dependence analysis. In: International Conference on Supercomputing (1993)
20. Pharr, M., Mark, W.R.: ispc: A SPMD compiler for high performance CPU programming. In: Conf. InPar (2012)
21. Tournavitis, G., Wang, Z., Franke, B., OBoyle, M.F.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (2009)
22. Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., Rosen, I.: Polyhedral-model guided loop-nest auto-vectorization. In: International Conference on Parallel Architectures and Compilation Techniques (2009)