Performance and efficiency investigations of SIMD programs of Coulomb solvers on multi- and many-core systems with vector units

Ronny Kramer
Department of Computer Science
Chemnitz University of Technology
09107 Chemnitz, Germany
kramr[at]cs.tu-chemnitz.de

Gudula Rünger
Department of Computer Science
Chemnitz University of Technology
09107 Chemnitz, Germany
ruenger[at]cs.tu-chemnitz.de

To cite this version:
Kramer R., Rünger G.: Performance and efficiency investigations of SIMD programs of Coulomb solvers on multi- and many-core systems with vector units. In: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing – PDP 2020, (2020), ISBN: 978-1-7281-6582-0, IEEE Computer Society Conference Publishing Services (CPS)

Abstract

Vectorization is a key technology to increase the performance and efficiency of computer systems, however the performance increase usually strongly depends on the specific application program. In this article, we introduce SIMD program versions of a Coulomb solver based on manual vectorization as well as automatic vectorization using the Intel C Compiler. The implementation uses the latest Advanced Vector Extensions with 512-bit vector size (AVX-512). The comparison of the performance results visualizes the reduction of the execution time achieved by the manual vectorization as well as the automatic vectorization compared to the code without vectorization. The automatic vectorization achieves a good speed up that should be sufficient for most use cases. However, we also show that the manual vectorization is able to outperform the automatic vectorization. The control over the instructions used by the manual vectorization can reduce the amount of expensive vector reduction operations which are introduced by the automatic vectorization.

Code Optimization, SIMD, Vectorization, AVX-512, HPC, Intel Xeon Phi, Intel C Compiler

1 Introduction

Vendors of Central Processing Units (CPUs) are constantly improving their designs for higher compute performance and better energy efficiency. One method to achieve this goal is to add and expand function units that are specialized to perform certain tasks. To increase the computational throughput a function unit called the Vector Processing Unit (VPU) has been introduced. A VPU enables a CPU to perform a single instruction on multiple data elements (SIMD) simultaneously.

The Advanced Vector Extensions (AVX) are a set of SIMD instructions that allow the execution of operations utilizing a VPU. These SIMD instructions are accessible as assembler instruction or as higher level functions, which are called intrinsics. AVX is available in different versions; AVX-512 is the most recent version and is used in this article. AVX-512 introduces a vector size of 512-bit which enables the simultaneous execution of eight double precision floating point calculations.

For the creation of a SIMD versions for a given scalar program, there are two main approaches, which are a manual vectorization or an automatic vectorization. The problem with the manual vectorization using intrinsics is that it can be quite time consuming, error prone and may produce a lot of code that is hard to maintain. On the other hand, during the translation of program code to machine code a compiler is able to perform optimizations based on heuristics. The automatic vectorization is possible since modern compilers, such as the Intel C Compiler (ICC), are able to determine areas of the code that can be vectorized and to perform the vectorization automatically.

The goal of this article is to investigate for a real world application whether the manual vectorization using intrinsics is necessary or if the automatic vectorization provided by the ICC achieves a similarly good speedup. The application investigated in this article is a particle simulation using Coulomb forces based on ScaFaCoS, which is a library of scalable fast Coulomb solvers, including a solver to directly compute the particle-particle interaction as well as several other approximation solvers. The direct solver is particularly suited for the investigation in this article, since its algorithmic structure requires complex computations but its memory accesses are purely regular [1].

The original particle code of the ScaFaCos library had to be prepared in order to make an automatic vectorization possible. The preparation step includes program transformations, such as loop interchange, loop fusion and unswitching to improve the code as well as data flow. All loops have been modified to avoid any conditions that terminate the loops prematurely. Especially, the fusion of loops has improved the compatibility to larger vector sizes. The starting version of the investigation in this article is this implementation of the Direct solver that is already optimized and improves the cache usage and branch prediction of the original ScaFaCoS implementation. Additionally, a manually vectorized implementation designed for Intel Xeon Phi coprocessor, a legacy expansion card that was intended to increase the computation capabilities of workstations and servers, is available.In this article, both implementations with and without manual vectorization are used and executed on a newer architectures supporting AVX-512. The goal is to investigate how such a legacy manually vectorized legacy program code performs compared to an automatically vectorized version using modern compilers. Furthermore, it is described how the manually vectorized program code can be improved further by changing the implementation of reduction operations.

The organization structure of the article is as follows: Related work is presented in Section 2. The direct solver is introduced in Section 3, including an overview of implementation improvements performed in Section 3.2. Section 4 describes a general approach to improve the performance when manual vectorization is combined with the parallel reduction provided by OpenMP. The different vectorized implementations to be investigated are presented in Section 5. Section 6 describes the hardware used as benchmark platform as well as the compilation parameters. Section 7 compares different compiler versions as well as the instructions sets for the automatically vectorized program version. The comparison of the execution times of the automatically vectorized version and the manually vectorized version using intrinsics is given in Section 8 with an emphasis on the performance optimisation related to the reduction improvements in Section 8.2. Section 9 provides concluding remarks.

2 related work

Vectorization is an important topic to improve the efficiency and reduce the execution time of calculations. Especially AVX-512 provides new possibilities and challenges for improvements. A introduction into Advanced Vector Extensions (AVX), including AVX-512, explaining the programming concepts as well as optimization strategies and techniques is given [2]. Also, the Intel 64 and IA-32 Architectures Optimization Reference Manual [3] is a good source for optimization strategies as well as a comparisons between the older AVX2 and more recent AVX-512.

The scalability of AVX-512 has been investigated in [4]. Several SIMD benchmarks have been executed using different vectorization instruction sets to compare the speedup and energy reduction. It was shown that not all applications achieved a linear speedup when being vectorized. It was also shown that even without a speedup, the energy efficiency can be improved by the vectorization.

Several investigations have studied specific applications in the context of vectorization. In [5], it is shown that the sparse matrix-vector product can be calculated efficiently using AVX-512 instructions also covering block-based sparse matrix formats as well as optimization routines for block sizes. A quicksort algorithm vectorized using AVX-512 is described in [6]. The work shows that the SIMD quicksort algorithm outperforms two reference implementations by GNU and Intel.

The effect of irregular memory accesses on vectorization was investigated in [7]. A comparable investigation on similar hardware based on a lattice Quantum Chromodynamics simulation using AVX-512 is investigated in [8].

The advantage of AVX-512 over AVX2 for shallow water solvers can be seen in [9]. That AVX-512 vector instructions do not always provide the best performance over older instruction sets is shown in [10]. The work evaluates different SIMD versions for the local sequence alignment using the Smith-Waterman algorithm which enables a faster identification of genetic variants associated with diseases. To find the best vectorization method for this use case, AVX-512 in two different sub-variants are compared against the AVX2 as well as SSE4.1. The result for this use case is that the older AVX2 instruction set is the best choice with respect to performance.

3 Direct summation solver

3.1 Basic version of the direct solver

ScaFaCoS is a library of scalable fast Coulomb solvers [1]. One of the solvers included in the library is the so-called "Direct summation" solver, in the following referred to as direct solver. The direct solver is based on the calculation of pairwise interactions between all given particles. The part of this direct solver that requires most of the execution time is the evaluation on fully periodic systems. To perform these evaluations, the direct solver implements the calculation of the electrostatic potential ϕ and the electrostatic field $\mathbf{E}$ as lattice summations:

$$\phi(\mathbf{x}_j) = \sum_{\mathbf{r}\in\mathbb{Z}^3} \sum_{\substack{l=1 \\ l\neq j \text{ for } \mathbf{r}=\mathbf{0}}}^{M} \frac{q_l}{\|\mathbf{x}_j - \mathbf{x}_l + \mathbf{r}B\|_2}, j = 1, \ldots, M$$

$$\mathbf{E}(\mathbf{x}_j) = -\sum_{\mathbf{r}\in\mathbb{Z}^3} \sum_{\substack{l=1 \\ l\neq j \text{ for } \mathbf{r}=\mathbf{0}}}^{M} q_l \frac{\mathbf{x}_j - \mathbf{x}_l + \mathbf{r}B}{\|\mathbf{x}_j - \mathbf{x}_l + \mathbf{r}B\|_2^3}, j = 1, \ldots, M$$

The M charged particles given as input are stored in a cubic simulation box with edge length B. For each particle, a charge $q_l \in \mathbb{R}$ as well as its location at $\mathbf{x} \in [0, B)^3$ is stored, see [11–13].

The direct solver provided by the ScaFaCoS is a parallel implementation for distributed memory using the Message Passing Interface (MPI), which has been developed by experts in the application field of particle simulation. The algorithmic structure of this program version is the starting point of our investigation.

3.2 Shared memory implementation of the direct solver

Vectorization for multi- or many-cores requires a mixed programming model exploiting the different cores by multi-threaded programs and additionally exploiting the vectorization units on each core. To perform an efficient vectorization an optimal parallel code base is crucial. First, the code needs to be parallelized to enable the usage of all vector units, since the vectorization is limited to one vector unit per thread. For a vectorization, a code base can be considered optimal if the CPU is able to incrementally read from continuous memory while minimizing the memory access and synchronization operations.

In case of the ScaFaCoS library a parallelization using message passing on distributed memory via the Message Passing Interface (MPI) exists. The problem with using distributed memory implementations for shared memory systems are memory transactions related to the message passing, which are unnecessary for shared memory systems and usually lead to an unnecessary overhead especially for additional vectorization. Thus, the direct solver implementation from the ScaFaCos library is ported to a shared memory parallelization without changing the algorithmic and numerical properties.

The direct solver described above has a large potential of shared memory parallelization. With the exception of the summation itself, which can be performed via a parallel reduction operations, all iterations within the summations are independent from each other. Thus, all iterations can be performed in parallel to each other and synchronizations are limited to the reduction operation. A shared memory implementation using OpenMP is possible though the `#pragma omp parallel for`. All temporary variables can be defined as `private` and all input data have only read accesses and can therefore be defined as `shared` without any access protection. The results calculated for the electrostatic potential ϕ and the electrostatic field $\mathbf{E}$ are reduced by `reduction(+:`ϕ`,`$\mathbf{E}$`)`

In terms of the requirement for a continuous memory access, the OpenMP direct solver is already well suited and no further changes are required. However, to minimize the memory accesses the temporal locality has been improved by altering the order of the loops. By calculating the boundary conditions for each particle before continuing with the calculation for the next particle, the data already loaded from the memory can be reused.

To support the CPU in the decision finding for branch predictions and memory preloading, the spatial locality of the direct solver has also been improved. The original implementation of the direct solver has two situations in which the CPU performs a branch prediction that can be avoided. The first situation is the $l \neq j$ for $\mathbf{r} = \mathbf{0}$ condition in Formula 3.1, which can be avoided by precalculating the $\mathbf{r}B$ values. Due to the precalculation the values are provided as continuous memory over which the solver iterates and, thus, the need for this check is removed. The second situation in which a branch prediction is required is unrelated to the algorithm but related to a debug feature introduced to ease the comparison of other solvers to the direct solver. To keep the debug feature available, the solver was split into a fallback code including this debug and the actual direct solver without this debug feature.

4 Reduction Optimization

The direct solver requires several reduction operations since the influence of a particle system on a single particle is expressed by four values: the potential as well the force in three dimension. The assembler output of the compiler shows that most intrinsics, such as `_mm512_add_pd` used to add two 512-bit vectors, provide their functionality as one single machine operation. However, some intrinsics, such as `_mm512_reduce_add_pd` used to reduce a 512-bit vector to one scalar, require multiple machine operations to provide their functionality. In the case of `_mm512_reduce_add_pd`, the compiler implements the reduction using one shuffle, two permutations and three vector additions. As the vector reduction operation is six times as expensive as the other operations required for the calculation, the efficiency can be improved by minimizing the amount of vector reductions.

As explained in Section 3.2 the algorithm performs the calculation of the boundary condition without the usage of those intermediate results. Therefore, it is possible to use a vector to store the intermediate results. This however is prevented by the OpenMP summation used in the shared memory parallelization which is only defined for scalars. By declaring a custom reduction operation using `#pragma omp declare reduction`, a support for the reduction of two vectors to one can be added to the OpenMP parallelisation. As the ScaFaCos library expects the direct solver to return scalars, a final vector reduction is required to be performed. This reduction can happen outside the boundary condition loop as shown in the following example:

```
#pragma omp declare reduction (mm512_add_pd : __m512d : omp_out
     = _mm512_add_pd(omp_out, omp_in))
#pragma omp for reduction(mm512_add_pd:p)
for (...) p = _mm512_add_pd(p, partial_value);
p_sum += _mm512_reduce_add_pd(p);
```

This improvement reduces the amount of vector reductions from n to 1 for each particle-particle interaction. This reduces the complexity of the reduction from $\mathcal{O}(n*6)$ for the original reduction down to $\mathcal{O}(n+6)$ operations. For a three dimensional dataset with a boundary condition of one boundary box in each direction, the number of iterations per particle-particle interaction is computed $3^3 - 1$ times. For double precision float values, this equals $n = \lceil 26/8 \rceil = 4$ iterations with four such reductions for the potential as well as the force in three dimensions. The optimization of the reduction improves the amount of machine operations from 64 down to 22. The amount of operations to calculate the boundary condition, excluding the reduction, is 26 per iteration. In total this optimization reduces the amount of machine operations for the boundary condition calculation from 200 down to 126, an improvement of 37%.

5 Vectorization variants

All vectorization variants given in Fig. 1 are based on the source code preparation described in Section 3.2 without any debug features.

The **Auto**-vectorization variant includes no manual vectorization and is used to investigate the influence of the automatic vectorization. Without any AVX instruction set optimization, this code provides the scalar base variant. The

Label	Description
Auto	Source code variant without manual vectorization
Direct	Source code variant with a manual vectorization of the original implementation through intrinsics without further code changes
IRED	Source code variant with a manual vectorization that minimizes the amount of machine operations for each particle

Figure 1: Code variant names used in the Figures 7,8,9 and 10

manual vectorization is split into two variants. The **Direct**-vectorization variant, represents the approach to directly translate all performed computations into the corresponding intrinsics equivalent. Despite this translation no further changes to the algorithm where performed. The **IRED**-vectorization variant is an advancement of the Direct-vectorization variant as described in Section 4.

6 Benchmark platform and structure

As benchmark platform to collect measurement data about the execution time, two machines bought in 2018 with support for AVX-512 have been used. The technical specifications of these two machines are given in Fig. 2. The two machines represent different system requirements. The Intel Xeon Phi based on the Intel Many Integrated Core (MIC) Architecture provides a "high degree of parallelism in smaller, lower-power performance Intel processor cores" [14]. This system fulfills the requirement to be as fast as possible for parallel workloads. The down side of this system is that the performance for less or non-parallel tasks is low. The Intel Xeon Gold is designed as server processor to fulfill a compromise between parallel and sequential workloads. As a result, this processor provides less CPU cores, which however provide a higher base frequency of 2,10 GHz [15].

Both processors reduce their base frequency depending on the kind of workload that is being executed to remain within their terminal design power. The technical specification of the Intel Xeon Gold 6130 server processor states that the AVX-512 base core frequency is 1.3 GHz while the AVX2 base core frequency is 1.7 GHz. The Intel Xeon Phi 7250 may throttle the base core frequency by 200 MHz if an application utilizes a high amount of AVX instructions. Both systems have the Intel Turbo Mode disabled which limits the maximum frequency to the base core frequency [15, 16].

Label	Processor(s)
Server	2x Intel Xeon Gold 6130 each with 16 Cores, 32 Threads at 2.10 GHz Intel Xeon Processor Scalable Family, Intel Skylake Architecture
Phi	1x Intel Xeon Phi 7250 with 68 Cores, 272 Threads at 1.40 GHz Intel Xeon Phi processor x200 product family, Intel Many Integrated Core Architecture

Figure 2: Overview over the benchmarked machines

Every compilation was performed using the `-O3` optimization including one of the instruction set targets as given in Fig. 3. To be able to observe the changes of the Intel C Compiler multiple generations have been chosen. The three generations chosen for the observation are the ICC from 2017, 2018 and 2019. The selection starts with the ICC 2017 as this is the first generation to provide support for the `-xCORE-AVX512` as well as the `-xMIC-AVX512` instruction set selection, see [17, 18].

Different program variants have been compiled with the ICC 17, ICC 18 and ICC 19 using the code variants from Fig. 1 in combination with all instruction set parameters from Fig. 3. Those program variants have been executed on the machines from Fig. 2 with one to the maximum number of threads supported by their CPUs. All program variants have processed the same input dataset containing six particle systems, which were taken from the example `Cloud Wall` dataset provided by ScaFaCoS with 300, 2400, 8100, 19200, 102900 and 153600 particles.

The execution times are measured over the actual execution of the solver; this excludes any preparation steps as well as any post processing. Each computation was performed ten times, where the first two iterations are used as heat up phase to exclude results which are influenced by low CPU temperatures. The measured execution times are stored in a SQL database indexed by their combination of code, compiler version, instruction set and particle system. The stored execution time for each computation is based on the mean value of the remaining eight measured execution times.

Label	Compiler parameter [19]	Description
NoAVX	no Parameter	no AVX in-struc-tion set de-fined
AVX2	`CORE-AVX2`	AVX2 in-struc-tion set opti-miza-tion for any Intel pro-cessor that sup-ports Intel AVX2 in-struc-tions
BASE	`COMMON-AVX512`	base AVX-512 in-struc-tion set opti-miza-tion for any Intel pro-cessor that sup-ports Intel AVX-512 in-struc-tions
ARCH		architecture spe-cific

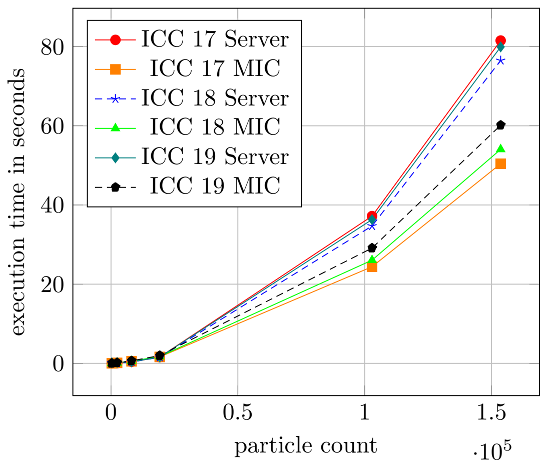

Figure 4: Comparison of the Auto-vectorization using the Architecture specific optimization between the Intel C Compiler 2017, 2018 and 2019 on the Server processor system and the MIC processor system

7 Benchmark evaluation regarding the automatic vectorization

To allow a clear overview for the comparison between the automatic vectorization and the manual vectorizations, it is required to reduce the data to be presented. As the center of all comparison is the automatic vectorization, this section targets to evaluate the system parameter that achieve the best execution time. All tests were performed using three different compilers; in Subsection 7.1 one compiler is then chosen to be used in the remainder of this article. To investigate the scalability of the automatic vectorization, Subsection 7.2 evaluates the amount of threads for each system leading to the best execution time. To verify that AVX-512 is the best instruction set for the automatic vectorization of the direct solver, Subsection 7.3 provides an overview that AVX-512 achieves overall better execution times then AVX2.

7.1 Comparison of different versions of the Intel C Compiler

To determine the best compiler version, the execution times of the program versions created using the architecture specific instruction set executed with the highest amount of threads where used. By comparing the execution times in Fig. 4, a difference between the compilers on each platform becomes visible.

For the Xeon server processors, the ICC 18 achieved the best execution time with the ICC 17 providing the worst execution time. For the Xeon Phi processor, the ICC 17 provided the best execution time, whereas the ICC 18 achieves the second best execution time. Comparing the average difference between all execution times, the ICC 18 provided the smallest overall execution

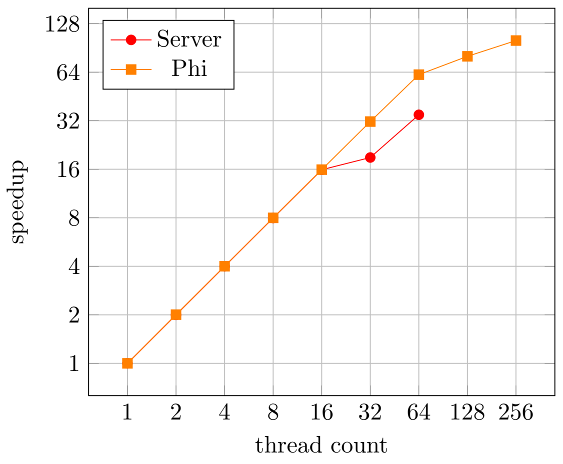

Figure 5: Speedup for varying thread counts based on the dataset with 153600 particles simulated by the Auto vectorization variant with the Architecture specific optimization

time, while the ICC 19 provided the largest execution time. The most likely reason, why the ICC 19 is the slowest in our test, is the introduction [20] of new mitigations to speculative execution side-channel issues. Based on those findings, the **Intel C Compiler 2018** (ICC 18) was chosen as it provided the best overall execution times in all tests.

7.2 Execution times in relation to the number of threads

To determine the efficiency of the OpenMP parallelization in conjunction with the automatic vectorization, the computations have been performed using different numbers of thread. The program version compiled by the ICC 18 using the architecture specific instruction set optimization was used. The largest dataset with 153600 particles was chosen as those execution times were the least fluctuating. The thread count was increased by the power of two, starting with one up to the maximum number of threads supported by the machine.

As seen in Fig. 5, the speedup for both machines is almost linear. For the Xeon server processors, the efficiency of 0.99 is only inhibited by the change from 16 threads to 32 threads. The reason is that the operating system schedules the additional 16 tasks to the hyperthreads provided by the first CPU instead of using the second CPU. However, for 64 threads the operating system utilized both CPUs, which results in a speedup similar to the first 16 threads. Based on the result for the highest thread count the efficiency is 0.53. The same result related to the core count achieves an efficiency of 1.06. For the Intel Xeon Phi, the speedup is also linear with an efficiency of 0.95 until all cores are occupied with one thread. Starting with 68 threads the efficiency is reduced for every additional thread. However, despite the decreased efficiency, the speedup is still increased. Based on the result for the highest thread count the efficiency is 0.38. The same result related to the core count achieves an efficiency of 1.54. The

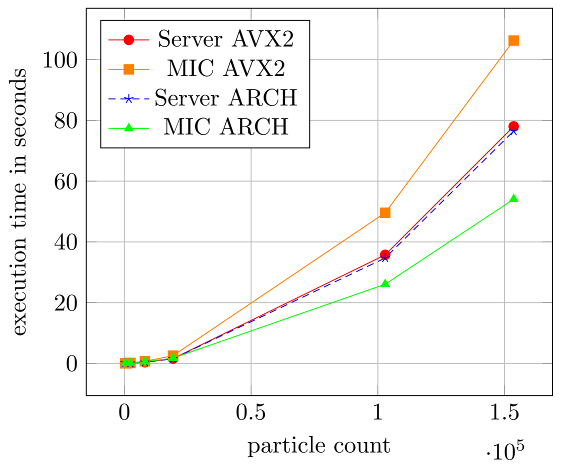

Figure 6: Comparison of the Auto-vectorization using AVX2 against the Auto-vectorization using AVX-512 with architecture specific optimization on the Server and MIC processor system

results show that the implemented parallelization and vectorization scales well with the numbers of cores. As the efficiency for the scaling based on the core count is very high with 0.95 to 0.99, the gains by the usage of the hardware threads is minimal. However, as the efficiency based on the core count is larger than 1 when all hardware threads are used, all subsequent comparisons will be performed using all those threads.

7.3 Comparison of the automatic vectorization utilizing AVX2 against the automatic vectorization utilizing AVX-512

As mentioned in the related work Section 2, there are studies for other use cases that have shown that a vectorization utilizing AVX2 might be faster than one utilizing AVX-512. To evaluate if those findings also apply to the use case of this paper, the **Auto**-vectorization was compiled with AVX2 as target instruction set for the automatic vectorization. The execution times of the automatic vectorization using AVX2 as well as AVX-512 can be seen in Fig. 6. A comparison of both variants shows that for the Intel Xeon Server there is just a minor difference between AVX2 and AVX-512 with 78 in contrast to 76 seconds. As AVX-512 with 512-bit vectors is able to process twice as much data at the same time as AVX2 with 256-bit vectors, the expected difference would be larger. The measurements show that the processor adjusts its core frequency based on the function units used as explained in Section 6. For the Intel Xeon Phi, the advantage of AVX-512 with its doubled vector size results in almost twice as much speed with an execution time of 54 seconds compared to 106 seconds for AVX2. This outcome confirms the expectation in Section 6 that the Intel Xeon Phi benefits more from AVX-512 then the Intel Xeon Server. For the Coulomb

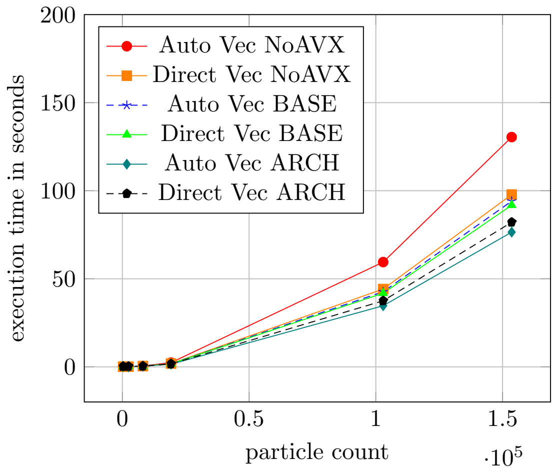

Figure 7: ICC18 Comparison of the Auto-vectorization using the optimization levels NoAVX, BASE and ARCH-iecture specific to the addition of the Direct-vectorization on the Server processor system

solver, the AVX-512-vectorizations achieved an overall better execution time compared to the AVX2-vectorization. Therefore, all AVX2-vectorizations are omitted in the subsequent evaluations.

8 Benchmark evaluation regarding the manual vectorization

This Section compares the execution times of all variants that purely rely on automatic vectorization against the manually vectorized SIMD variants. In this comparison, only variants built with the same compiler flags are directly comparable, as the effects of the AVX instruction set optimization influence the execution time beyond the automatic vectorization.

8.1 Comparison of the Auto-vectorization against the Direct-vectorization

Starting with the Intel Xeon Gold 6130 platform, Fig. 7 shows how the Direct-vectorization variant with no additional vectorization optimizations outperforms the scalar execution. The execution time is reduced from 131 seconds to 98 seconds, which correspond to a speedup of 1.33. By activating the BASE AVX optimizations, the execution time of the Auto-vectorization went down to 94 seconds while the execution time for Direct-vectorization is 91 seconds. For the BASE AVX optimization the speedup achieved through manual vectorization is 1.03. By enabling the architecture specific optimization ARCH, the execution time of the Auto-vectorization went down to 77 seconds, while the Direct-vectorization needs 82 seconds. This results in a disadvantage for the

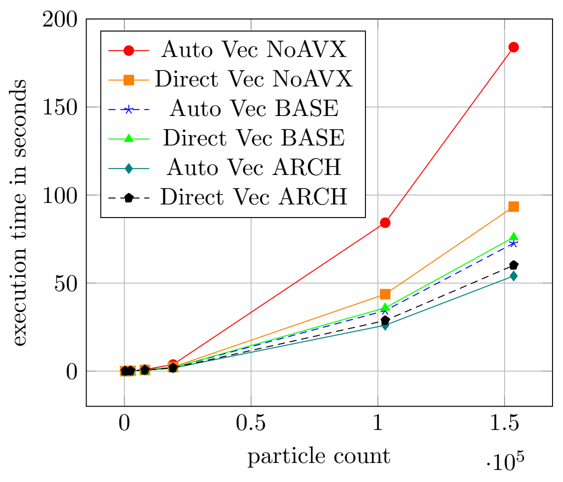

Figure 8: ICC18 Comparison of the Auto-vectorization using the optimization levels NoAVX, BASE and ARCH-iecture specific to the addition of the Direct-vectorization on the MIC processor system

usage of the Direct-vectorization, which is almost 7% slower. For the Intel Xeon Phi 7250, the result is similar to the Xeon Gold. As seen in Fig. 8 the advantage of the Auto-vectorization over the Direct-vectorization is already noticeable at the BASE AVX optimization, with 73 seconds compared to 76 seconds.

The reason for the reduced performance of the Direct variant can be seen by comparing the assembler code of the architecture specific optimized Auto-vectorization to the Direct-vectorization. Both variants result in a similar vectorization but the pure automatic vectorization performs the calculations in a slightly different order. Due to the changed order, the register usage for the vectorization is improved which results in less memory interactions.

8.2 Comparison of the Auto-vectorization against the IRED-vectorization with reduction optimization

The comparisons in the previous subsection have shown that for our use case the pure automatic vectorization can outperform a SIMD implementation. However, as described in Section 4 there is an additional optimization potential to the reduction operations.

As seen in Fig. 9, even without any further AVX-512 instruction set optimizations, the **IRED**-vectorization is clearly ahead of all variants exclusively relying on the automatic vectorization. Comparing the variants with the architecture specific AVX-512 instruction set optimizations, for the highest amount of particles the execution time is sped up by 1.67, this is a reduction from 77 seconds to 46 seconds. The percentage of this time saving corresponds to the calculated amount of reduced operations.

Fig. 10 shows a similar improvement on the Intel Xeon Phi. The execution time for the highest amount of particles is reduced from 54 seconds down to

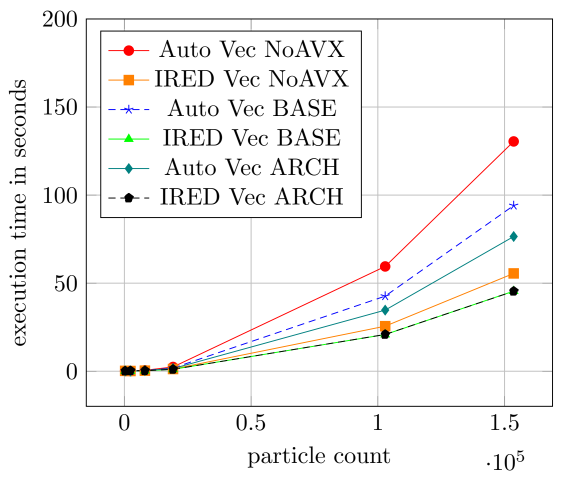

Figure 9: ICC18 Comparison of the Auto-vectorization using the optimization levels NoAVX, BASE and ARCH-iecture specific to the addition of the IRED-vectorization on the Server processor system

39 seconds, which correspond to a speedup of 1.38. The reduced speedup over the Intel Xeon Gold can be explained by unvectorizable parts of the solver that have a larger influence on the Intel Xeon Phi processor due to the lower core frequency.

The comparison of the scalar execution time on the Intel Xeon Gold with 130 seconds against its best execution time with 46 seconds shows a significant speedup of 2.82. The same comparison for the Intel Xeon Phi processor shows a difference from 183 seconds to 39 seconds, which is a speedup of 4.69.

9 conclusion

In this article, we have investigated whether a manual vectorization is still required to achieve a good performance or if an automatic vectorization can achieve the same performance as a manual one. For the examination, a real world application for particle interactions with a Coulomb solver has been chosen and manually vectorized utilizing the Advanced Vector Extensions with 512-bit vector size (AVX-512). The SIMD code has been benchmarked against an automatic vectorization utilizing AVX-512 as target instruction set.

The insight gained from this investigation is that if the source code of an application can be vectorized then the automatic vectorization using the suitable compiler options delivers good results. In the case of the parallelized Coulomb solver, the automatic vectorization achieved a speedup of 1.67 when executed with 64 threads on the Intel Xeon Server system and a speedup of 3.4 using 272 threads on an Intel Xeon Phi system. However, because of the limitations of the compiler heuristics, the manual vectorization can outperform the automatic vectorization. By reorganizing the solver algorithm, a minimization of the amount of vector machine operations was possible. Compared to the par-

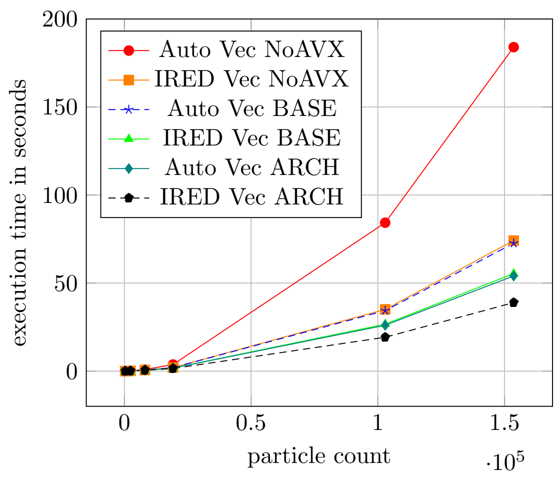

Figure 10: ICC18 Comparison of the Auto-vectorization using the optimization levels NoAVX, BASE and ARCH-iecture specific to the addition of the IRED-vectorization on the MIC processor system

allel non-vectorized execution, this improvement leads to a speedup of 2.88 for the Intel Xeon Server system using 64 threads and 4.71 for the Intel Xeon Phi system using 272 threads.

10 Acknowledgment

This work has been supported by the Federal Cluster of Excellence EXC 1075 *MERGE Technologies for Multifunctional Lightweight Structures* financed by the German Research Foundation (DFG) as well as the German Ministry of Science and Education (BMBF) project *SeASiTe Self-adaption of time-step-based simulation techniques on heterogeneous HPC systems*, Grant No. 01IH16012B.

References

[1] M. Bolten, F. Fahrenberger, R. Halver, F. Heber, M. Hofmann, I. Kabadshow, O. Lenz, M. Pippig, and G. Sutmann, "ScaFaCoS, C subroutine library." [Online]. Available: http://scafacos.github.com

[2] D. Kusswurm, *Modern X86 Assembly Language Programming Covers x86 64-bit, AVX, AVX2, and AVX-512*, 2nd ed., 2018.

[3] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual.*, April 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

[4] J. M. Cebrian, L. Natvig, and M. Jahre, "Scalability analysis of avx-512 extensions," *The Journal of Supercomputing*, 2019.

[5] B. Bramas and P. Kus, "Computing the sparse matrix vector product using block-based kernels without zero padding on processors with avx-512 instructions," *PeerJ Computer Science*, vol. 4, 2018.

[6] B. Bramas, "A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 10, 2017.

[7] M. A. A. Farhan and D. E. Keyes, "Optimizations of unstructured aerodynamics computations for many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2317 – 2332, 2018.

[8] I. Kanamori and H. Matsufuru, "Practical implementation of lattice qcd simulation on simd machines with intel avx-512," in *Computational Science and Its Applications – ICCSA 2018.* Cham: Springer International Publishing, 2018, pp. 456–471.

[9] B. Ginting and R.-P. Mundani, "Comparison of shallow water solvers: Applications for dam-break and tsunami cases with reordering strategy for efficient vectorization on modern hardware," *Water*, vol. 11, no. 4, 2019.

[10] E. Rucci, C. G. Sanchez, G. B. Juan, A. D. Giusti, M. Naiouf, and M. Prieto-Matias, "Swimm 2.0: Enhanced smith–waterman on intel's multicore and manycore architectures based on avx-512 vector extensions," *International Journal of Parallel Programming*, vol. 47, no. 2, pp. 296 – 316, 2019.

[11] A. Arnold, F. Fahrenberger, C. Holm, O. Lenz, M. Bolten, H. Dachsel, R. Halver, I. Kabadshow, F. Gähler, F. Heber, J. Iseringhausen, M. Hofmann, M. Pippig, D. Potts, and G. Sutmann, "Comparison of scalable fast methods for long-range interactions," *Phys. Rev. E*, vol. 88, p. 063308, Dec 2013.

[12] M. Hofmann, F. Nestler, and M. Pippig, "NFFT based Ewald summation for electrostatic systems with charges and dipoles," *Appl. Numer. Math.*, vol. 122, pp. 39–65, 2017.

[13] M. Bolten, F. Fahrenberger, R. Halver, F. Heber, M. Hofmann, I. Kabadshow, O. Lenz, M. Pippig, and M. Winkel, *ScaFaCoS Manual.* [Online]. Available: http://www.scafacos.de/documentation.html

[14] Intel, "Intel many integrated core architecture." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

[15] Intel, "Intel xeon processor scalable family specification update," June 2019. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf

[16] Intel, "Intel xeon phi processor product brief," 2016. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-processor-product-brief.pdf

[17] Intel, "Intel c++ compiler 15.0 release notes," Apr 2015. [Online]. Available: https://software.intel.com/sites/default/files/managed/1e/6a/ReleaseNotes_ISS_Compiler.pdf

[18] Intel, "Intel c++ compiler 17.0 release notes," Sep 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-c-compiler-170-for-windows-release-notes-for-intel-parallel-studio-xe-2017

[19] Intel, "Intel compiler options for intel sse and intel avx generation and processor-specific optimizations," Oct 2017. [Online]. Available: https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations

[20] Intel, "Intel c++ compiler 19.0 for windows* release notes for intel parallel studio xe 2019," Apr 2018. [Online]. Available: https://software.intel.com/en-us/articles/intel-c-compiler-190-for-windows-release-notes-for-intel-parallel-studio-xe-2019