Article

Acceleration of Particle Swarm Optimization with AVX Instructions

Jakub Safarik [1,*] and Vaclav Snasel [2]

[1] Laboratory of Big Data Analysis, IT4Innovations, VSB—Technical University of Ostrava, 70800 Ostrava, Czech Republic
[2] Faculty of Electrical Engineering and Computer Science, VSB—Technical University of Ostrava, 70800 Ostrava, Czech Republic
* Correspondence: jakub.safarik@vsb.cz

Abstract: Parallel implementations of algorithms are usually compared with single-core CPU performance. The advantage of multicore vector processors decreases the performance gap between GPU and CPU computation, as shown in many recent pieces of research. With the AVX-512 instruction set, there will be another performance boost for CPU computations. The availability of parallel code running on CPUs made them much easier and more accessible than GPUs. This article compares the performances of parallel implementations of the particle swarm optimization algorithm. The code was written in C++, and we used various techniques to obtain parallel execution through Advanced Vector Extensions. We present the performance on various benchmark functions and different problem configurations. The article describes and compares the performance boost gained from parallel execution on CPU, along with advantages and disadvantages of parallelization techniques.

Keywords: AVX; optimization; PSO; vector instructions

check for updates

Citation: Safarik, J.; Snasel, V. Acceleration of Particle Swarm Optimization with AVX Instructions. *Appl. Sci.* **2023**, *13*, 734. https://doi.org/10.3390/app13020734

Academic Editors: Giancarlo Mauri and Aleksander Mendyk

Received: 7 November 2022
Revised: 18 December 2022
Accepted: 30 December 2022
Published: 4 January 2023

1. Introduction

Intel's AVX (Advanced Vector Extension) utilizes parallel processing of single instructions on several data streams. It was developed to simplify and increase the efficiency of algorithms such as matrix multiplication. This allows the CPU architecture to adapt to more complex problems where the classical approach becomes slow.

A fair comparison of code performance is hard for the same programs, especially for different architectures or compilers. Generally, the only objective metric available is the execution time, and most authors have adopted this approach [1–5]. The speedup in execution time is also one of the main reasons for developing a parallel version of an algorithm.

Most papers focused on parallelization speedup compare highly parallel GPU versions of algorithms with single-core scalar CPU versions. However, this may lead to a misleading interpretation of the GPU vastly outperforming the CPU version. However, the modern CPU multicore architecture with SIMD (single instruction multiple data) instructions reduces the performance gap between GPUs and CPUs.

In recent years, SIMD has been adopted in various fields—image processing, signal processing, cryptography, machine learning, etc. The impact of AVX on 3D collision modeling was examined by the authors of [6]. The use of AVX in data science, computational statistics and high-performance computation was studied by the authors of [7,8]. They investigated various vectorization options and available compilers to investigate the impacts on the performances of different algorithms. The authors of [9] explored the optimization of matrix multiplication. The other general concept is sorting, a fast vectorized implementation presented in the paper [10].

The vector instruction helps to optimize cryptography too. The authors of [11] used AVX to optimize shuffling in the diffusion layer used in block ciphers. AVX-512 was deployed in the SIKE cryptosystem to minimize the latency in the key encapsulation mechanism, arithmetics and isogeny computation.

Related research aims at improving the performance of machine learning. AVX-512 was deployed to enhance deep learning in [12]. The study compares the performance of learning using a GPU and a CPU with AVX. The authors of [13] experimented with AVX for genetic programming, especially with generations of binary trees. AVX allowed them to execute faster and far longer than previous attempts. Another study aimed at improving graph partitioning problems [14].

This paper compares computing performances obtained by CPU-based parallelized versions of PSO. We have a scalar version of the PSO algorithm, allowing us to run the program with virtually any size of the swarm. We developed parallel versions of this scalar PSO with an internal structure optimized for SIMD instructions. As final implementations share similar limitations, we could compare the actual performances with different designs.

Our paper presents the novelty of comparing various options of vectorized PSO algorithms. The motivation was to compare the variants for the specific task of PSO. The results presented in this paper could guide other authors and ease the task of selecting the proper vectorization method for faster development and more efficient code.

We obtained all results on a suite of standard benchmark functions used for single-objective stochastic optimization algorithms. The paper is organized as follows: Section 2 provides relevant background on the PSO algorithm and its parallel implementation with SIMD instructions. We present our experiment design, benchmark test functions and evaluation of results in Section 3. In the last Section 4, we offer conclusions and future outlines.

2. Particle Swarm Optimization

The algorithm utilizes a swarm theory inspired by observing animals and their social behavior (e.g., herds, fish schools, and bird flocks). The algorithm has a population of entities called particles serving as candidate solutions. Those particles are randomly initialized in the search space of the problem.

Every particle starts with a random velocity vector that defines its movement. The particle adjusts this velocity vector with regard to its own experience and the best solution (position of a best-performing particle) in the current swarm. This way, the particle explores the search space, and the whole swarm should converge to the optimum solution.

The problem's solution space is a J dimensional area with a population of I particles. Each particle i has a position $X_i = x_{i1}, x_{i2}, \ldots, x_{ij}$ and a velocity vector $V_i = v_{i1}, v_{i2}, \ldots, v_{ij}$. The particle updates the position during each iteration and moves towards a personal best-known position $L_i = l_{i1}, l_{i2}, \ldots, l_{ij}$. The second attractor is the best position known so far $G_{best} = g_1, g_2, \ldots, g_j$. The velocity update is then a combination of the previous velocity and the aforementioned attractors:

$$v_{ij}(t+1) = w * v_{ij}(t) + c_1 r_1(t)[l_{ij}(t) - x_{ij}(t)] + c_2 r_2(t)[g_j(t) - x_{ij}(t)], \quad (1)$$

where $v_{ij}(t)$ is the velocity of particle i in dimension j at time t. The inertial weight w influences the velocity from the previous iteration to control the exploration and exploitation of search space. Small w urges the particle to exploit the area in its near position, and a high value prevents getting stuck in local minima [15].

Values r_1 and r_2 are independent random numbers (having uniform distribution from 0 to 1). c_1 and c_2 constants control the acceleration to attractors. c_1 influences the cognitive part of the particle (moving towards the historically best position), representing primitive thinking. The constant c_2 takes part in social behavior and represents the cooperation among particles.

Various recommendations of c_1 and c_2 values exist. Some publications rely on using complementary values less than or equal to 1 [4,16]. Engelbrecht and others [15,17] confirmed that having high inertial weight in compare to c_1 and c_2 (see Equation (2)) leads particles to divergent or cyclic trajectories:

$$w > \frac{1}{2}(c_1 + c_2) - 1. \tag{2}$$

Similar results were found by other researchers in [18].

The updated position of the particle is then computed simply as:

$$X_i(t+1) = X_i(t) + V_i(t+1), \tag{3}$$

where $X_i(t)$ is the actual position of particle in search space and $V_i(t+1)$ is the new velocity vector.

The basic PSO terminate safter reaching a specific number of iterations when the value of G_{best} reaches a designated goal. One iteration (or generation) means the updating of particle position, evaluation of the fitness function, and updating of internal attractors (local l_{ij} and global G_{best} best) in the whole swarm. Table 1 summarizes PSO control parameters. The choice of values is similar to that of Engelbrecht [15,19,20].

Table 1. PSO parameters during experiments.

Param	**Value**
PSO iterations	1000
Particles in swarm	32, 64, 128, 256, 512
Number of dimensions	8, 16, 32, 64, 128, 256, 512
Inertia weight	0.729
Max. velocity	by problem's range
Local weight (c1)	1.49445
Global weight (c2)	1.49445

Parallelization of PSO

A real-world optimization requires much computational effort typically. In the case of the PSO algorithm, there could be thousands or even millions of dimensions. Parallelization of these high-dimensional problems provides a needed performance boost. The PSO is among the most efficient stochastic search algorithms. It is intrinsically parallel and best suited for parallelization, being better than other evolutionary or swarm intelligence algorithms [5].

Improvements in vector processors allow efficient parallelization even without a GPU. The vectorization of PSO using AVX2 instructions is natural because all particles act in parallel implicitly [21].

The theoretical performance peak of a GPU is a hundred times higher than the performance of a single CPU. However, a GPU shows only about a 30$\times$ speedup for PSO and similar problems [22]. Various research confirms the potential of multicore SIMD execution, providing a similar performance boost [23,24].

Vectorized PSO requires sharing of a few values: an array with G_{best} position and corresponding fitness evaluation for this position [15,25]. All other values and particles are mutually independent and easily parallelizable.

Most compilers have the ability of automatic vectorization. If they detect potentially vectorized code, they will try to replace it with a vectorized counterpart. However, the compiler usually fails to vectorize a part code for various reasons, as confirmed by our test of auto-vectorization of C++ PSO code. There are several ways how to achieve vectorized C++ code:

- Compiler's auto-vectorization (scalar);
- Vectorized classes (avavx);
- AVX intrinsics (intavx);
- ASM code with AVX instructions (asmavx).

The methods are ordered from the most programmer-friendly to the sophisticated approach [26–28]. Writing highly vectorized and fast code decreases readability and maintainability. The most convenient way is using the auto-vectorization capability of the compiler. Even the auto-vectorization of simple loops has some hidden requisites, which are specific to parallel programming [22]. One of the most efficient ways to achieve vectorized code is using already vectorized libraries, vector classes and SIMD intrinsics.

3. Empirical Analysis Additionally, Experimental Procedure

This part aims to provide an empirical comparison of PSO implementation with AVX instructions. We tested the performance on a set of benchmark functions described below closely.

We compared the algorithm's performance on four C++ implementations, which differ in just one detail. The scalar implementation uses a straightforward approach without any vector instructions. This implementation provides just baseline results for further experiments. Parallel variants use Advanced Vector Extensions (AVX2) to compute particle movement updates. This crucial feature is achieved with auto-vectorization, SIMD intrinsics and raw assembly instructions.

The optimized code affects only the PSO algorithm itself (especially the velocity and position update), not the computation of the fitness function. Our goal was to monitor the change in algorithm performance, not the optimization of test functions.

We executed the PSO algorithm 200 times for each benchmark function to obtain independent results. All algorithms used the same PSO settings and synchronous updates. The exact values of control parameters are contained in Table 1.

We chose to follow the approach from Engelbrecht [19] and used functions from the CEC2005 test suite, in combination with other standard benchmark functions. Table 2 lists the benchmark functions with their range constraints.

We tested a few implementations of PSO, all implemented in C++. Each PSO run used specific settings of dimensions and number of particles for each experimental run. All PSO variants were optimized to achieve maximum performance and compiled with optimizations for speed.

The first implementation used a scalar approach without any parallelization and served as an AVX-independent measure of performance. The second approach (Avavx) used optimized C++ code with enabled auto-vectorization capabilities of the compiler. Another variant (Intavx) relied on AVX intrinsics, which are C-style functions providing access to AVX instructions without the need to write assembly code. The last implementation (Asmavx) was C++ code combined with raw assembly code. For all implementations using specific AVX instructions, we optimized only the same crucial part of PSO computation.

All benchmarks were performed on an Intel Core i7-4770K (4 physical, 8 logical cores) processor with 16GB RAM, running Windows 10. The runtime performance on the complete test suite is provided in Table 3. The table shows the cumulative runtimes of various versions sharing the same configuration of 32 particles but having varying dimensionality for the test functions.

Table 3 shows the performance boost provided by SIMD. All AVX versions easily outperformed the scalar implementation. We can see the performance change in low dimensions, where the auto-vectorized version struggled more than the intrinsic or assembly version of PSO. This could be caused by inefficient data fetching and cache missing. After processing larger chunks of data, the Avavx's version performance increases.

The detailed performance comparison of various configurations is displayed Figures 1–3. As all parallel implementations outperformed the scalar version, we omitted details of its results.

In the first experiment, we compared the runtimes of Intavx and Avavx code with the various configurations on the whole test suite. This experiment showed only slight differences between implementations. Figure 1 illustrates the PSO runs (with appropriate configurations) used on the whole test suite (cumulative sum of each test function runtimes). The differences between code variants were small, but the Intavx code slightly outperformed Avavx, and the average reduction in total runtime was 3.13%.

Table 2. Benchmark functions.

	Benchmark Function	**Range**
f01	Absolute	$[-100, 100]$
f02	Ackley	$[-32.77, 32.77]$
f03	Alpine	$[0, 20]$
f04	Egg holder	$[-512, 512]$
f05	Elliptic	$[-100, 100]$
f06	Griewank	$[-600, 600]$
f07	Hyperellipsoid	$[-5.12, 5.12]$
f08	Michalewicz	$[0, \pi]$
f09	Norwegian	$[-1.1, 1.1]$
f10	Quadratic	$[-100, 100]$
f11	Quadratic	$[-1.28, 1.28]$
f12	Rastrigrin	$[-5.12, 5.12]$
f13	Rosenbrock	$[-30, 30]$
f14	Salomon	$[-100, 100]$
f15	Schaffer 6	$[-100, 100]$
f16	Schwefel 1.2	$[-100, 100]$
f17	Schwefel 2.6	$[-100, 100]$
f18	Schwefel 2.13	$[-\pi, \pi]$
f19	Schwefel 2.21	$[-100, 100]$
f20	Schwefel 2.22	$[-10, 10]$
f21	Shubert	$[-10, 10]$
f22	Spherical	$[-5.12, 5.12]$
f23	Step	$[-100, 100]$
f24	Vincent	$[0.25, 10]$
f25	Weierstrass	$[-0.5, 0.5]$

Table 3. Runtime performances of various PSO versions (32 particles, complete test suite).

Dimension	**Scalar [s]**	**Avavx [s]**	**Intavx [s]**	**Asmavx [s]**
8	4.32×10^{-1}	3.54×10^{-1}	2.25×10^{-1}	2.32×10^{-1}
16	7.32×10^{-1}	3.14×10^{-1}	2.85×10^{-1}	2.63×10^{-1}
32	1.20×10^{0}	5.552×10^{-1}	5.32×10^{-1}	5.23×10^{-1}
64	2.42×10^{0}	1.05×10^{0}	1.01×10^{0}	9.76×10^{-1}
128	4.92×10^{0}	2.06×10^{0}	1.99×10^{0}	1.87×10^{0}
256	9.92×10^{0}	4.10×10^{0}	3.96×10^{0}	3.84×10^{0}
512	1.99×10^{1}	8.12×10^{0}	7.78×10^{0}	7.55×10^{0}

The graph shows a linear increase in runtime as the dimensionality and particle number grow. The runtime measures the time from the initialization of the PSO till the last iteration. When the dimensionality doubles, the algorithm needs to compute twice the updates, and vice versa. The same condition is true for the number of particles.

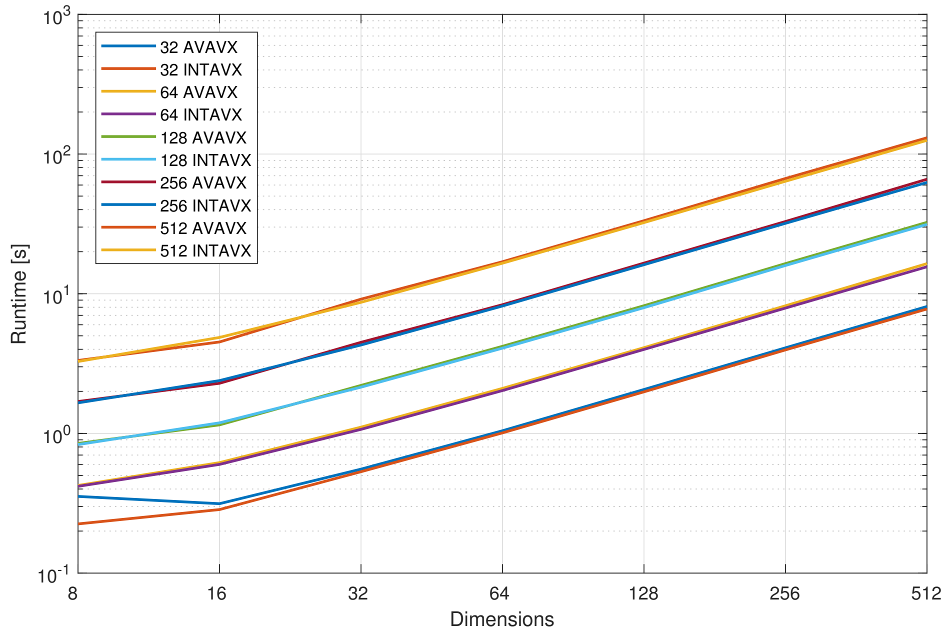

Figure 1. Runtime performance on benchmark suite—various configurations of particles, and dimensions.

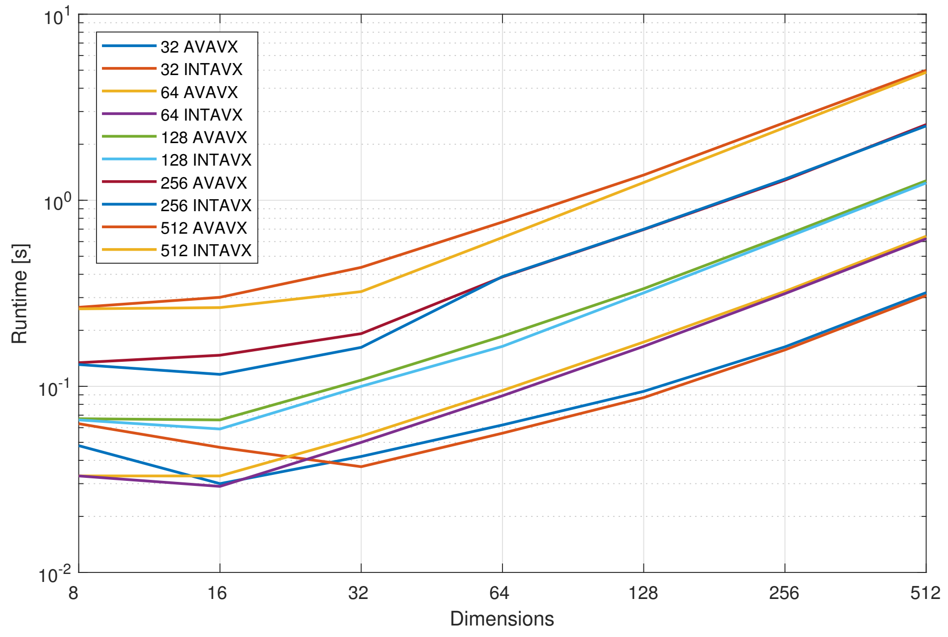

Figure 2. Runtime performance of PSO (Vincent function)—various configurations.

However, if we look closely at individual performance for each test function, we can see a significant difference in performance, especially for the lower numbers of dimensions and particles. As the numbers grow, the difference in runtime decreases. The architecture of AVX instruction could cause this—before execution of AVX instruction, the processor needs additional time. Fast switching between AVX instruction and scalar instruction causes another overhead.

Table 4 summarizes average runtime for each test function. In this experiment, we performed 100 PSO runs with the same configuration of 128 dimensions and 32 particles. The remaining parameters used the values mentioned in Table 1.

The results in Table 4 show the highest similarity in performance of both implementations for test function number 24—the Vincent function. Figure 2 illustrates the runtime performances of various PSO configurations on the Vincent function. The graph shows similar performance in multiple settings.

We selected the Vincent function with the highest similarity to mitigate the impacts of other factors. The final performance depends on memory allocation, data prefetching, loop unrolling, blocking, etc. Using the highest similarity, we could see the lowest expected increase in performance.

The Intavx code with AVX intrinsics generally outperformed auto-vectorized code in most PSO problem configurations. The average performance boost for Intavx was 6.48%, and the highest variability of runtimes occurred for a low number of dimensions and particles.

We conducted the last series of experiments with the Asmavx version. It improved in performance by an average of 3.66% against Intavx, especially for the higher numbers of particles and dimensions. Intavx achieved the same or even better runtimes in lower dimensions and for smaller swarms. Results are summarized in Figure 3, where we omit Avavx results for clarity.

Table 4. PSO runtime performance on the test suite: 128 dimensions, 32 particles.

	Avavx		**Intavx**		**Difference**	
Fn	**Average [s]**	**St. Deviation**	**Average [s]**	**St. Deviation**	**Absolute**	**Relative**
1	7.79×10^{-2}	9.02×10^{-4}	8.34×10^{-2}	1.89×10^{-3}	5.46×10^{-3}	6.76%
2	9.93×10^{-2}	4.65×10^{-3}	1.14×10^{-1}	1.39×10^{-3}	1.45×10^{-2}	13.64%
3	1.11×10^{-1}	5.44×10^{-3}	1.04×10^{-1}	1.49×10^{-3}	6.78×10^{-3}	6.32%
4	1.75×10^{-1}	3.42×10^{-3}	1.97×10^{-1}	2.30×10^{-3}	2.17×10^{-2}	11.66%
5	1.20×10^{-1}	1.95×10^{-3}	1.23×10^{-1}	3.08×10^{-3}	2.91×10^{-3}	2.40%
6	1.51×10^{-1}	1.32×10^{-3}	1.39×10^{-1}	2.75×10^{-3}	1.21×10^{-2}	8.34%
7	8.86×10^{-2}	8.64×10^{-4}	9.59×10^{-2}	1.65×10^{-3}	7.28×10^{-3}	7.90%
8	1.03×10^{-1}	9.35×10^{-3}	1.12×10^{-1}	5.17×10^{-3}	8.52×10^{-3}	7.93%
9	7.65×10^{-2}	1.06×10^{-3}	7.55×10^{-2}	2.24×10^{-3}	9.90×10^{-4}	1.31%
10	7.94×10^{-2}	2.28×10^{-3}	8.48×10^{-2}	5.02×10^{-3}	5.43×10^{-3}	6.61%
11	1.13×10^{-1}	2.32×10^{-3}	1.94×10^{-1}	5.90×10^{-3}	8.06×10^{-2}	52.51%
12	8.81×10^{-2}	5.10×10^{-3}	8.67×10^{-2}	1.75×10^{-3}	1.45×10^{-3}	1.66%
13	7.72×10^{-2}	4.77×10^{-4}	7.89×10^{-2}	5.94×10^{-4}	1.66×10^{-3}	2.13%
14	8.13×10^{-2}	5.63×10^{-4}	8.42×10^{-2}	8.00×10^{-4}	2.99×10^{-3}	3.61%
15	1.27×10^{-1}	1.59×10^{-3}	1.45×10^{-1}	1.20×10^{-3}	1.79×10^{-2}	13.14%
16	7.78×10^{-2}	7.66×10^{-4}	7.92×10^{-2}	5.47×10^{-4}	1.36×10^{-3}	1.73%
19	7.53×10^{-2}	1.02×10^{-3}	7.81×10^{-2}	1.32×10^{-3}	2.86×10^{-3}	3.72%
20	8.71×10^{-2}	1.72×10^{-3}	8.75×10^{-2}	8.11×10^{-4}	3.60×10^{-4}	0.41%
21	1.17×10^{-1}	4.47×10^{-3}	2.99×10^{-1}	3.14×10^{-3}	1.82×10^{-1}	87.78%
22	8.74×10^{-2}	9.44×10^{-4}	1.25×10^{-1}	2.15×10^{-3}	3.75×10^{-2}	35.34%
23	2.26×10^{-1}	2.22×10^{-3}	9.71×10^{-2}	1.40×10^{-3}	1.29×10^{-1}	79.81%
24	9.12×10^{-2}	3.85×10^{-3}	9.13×10^{-2}	1.48×10^{-3}	1.40×10^{-4}	0.15%
25	7.66×10^{-2}	7.13×10^{-4}	7.48×10^{-2}	1.15×10^{-3}	1.83×10^{-3}	2.42%

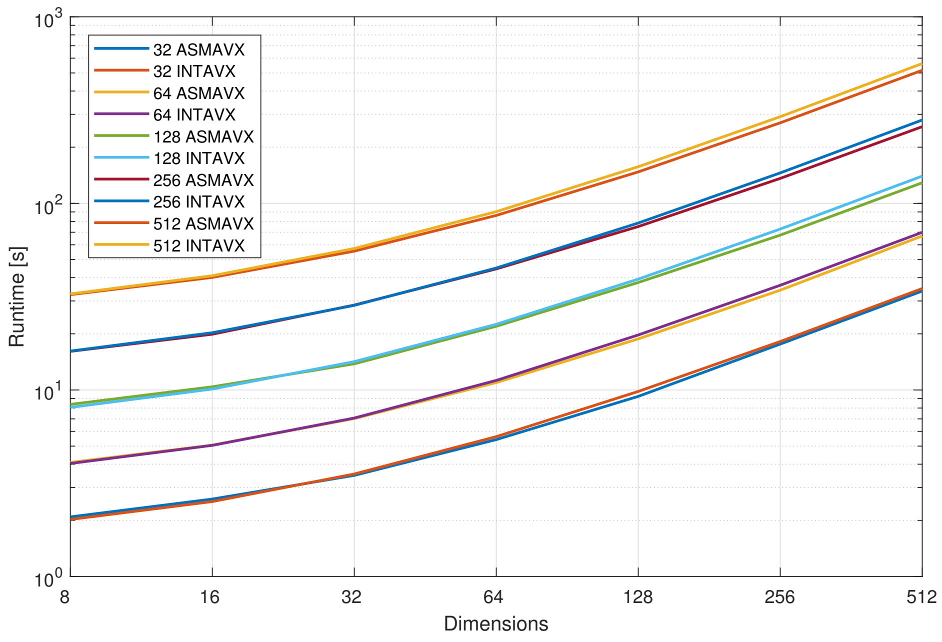

Figure 3. Runtime performance of PSO, benchmark suite—Asmavx vs. Intavx.

4. Conclusions

We tested the performance of a parallel CPU-based implementation of the PSO algorithm. Our goal was to compare the impact of AVX instructions on the execution speed.

We assessed the performance of four versions of PSO implemented in C++ programming language. All versions were optimized for the target architecture and used the same code, except for the crucial part of the PSO computation.

The results confirm that the best-performing version uses assembly code . Similar performance was achieved by the more independent code programmed with AVX intrinsics, which resulted in only a slight performance decrease compared to the assembly code. The code optimized just by the auto-vectorization capability of the compiler suffers from another reduction in performance but still vastly outperforms the scalar version of PSO.

In other words, automatic compiler vectorization creates fast parallel code. This approach combines the benefits of rapid development without using any AVX-specific instructions. There are still some limitations and specific requirements on the source code affecting vectorization by the compiler.

The most suitable approach for increasing code performance involves AVX intrinsics. The final assembly code was almost identical to the Asmavx version and provided better performance than the Avavx version.

Asmavx code benefits from the problems usually caused by high dimensions and big swarms—providing further performance gains because of its low-level nature and other assembly tweaks. On the other hand, there are higher requirements for programmers, maintenance, portability and development time than for the previous versions.

We generally achieved about two times faster code using vectorization instead of the simple scalar version. The overall performance gain affects several factors. Not all code can be parallel, and we vectorized only the crucial part of PSO computation. The PSO algorithm's performance depends significantly on the test function used. As the test function varies or is unknown for real-world problems, we oriented the optimizations only toward the core of the PSO algorithm.

The results suggest that the use of auto-vectorization brings significant improvement in performance without further requirements on developers. The code is competitive with intrinsic and assembler versions. The performance gain became helpful for various real-world applications and should be used as default settings. However, to fully utilize the

AVX performance in complex computations, optimization by using intrinsic or assembler instructions is still required.

Author Contributions: Conceptualization, J.S. and V.S.; methodology, J.S. and V.S.; software, J.S.; validation, J.S.; formal analysis, J.S.; writing—original draft preparation, J.S.; writing—review and editing, J.S. and V.S.; visualization, J.S.; supervision, V.S.; funding acquisition, V.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15 003/0000466).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kretz, M.; Lindenstruth, V. Vc: A C++ library for explicit vectorization. *Softw. Pract. Exp.* **2012**, *42*, 1409–1430. [CrossRef]
2. Krzikalla, O.; Feldhoff, K.; Muller-Pfefferkorn, R.; Nagel, W.E. Scout: A Source-to-Source Transformator for SIMD-Optimizations. In *Euro-Par 2011: Parallel Processing Workshops*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 137–145. [CrossRef]
3. McFarlin, D.S.; Arbatov, V.; Franchetti, F.; Puschel, M. Automatic SIMD vectorization of fast fourier transforms for the larrabee and AVX instruction sets. In Proceedings of the International Conference on Supercomputing—ICS '11, Vienna, Austria, 7–11 July 1997; ACM Press: New York, NY, USA, 2011; p. 265. [CrossRef]
4. Sha, D.; Lin, H.H. A multi-objective PSO for job-shop scheduling problems. *Expert Syst. Appl.* **2010**, *37*, 1065–1070. [CrossRef]
5. Cagnoni, S.; Bacchini, A.; Mussi, L. OpenCL Implementation of Particle Swarm Optimization. In *Applications of Evolutionary Computation*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 406–415. [CrossRef]
6. Tonti, L.; Patti, A. Fast Overlap Detection between Hard-Core Colloidal Cuboids and Spheres. The OCSI Algorithm. *Algorithms* **2021**, *14*. [CrossRef]
7. Holbrook, A.J.; Nishimura, A.; Ji, X.; Suchard, M.A. Computational Statistics and Data Science in the Twenty-first Century. *arXiv* **2022**, arXiv:2204.05530.
8. Shabanov, B.M.; Rybakov, A.A.; Shumilin, S.S. Vectorization of High-performance Scientific Calculations Using AVX-512 Intruction Set. *Lobachevskii J. Math.* **2019**, *40*, 580–598. [CrossRef]
9. Hemeida, A.; Hassan, S.; Alkhalaf, S.; Mahmoud, M.; Saber, M.; Bahaa Eldin, A.M.; Senjyu, T.; Alayed, A.H. Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor. *Ain Shams Eng. J.* **2020**, *11*, 1179–1190. [CrossRef]
10. Bramas, B. A fast vectorized sorting implementation based on the ARM scalable vector extension (SVE). *PeerJ Comput. Sci.* **2021**, *7*. [CrossRef] [PubMed]
11. Chen, J.; Gong, Z.; Tang, Y.; Zhang, Y.; Li, B. Optimizing Diffusion Layer with AVX Shuffling: A Study on SKINNY. In Proceedings of the 2022 7th IEEE International Conference on Data Science in Cyberspace (DSC), Guilin, China, 11–13 July 2022; pp. 227–233. [CrossRef]
12. Carneiro, A.R.; Serpa, M.S.; Navaux, P.O.A. Lightweight Deep Learning Applications on AVX-512. In Proceedings of the 2021 IEEE Symposium on Computers and Communications (ISCC), Athens, Greece, 5–8 September 2021; pp. 1–6. [CrossRef]
13. Langdon, W.B.; Banzhaf, W. Long-Term Evolution Experiment with Genetic Programming. *Artif. Life* **2022**, *28*, 173–204. [CrossRef] [PubMed]
14. Hossain, M.M.; Saule, E. Impact of AVX-512 Instructions on Graph Partitioning Problems. In Proceedings of the 50th International Conference on Parallel Processing Workshop, Lemont, IL, USA, 9–12 August 2021; pp. 1–9. [CrossRef]
15. Engelbrecht, A.P. *Computational Intelligence*, 2nd ed.; John Wiley & Sons: Hoboken, NJ, USA, 2007.
16. Zhang, G.; Shao, X.; Li, P.; Gao, L. An effective hybrid particle swarm optimization algorithm for multi-objective flexible job-shop scheduling problem. *Comput. Ind. Eng.* **2009**, *56*, 1309–1318. [CrossRef]
17. Harrison, K.R.; Engelbrecht, A.P.; Ombuki-Berman, B.M. Inertia weight control strategies for particle swarm optimization. *Swarm Intell.* **2016**, *10*, 267–305. [CrossRef]
18. Cleghorn, C.W.; Engelbrecht, A. Particle swarm optimizer. In Proceedings of the 2016 IEEE Symposium Series on Computational Intelligence (SSCI), Perth, WA, Australia, 27 November 1995–1 December 1995; pp. 1–7. [CrossRef]
19. Engelbrecht, A.P. Particle swarm optimization with crossover. *Artif. Intell. Rev.* **2016**, *45*, 131–165. [CrossRef]
20. Engelbrecht, A. Particle Swarm Optimization. In Proceedings of the 2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence, Ipojuca, Brazil, 8–11 September 2013; pp. 124–135. [CrossRef]
21. Lim, S.P.; Haron, H. Performance Comparison of Genetic Algorithm, Differential Evolution and Particle Swarm Optimization towards Benchmark Functions. In Proceedings of the 2013 IEEE Conference on Open Systems (ICOS), Kuching, Malaysia, 2–4 December 2013; pp. 41–46. [CrossRef]
22. Zhou, Y.; He, F.; Hou, N.; Qiu, Y. Parallel ant colony optimization on multi-core SIMD CPUs. *Future Gener. Comput. Syst.* **2018**, *79*, 473–487. [CrossRef]
23. Atashpendar, A.; Dorronsoro, B.; Danoy, G.; Bouvry, P. A scalable parallel cooperative coevolutionary PSO algorithm for multi-objective optimization. *J. Parallel Distrib. Comput.* **2018**, *112*, 111–125. [CrossRef]

24. Nuzman, D.; Dyshel, S.; Rohou, E.; Rosen, I.; Williams, K.; Yuste, D.; Cohen, A.; Zaks, A. Vapor SIMD. In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2011), Chamonix, France, 2–6 April 2011; pp. 151–160. [CrossRef]
25. Nedjah, N.; de Moraes Calazan, R.; de Macedo Mourelle, L. Particle, Dimension and Cooperation-Oriented PSO Parallelization Strategies for Efficient High-Dimension Problem Optimizations on Graphics Processing Units. *Comput. J.* **2016**, *59*, 810–835. [CrossRef]
26. Govindaraju, V.; Nowatzki, T.; Sankaralingam, K. Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG. In Proceedings of the Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, UK, 7–11 September 2013; pp. 341–351. [CrossRef]
27. Trouvé, A.; Cruz, A.J.; Brahim, D.B.; Fukuyama, H.; Murakami, K.J.; Clarke, H.; Arai, M.; Nakahira, T.; Yamanaka, E. Predicting Vectorization Profitability Using Binary Classification. *IEICE Trans. Inf. Syst.* **2014**, *E97.D*, 3124–3132. [CrossRef]
28. Boettcher, M.; Al-Hashimi, B.M.; Eyole, M.; Gabrielli, G.; Reid, A. Advanced SIMD. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 24–28 March 2014; pp. 1–4. [CrossRef]