Automatic SIMD Vectorization of Loops: Issues, Energy Efficiency and Performance on Intel Processors

Olga Moldovanova✉ and Mikhail Kurnosov

Siberian State University of Telecommunications and Information Sciences, Novosibirsk, Russia
Rzhanov Institute of Semiconductor Physics, Siberian Branch of Russian Academy of Sciences, Novosibirsk, Russia
{ovm,mkurnosov}@isp.nsc.ru

Abstract. In this paper we analyse how well compilers vectorize a well-known benchmark ETSVC consisting of 151 loops. The compilers we evaluated were Intel C/C++ 17.0, GCC C/C++ 6.3.0, LLVM/Clang 3.9.1 and PGI C/C++ 16.10. In our experiments we use dual CPU system (NUMA server, 2 x Intel Xeon E5-2620 v4, Intel Broadwell microarchitecture) with the Intel Xeon Phi 3120A co-processor. We estimate time, energy and speedup by running the loops in scalar and vector modes for different data types (double, float, int, short int) and determine loop classes which the compilers fail to vectorize. The Running Average Power Limit (RAPL) subsystem is used to obtain the energy measurements. We analyzed and proposed transformations for the loops that compilers failed to vectorize. After applying proposed transformations loops were successfully auto-vectorized by all compilers. The most part of the transformations based on loop interchange, fission by name and distribution.
Keywords: Loops · Compilers · Automatic Vectorization · CPU Energy Consumption · Intel Xeon · Intel Xeon Phi

1 Introduction

Modern high-performance computer systems are multiarchitectural systems and implement several levels of parallelism: process level parallelism (PLP, message passing), thread level parallelism (TLP), instruction level parallelism (ILP), and data level parallelism (data processing by several vector arithmetic logic units). Processor vendors pay great attention to the development of vector extensions (Intel AVX, IBM AltiVec, ARM NEON SIMD). In particular, Fujitsu announced in its future version of the exascale K Computer system a transition to processors with the ARMv8.2-A architecture, which implements scalable vector extensions. And Intel extensively develops AVX-512 vector extension. That is why problem definitions and works on automatic vectorizing compilers have given the new stage in development in recent decades: OpenMP and Cilk Plus

SIMD directives; Intel ISPC and Sierra language extensions; libraries: C++17 SIMD Types, Boost.SIMD, gSIMD, Cyme.

In this work we studied time, energy and speedup by running the loops in scalar and vector modes for different data types (`double`, `float`, `int`, `short int`) and compilers (Intel C/C++ Compiler, GCC C/C++, LLVM/Clang, PGI C/C++). The main goal is to determine loop classes which the compilers fail to vectorize. The Running Average Power Limit (RAPL) subsystem is used to obtain the energy measurements.

Since there was no information about vectorizing methods implemented in the commercial compilers, the evaluation was implemented by the "black box" method. We used the Extended Test Suite for Vectorizing Compilers [1–4] as a benchmark for our experiments to estimate an evolution of vectorizers in modern compilers comparing to an evaluation made in [1]. We determined classes of typical loops that the compilers used in this study failed to vectorize and evaluated them.

The rest of this paper is organized as follows: Section 2 discusses the main issues that explain effectiveness of vectorization; Section 3 describes the benchmark we used; Section 4 presents results of our experiments; and finally Section 5 concludes.

2 Vector Instruction Sets

Instruction sets of almost all modern processor architectures include vector extensions: MMX/SSE/AVX in the IA-32 and Intel 64 architectures, AltiVec in the Power architecture, NEON SIMD in the ARM architecture family, MSA in the MIPS. Processors implementing vector extensions contain one or several vector arithmetic logic units (ALU) functioning in parallel and several vector registers. Unlike vector systems of the 1990s, modern processors support execution of instructions with relatively short vectors (64-512 bits), loaded in advance from the RAM to the vector registers ("register-register" vector systems).

The main application of the vector extensions consists in decreasing of time of one-dimensional arrays processing. As a rule, a speedup achieved using the vector extensions is primarily determined by the number of array elements that can be loaded into a vector register. For example, each of 16 AVX vector registers is 256-bit wide. This allows loading into them 16 elements of the `short int` type (16 bits), 8 elements of the `int` or `float` type (32 bits) and 4 `double` elements (64 bits). Thus, when using AVX the expected speedup is 16 times for operations with `short int` elements, 8 times for `int` and `float`, and 4 for `double`.

The Intel Xeon Phi processors support AVX-512 vector extension and contain 32 512-bit wide vector registers. Each processor core with the Knights Corner microarchitecture contains one 512-bit wide vector ALU, and processor cores with the Knights Landing microarchitecture have two ALUs.

To achieve a maximum speedup during vector processing it is necessary to consider the microarchitectural system parameters. One of the most important of them is an alignment of array initial addresses (32-byte alignment for AVX and

64-byte alignment for AVX-512). Reading from and writing to unaligned memory addresses is executed slower. Effectiveness decreasing can also be caused by a mixed usage of SSE and AVX vector extensions. In such a case during transition from execution of one vector extension instructions to another one a processor stores (during transition from AVX to SSE) or restores (in another case) highest 128 bits of `YMM` vector registers (AVX-SSE transition penalties) [5].

When vector instructions are used, the achieved speedup can exceed the expected one. For example, after vectorization of the loop, which calculates an elementwise sum of two arrays, the processor overhead decreases due to reducing the number of add instruction loads from the memory and its decoding by the processor; the number of memory accesses for operands of the add instruction; the amount of calculations of loop end condition (the number of accesses to the branch prediction unit of the processor).

Besides that, a parallel execution of vector instructions by several vector ALUs can be a reason of additional speedup. Thus, an efficiently vectorized program overloads subsystems of a superscalar pipelined processor in a less degree. This is the reason of less processor energy consumption during execution of a vectorized program as compared to its scalar version [6].

Application developers have different opportunities to use vector instructions:

- inline assembler – full control of vectorization usage, least portable approach;
- intrinsics – set of data types and internal compiler functions, directly mapping to processor instructions (vector registers are allocated by compiler);
- SIMD directives of compilers, OpenMP and OpenACC standards;
- language extensions, such as Intel Array Notation, Intel ISPC, Apple Swift SIMD and libraries: C++17 SIMD Types, Boost.SIMD, SIMD.js;
- automatic vectorizing compiler – ease of use, high code portability.

In this work, we study the last approach. Such vectorizing technique does not require large code modification and provides its portability between different processor architectures.

3 Related Works and Benchmarks

We used the Extended Test Suite for Vectorizing Compilers (ETSVC) [2] as a benchmark containing main loop classes, typical for scientific applications in C language. The original package version was developed in the late 1980s by the J. Dongarra's group and contained 122 loops in Fortran to test the analysis capabilities of automatic vectorizing compilers for vector computer systems: Cray, NEC, IBM, DEC, Fujitsu and Hitachi [3, 4]. In 2011 the D. Padua's group translated the TSVC suite into C and added to it new loops [1]. The extended version of the package contains 151 loops. The loops are divided into categories: dependence analysis (36 loops), vectorization (52 loops), idiom recognition (reductions, recurrences, etc., 27 loops), language completeness (23 loops). Besides that, the test suite contains 13 "control" loops, trivial loops that are expected to be vectorized by every vectorizing compiler.

The loops operate on one- and two-dimensional 16-byte aligned global arrays. The one-dimensional arrays contain $125 \cdot 1024/$`sizeof(TYPE)` elements of the given type `TYPE`, and the two-dimensional ones contain 256 elements by each dimension.

Each loop is contained in a separate function (see Listing 1). In the `init` function (line 5) an array is initialized by individual for this test values before loop execution. The outer loop (line 7) is used to increase the test execution time (for statistics issues). A call to an empty `dummy` function (line 10) is used in each iteration of the outer loop so that, in case where the inner loop is invariant with respect to the outer loop, the compiler is still required to execute each iteration rather than just recognizing that the calculation needs to be done only once [4].

After execution of the loop is complete, a checksum is computed by using elements of the resulting array (`check` function, line 16).

Listing 1. Example loop from the ETSVC benchmark

```
1  #define TYPE float
2  #define lll (125 * 1024 / sizeof(TYPE))
3  #define ntimes 200000
4  int s000() {
5      init("s000 ");
6      clock_t start_t = clock();
7      for (int nl = 0; nl < 2 * ntimes; nl++) {
8            for (int i = 0; i < lll; i++)
9                X[i] = Y[i] + 1;
10         dummy((TYPE*)X, (TYPE*)Y, (TYPE*)Z, (TYPE*)U,
11               (TYPE*)V, aa, bb, cc, 0.0);
12     }
13     clock_t end_t = clock(); clock_dif = end_t - start_t;
14     printf("S000\t %.2f \t\t",
15            (double)(clock_dif / 1000000.0));
16     check(1);
17     return 0;
18 }
```

4 Results of Experiments

4.1 Test Environment

We used two systems for our experiments. The first system was a server based on two Intel Xeon E5-2620 v4 CPUs (Intel 64 architecture, Broadwell microarchitecture, 8 cores, Hyper-Threading was on, AVX 2.0 support), 64 GB RAM DDR4, GNU/Linux CentOS 7.3 x86-64 operating system (linux 3.10.0-514.2.2.el7 kernel). The second system was Intel Xeon Phi 3120A co-processor (Knights Corner microarchitecture, 57 cores, AVX-512 support, 6 GB RAM, MPSS 3.8) installed in the server.

The compilers evaluated in these experiments were Intel C/C++ Compiler 17.0; GCC C/C++ 6.3.0; LLVM/Clang 3.9.1; and PGI C/C++ 16.10. The vectorized version of the ETSVC benchmark was compiled with the command line options shown in Table 1 (column 2). To generate the scalar version of the test suite the optimization options were used with the disabled compilers vectorizer (column 3, Table 1).

Table 1. Compilers options

Compiler	Compilers Options	Disabling Vectorizer
Intel C/C++ 17.0	`-O3 -xHost -qopt-report3 -qopt-report-phase=vec,loop -qopt-report-embed`	`-no-vec`
GCC C/C++ 6.3.0	`-O3 -ffast-math -fivopts -march=native -fopt-info-vec -fopt-info-vec-missed`	`-fno-tree-vectorize`
LLVM/Clang 3.9.1	`-O3 -ffast-math -fvectorize -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize`	`-fno-vectorize`
PGI C/C++ 16.10	`-O3 -Mvect -Minfo=loop,vect -Mneginfo=loop,vect`	`-Mnovect`

32-byte aligned global arrays were used for the Intel Xeon processor, and 64-byte aligned global arrays were used for the Intel Xeon Phi processor. We used arrays with elements of `double`, `float`, `int` and `short` data types for our evaluation.

4.2 Results for Intel 64 Architecture

The following results were obtained for the `double` data type on the Intel 64 architecture (Intel Xeon Broadwell processor). The Intel C/C++ Compiler vectorized 95 loops in total, 7 from which were vectorized by it alone. For GCC C/C++ the total amount of vectorized loops was 79. But herewith there was no loop that was vectorized only by this compiler. The PGI C/C++ vectorized the largest number of loops, 100, 13 from them were vectorized by it alone. The minimum number of loops was vectorized by the LLVM/Clang compiler, 52, 4 from which were vectorized only by it. The number of loops unvectorized by any compiler was equal to 28.

We compared the obtained results with the evaluation done in [1]. The comparison shows that the vectorizer of the GCC C/C++ compiler has been significantly improved: 52.3 % of vectorized loops from ETSVC in 2017 versus 32 % in 2011 (see Table 2).

The similar results were obtained for arrays with elements of the `float` and `int` types by all compilers. The consistent results were obtained for the `short`

Table 2. Comparison of results with previous evaluations of compilers

2011 (Padua et al. [1])		**2017 (our work)**	
Intel C/C++ 12.0	90 loops (59.6 %)	**Intel C/C++ 17.0**	95 loops (62.9 %)
GCC C/C++ 4.7.0	59 loops (39 %)	**GCC C/C++ 6.3.0**	79 loops (52.3 %)

type when Intel C/C++ Compiler, GCC C/C++ and LLVM/Clang were used. The exception to this rule was the PGI C/C++ compiler that vectorized no loops processing data of this type.

Figure 1 shows the results of loop vectorization for the `double` data type on the Intel 64 architecture. Abbreviated notations of the vectorization results are shown in the table cells. They were obtained from vectorization reports of compilers for all 151 loops. The full form of these notations is shown in Table 3. The similar results were obtained for other data types.

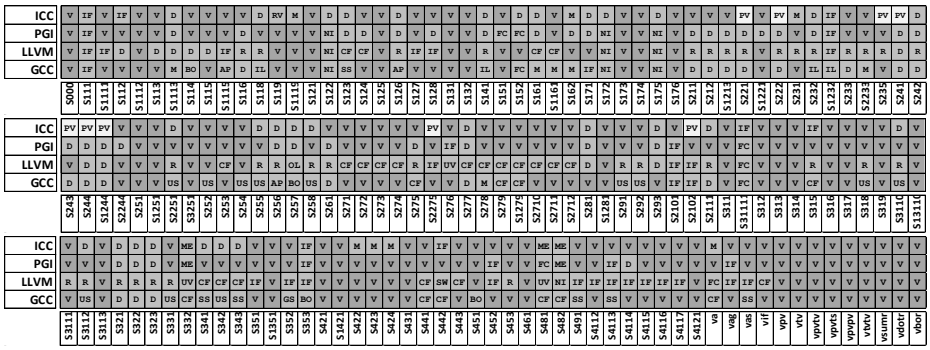

Fig. 1. Results of loops vectorization (Intel 64 architecture, `double` data type)

In the "Dependence analysis" category 9 loops were not vectorized by any compiler for the `double` data type. The compilers used in this study failed to vectorize loops with linear dependences (1st order recurrences), induction variables together with conditional and unconditional (`goto`) branches, loop nesting and variable values of lower and/or upper loop bounds and/or iteration step. In the last case, no compiler could determine whether a data dependence was present and took a pessimistic decision that the dependence existed.

In the "Vectorization" category the compilers failed to vectorize 11 loops. These loops required transformations as follows: loop fission, loop interchange, node splitting (to avoid cycles in data dependence graphs and output and anti-dependences [7]) and array expansions. Among causes of problems were interdependence of iteration counts of nested loops; linear dependencies in a loop body (1st order recurrences); conditional and unconditional branches in a loop body.

Table 3. Abbreviated notations of vectorization results

V	Loop is vectorized
PV	Partial loop is vectorized (loop fission with succeeding vectorization of obtained loops)
RV	Remainder is not vectorized
IF	Vectorization is possible but seems inefficient
D	Vector dependence prevents vectorization (supposed linear or non-linear data dependence in a loop)
M	Loop is multiversioned (multiple loop versions are generated, unvectorized version is selected in runtime)
BO	Bad operation or unsupported loop bound (e.g., `sinf` or `cosf` function is used)
AP	Complicated access pattern (e.g., value of iteration count is more than 1)
R	Value that could not be identified as function is used outside the loop (induction variables are present in a loop)
IL	Inner-loop count not invariant (e.g., iteration count of inner loop depends on iteration count of outer loop)
NI	Number of iterations cannot be computed (lower and/or upper loop bounds are set by function's arguments)
CF	Control flow cannot be substituted for a select (conditional branches inside loop)
SS	Loop is not suitable for scatter store (e.g., in case of packing a two-dimensional array into a one-dimensional array)
ME	Loop with multiple exits cannot be vectorized (`break` or `exit` are present inside a loop)
FC	Loop contains function calls or data references that cannot be analyzed
OL	Value cannot be used outside the loop (scalar expansion or mixed usage of one- and two-dimensional arrays in one loop)
UV	Loop control flow is not understood by vectorizer (conditional branches inside a loop)
SW	Loop contains a `switch` statement
US	Unsupported use in statement (scalar expansion, wraparound variables recognition)
GS	No grouped stores in basic block (unrolled scalar product)

The following idioms (6 loops) from the "Idiom recognition" category were not vectorized by the compilers used: 1st and 2nd order recurrences, array searching, loop rerolling and reduction with function calls. The loops with recurrences were not vectorized because of linear data dependence. In a loop with array searching for the first element meeting a condition the unconditional branch `goto` prevented vectorization.

Compilers execute rerolling for loops that were unrolled by hand before vectorization [8]. The compilers in this study decided that vectorization of such loops was possible but inefficient. The reason was an indirect addressing in array elements access: $X[Y[i]]$, where X is a one-dimensional array of the `float`

type, Y is a pointer to a one-dimensional array of integers, i is a loop iteration count.

The next challenging idiom was a reduction, namely sum of elements of a one-dimensional array. In this case the idiom was not vectorized because of test function calls. This function calculated sum of 4 array elements beginning from the one passed as the function argument. The Intel C/C++ Compiler reported that vectorization was possible but inefficient. Other compilers reported a function call as a reason of vectorization failing.

The "Language completeness" category contain 2 loops unvectorized by any compiler. The problem of both loops consisted in breaking loop computations (exit in the first case and break in the second case). Compiler vectorizers could not analyze control flow in these loops.

Total execution time of the benchmark (all loops) for each data type and compiler is shown in Figure 2. A median value and maximum speedups of vectorized loops are shown in Figs. 3 and 4. The maximum speedup obtained on the Intel 64 architecture by the Intel C/C++ was 6.96 for the `double` data type, 13.89 for the `float` data type, 12.39 for `int` and 25.21 for `short int`. The maximum speedup obtained by GCC C/C++ was equal to 4.06, 8.1, 12.01 and 24.48 for types `double`, `float`, `int` and `short int`, correspondingly. The LLVM/Clang obtained results as follows: 5.12 (`double`), 10.22 (`float`), 4.55 (`int`) and 14.57 (`short int`). For PGI C/C++ these values were 14.6, 22.74, 34.0 and 68.0, correspondingly. The speedup is the ratio of the running time of the scalar code over the running time of the vectorized code.

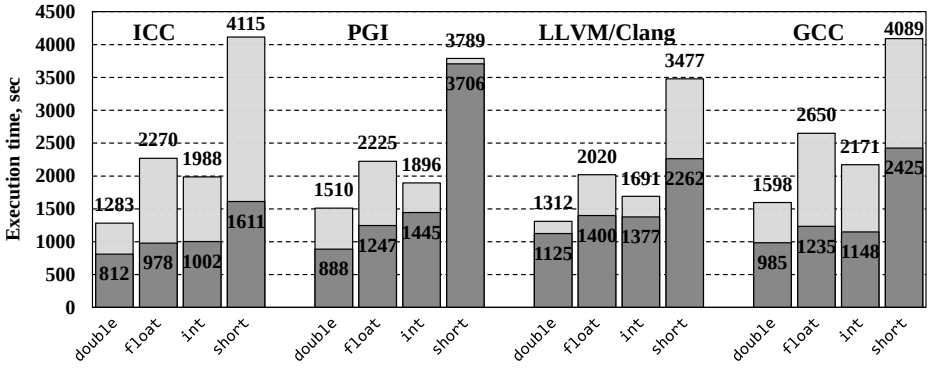

Fig. 2. Execution time of the benchmark (all loops) on the Intel Xeon E5-2620 v4 CPU: *outer columns* – scalar version of the benchmark; *inner columns* – vectorized version of the benchmark

As our evaluation showed maximum speedups for Intel C/C++ Compiler, GCC C/C++ and LLVM/Clang correspond to the loops executing reduction operations (sum, product, minimum and maximum) with elements of one-dimensional arrays of all data types. These loops belong to the "Idiom recognition" cat-

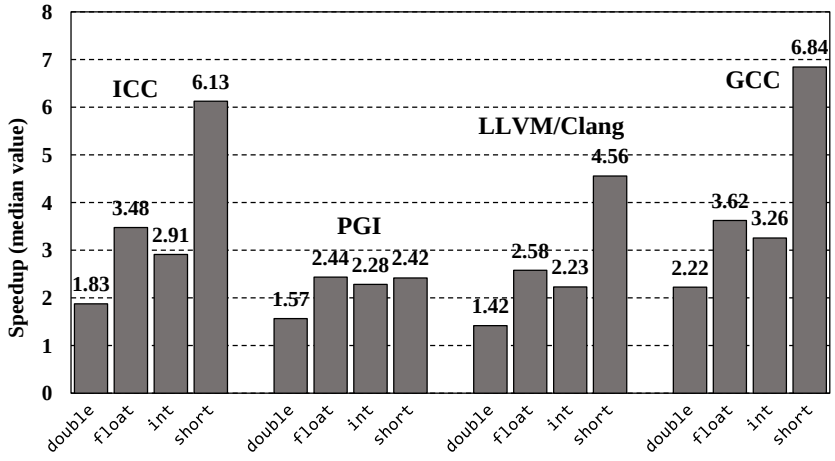

Fig. 3. Median value of speedup for vectorized loops on the Intel Xeon E5-2620 v4 CPU (only speedups above 1.15 were considered)

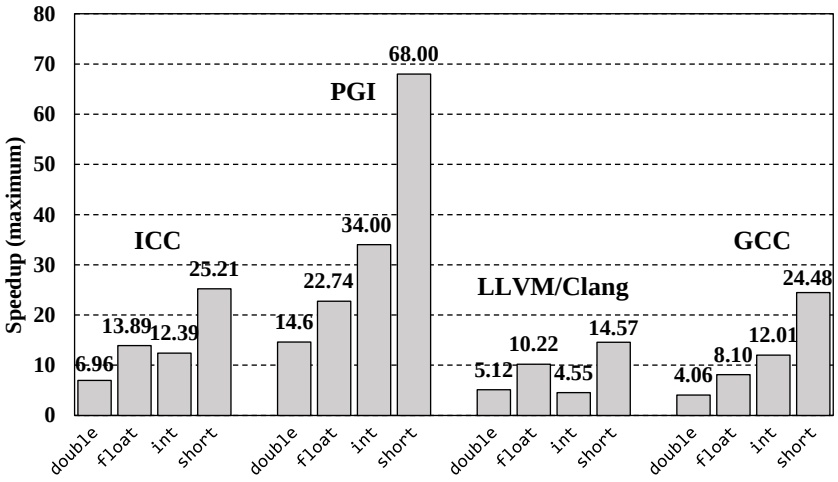

Fig. 4. Maximum speedup for vectorized loops on the Intel Xeon E5-2620 v4 CPU

egory in the ETSVC. For PGI C/C++ maximum speedup was achieved for the loop calculating an identity matrix ("Vectorization" category) for the `double` and `float` data types. And for `int` and `short` this value was obtained in the loop calculating product reduction ("Idiom recognition" category).

However, the obtained speedup is not always a result of vectorization. For the PGI C/C++ compiler the speedup value 68.0 for the `short` data type can be explained by the fact that calculations in a loop are not executed at all because of the compiler optimization.

4.3 Results for Intel Xeon Phi Architecture

On the Intel Xeon Phi architecture we studied vectorizing capabilities of the Intel C/C++ Compiler 17.0. The `-mmic` command line option was used instead of the `-xHost` during compilation. The results of the experiments for two data types are shown in Fig. 5. The compiler could vectorize 99 loops processing data of the `double` type and 102 of the `float` type. Supposed data dependencies (28 loops for the `double` type and 27 for the `float` type) were the main reason of loop vectorization failing. 12 loops were partially vectorized for both data types. Similar results were obtained for the `int` and `short` types.

In this case the maximum speedup for the `double` type was 13.7, for `float` – 19.43, `int` – 30.84, and `short` – 46.3. For `float` and `short` maximum speedups were obtained for loops executing reduction operations for elements of one-dimensional arrays. For the `double` data type `sinf` and `cosf` functions were used in a loop. In the case with `int` it was a "control" loop `vbor` calculating a scalar product of six one-dimensional arrays.

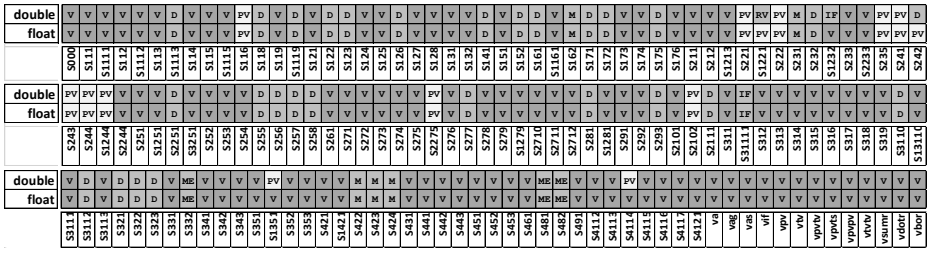

Fig. 5. Results of loops vectorization (Intel Xeon Phi architecture)

4.4 Effect of Vectorization on CPU Energy Consumption

We modified the ETSVC benchmark to measure the CPU (Intel Xeon E5-2620v4) energy for each loop. The measurements were accomplished by using the Intel RAPL (Running Average Power Limit) subsystem before and after each loop execution. We requested information about total CPU energy consumption (RAPL PKG domain) and DRAM controller energy consumption (RAPL DRAM domain) from the RAPL subsystem.

For every loop we determined the decrease E of CPU energy consumption (RAPL PKG domain) for vectorized loop execution against its scalar version execution:

$$E = (E_{novec} - E_{vec})/E_{novec} \cdot 100\%, \tag{1}$$

where E_{novec} is CPU energy for scalar loop execution ($[E_{novec}] = \text{J}$), E_{vec} is CPU energy for vectorized loop execution ($[E_{vec}] = \text{J}$).

In Table 4 we show the results for the decrease E of CPU energy consumption only for successfully vectorized loops, execution time of which is less than execution time of their scalar versions at least on 15%.

For arrays with elements of `double` type vectorized loops decreased the CPU energy consumption by a mean of 45% as compared to their scalar versions. For `float`, `int` and `short int` the CPU energy consumption decrease was 64%, 58% and 90%, correspondingly.

It is apparent that for the ETSVC benchmark increasing the number of array elements which can be loaded into a vector register (due to decreasing the size of data type) results in decreasing the CPU energy consumption.

Table 4. Statistical characteristics for the the decrease E of CPU energy consumption (for successfully vectorized loops, execution time of which is less than execution time of their scalar versions at least on 15%)

Compiler	Data Type	Min, %	Max, %	Avg, %	Median, %
Intel C/C++ 17.0	`double`	13	85	42	41
	`float`	14	91	64	70
	`int`	13	92	62	65
	`short`	70	99	94	96
GCC C/C++ 6.3.0	`double`	17	75	52	60
	`float`	16	99	71	73
	`int`	13	91	67	70
	`short`	13	96	80	85
LLVM/Clang 3.9.1	`double`	17	79	37	28
	`float`	21	99	62	59
	`int`	26	77	50	52
	`short`	46	99	92	96
PGI C/C++ 16.10	`double`	11	93	48	39
	`float`	15	96	60	58
	`int`	10	96	52	54
	`short`	52	99	92	96

5 Conclusion

In this work we studied auto-vectorizing capabilities of modern optimizing compilers Intel C/C++ Compiler, GCC C/C++, LLVM/Clang, PGI C/C++ on the Intel 64 and Intel Xeon Phi architectures. Our study shows that the compilers evaluated could vectorize 39-77 % of the total number of loops in the ETSVC package. The best results were shown by the Intel C/C++ Compiler, and the worst ones – by the LLVM/Clang compiler. The compilers failed to vectorize loops containing conditional and unconditional branches, function calls, induction variables, variable loop bounds and iteration count, as well as such idioms as 1st or 2nd order recurrences, search loops and loop rerolling. We analyzed and

proposed transformations for the loops that compilers failed to vectorize. After applying proposed transformations loops was successfully auto-vectorized by all compilers. The most part of the transformations based on loop interchange, fission and distribution.

We estimated the CPU energy consumption for execution of vectorized loops against their scalar versions. The experiments show that increasing the number of array elements which can be loaded into a vector register (due to decreasing the size of data type) results in decreasing the CPU energy consumption.

The future work will consist of evaluation and development of vectorizing methods (polyhedral model) for the obtained class of challenging loops, applicability analysis of JIT compilation [9] and profile-guided optimization.

References

1. Maleki, S., Gao, Ya. Garzarán, M.J., Wong, T., Padua, D.A.: An Evaluation of Vectorizing Compilers. Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 372–382 (2011)
2. Extended Test Suite for Vectorizing Compilers. URL: `http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz`
3. Callahan, D., Dongarra, J., Levine, D.: Vectorizing Compilers: A Test Suite and Results. Proc. of the ACM/IEEE Conf. on Supercomputing, 98–105 (1988)
4. Levine, D., Callahan, D., Dongarra, J.: A Comparative Study of Automatic Vectorizing Compilers. Journal of Parallel Computing. Vol. 17, 1223–1244 (1991)
5. Konsor, P.: Avoiding AVX-SSE Transition Penalties. URL: `https://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties`
6. Jibaja, I., Jensen, P., Hu, N., Haghighat, M., McCutchan, J., Gohman, D., Blackburn, S., McKinley, K.: Vector Parallelism in JavaScript: Language and Compiler Support for SIMD. Proc. of the Int. Conf. on Parallel Architecture and Compilation Techniques, 407–418 (2015)
7. Program Vectorization: Theory, Methods, Implementation (1991)
8. Metzger, R.C., Wen, Zh.: Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization (2000)
9. Rohou, E., Williams, K., Yuste, D.: Vectorization Technology To Improve Interpreter Performance. ACM Trans. on Architecture and Code Optimization, 9(4), 26:1-26:22 (2013)

Acknowledgement.This work is supported by Russian Foundation for Basic Research (projects 16-07-00992, 15-07-00653).