Predicting Loop Vectorization through Machine Learning Algorithms

Esraa H. Alwan
Department of Computer Science, Collage of science for women University of Babylon, Iraq

Emails: esraa.hadi@uobabylon.edu.iq

Abstract

Automatic vectorization is often utilized to improve the speed of compute-intensive programs on current CPUs. However, there is enormous space for improvement in present compiler auto-vectorization capabilities. Execution with optimizing code on these resource-controlled strategies is essential for both energy and performance efficiency. While vectorization suggests major performance developments, conventional compiler auto-vectorization techniques often fail. This study investigated the prospective of machine learning algorithms to enhance vectorization. The study proposes an ensemble learning method by employing Random Forest (RF), Feedforward Neural Network (FNN), and Support Vector Machine (SVM) algorithms to estimate the effectiveness of vectorization over Trimaran Single-Value Code (TSVC) loops. Unlike existing methods that depend on static program features, we leverage dynamic features removed from hardware counter-events to build efficient and robust machine learning models. Our approach aims to improve the performance of e-business microcontroller platforms while identifying profitable vectorization opportunities. We assess our method using a benchmark group of 155 loops with two commonly used compilers (GCC and Clang). The results demonstrated high accuracy in predicting vectorization benefits in e-business applications.

Keywords: Automatic vectorization; ensemble learning; Random Forest; Feedforward Neural Network (FNN); compiler optimization

1. Introduction

The latest computers typically utilise vector instructions to execute several essential operations concurrently. Present-day systems comprise different parallelism stages like functional areas in a CPU core, sophisticated instruction sets, and concurrent thread execution that the instructions use for concurrent processing [1]. Interestingly, single instruction multiple data (SIMD) instructions can simultaneously process data "vectors". Therefore, vectorisation means adapting code to use such instructions [2, 3]. Despite noteworthy performance benefits associated with manual vectorisation, there are concerns like error rate, time requirement, and code maintenance and portability characteristics. In contrast, automatic vectorisation is the predominant technique because it does not require coders to understand intricate SIMD instruction sets or face the stated disadvantages of manual vectorisation [4]. Commonly used compilers such as Open64, Intel® C Compiler (ICC), GNU Compiler Collection (GCC), and Clang offer automatic vectorisation support, facilitating better performance than scalar code. Nevertheless, the performance of vectorised code may be significantly degraded when benchmarked against peak CPU performance despite the dataset residing completely in the L1 cache and zero instances of cache miss. Mainstream compilers convert code, such as high-level nested loops, into assembly language; hence, the vectorised form is produced slowly. The intricacies lead to challenges regarding efficient and effective code conversion. Moreover, compilers build vectors using heuristics at the middle and back end. In order to reduce performance decline, compilers may sometimes disregard vectors and attach the original scalar version if vectorisation fails to

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024

enhance speed [4]. Nevertheless, compilers cannot always accurately detect scenarios where vectorisation does not offer benefits. This can be attributed to the rising intricacies of computer architecture and compilers; hence, it becomes challenging to estimate behaviour and develop dependable optimisation techniques. Contemporary architectures are growing progressively complex, comprising new vector instructions, several parallelism stages (e.g., vector instructions, multi-core use, and instruction-level parallelism (ILP)), and complex memory stages. Such intricacies create immense challenges that prevent compilers from building precise models [1]. The compiler community is growing increasingly curious about employing machine learning (ML) models [5, 6]. This study assesses the ability of ML approaches to improve vectorisation. Prevailing ML-based approaches for optimising compilers typically use a single learning technique, which may be prone to model fitting challenges or suboptimal classification precision. Ensemble learning employs several approaches to enhance prediction characteristics and trim variance while controlling bias. Hence, this research recommends conducting binary classification using an ensemble learning approach. The approach is based on three ML methods: random forest (RF), a feedforward neural network (FNN), and a support vector machine (SVM) to estimate if a program may benefit from vectorisation. Evaluating vectorisation profitability is tackled as a classification problem in this study. The remaining work is divided into the following sections: Section 2 offers a short overview of related studies. Section 3 presents the suggested methods. Section 4 discusses the experimental outcomes and inferences related to the suggested approach. Lastly, Section 5 presents the conclusion.

2. Related works

Vectorisation has been researched extensively. Researchers in [1] used ML techniques to estimate the performance of SIMD code. Opposed to older techniques based on high-level programme data, the authors devised frameworks using features identified using the produced assembly code. The authors chose benchmarks for offline model training. It was then employed during compilation to differentiate several vectorised versions representing the input code. This technique's efficacy was established by facilitating automatic vectorisation on several tensor contraction kernels. The outcomes indicate 2x to 8x performance levels when benchmarked against Intel® ICC's auto-vectorised code baselines. Another research [7] recommends using a classification route to devise a vectorisation profitability scenario and conducting estimation using an SVM-specific route. This technique considers three compilers: ICC, GCC, and low-level virtual machine (LLVM). The control comprises a benchmark set of 151 loops comprising unrolling factors between 1 and 20. For each of the compilers, the researchers suggest a technique for integrating the outcomes of two SVMs. Specifically, this method attains speedup levels up to 2.16x, 2.47x, and 3.83x for ICC, GCC, and LLVM, respectively (averaging 9%, 18%, and 56%). Moreover, the likelihood of the compiler producing a suboptimal programme is reduced below 1%. In [8,] dynamic data (hardware counters) are assessed based on supervised ML approaches for the vector estimation problem. The framework estimates if a nested loop set should be vectorised automatically or manually, including the probable benefits of vectorisation. Such an estimation benefits from the auto-vectorisation function provided by two mainstream compilers: ICC and GCC. Dynamic data efficacy is evaluated comprehensively using six ML techniques: SVMs, Naïve Bayes, RF Trees, K-nearest neighbours, logistic regression, and others. Researchers [9] outlined an ensemble learning-specific approach to enhance the efficacy of automatic vectorisation. The authors devised an ensemble learning system that predicted the efficacy of tensor contraction kernels based on several vectorisation approaches and identified the most suitable vectorisation technique in the context of those kernels. Our recommendation comprises a static feature representation method that uses tensor contraction kernels' storage access outlines. The ELAV approach consumes TC kernels' VEC features as inputs; it is a static feature encoding technique that denotes the storage use patterns of such kernels. The ELAV technique estimates programme characteristics through varying vectorisation approaches. Moreover, the ELAV framework possesses an average estimated time of some seconds, allowing straightforward integration with present compilation techniques that have different instruction sets, varied data types, and compilers. The research cited in [10] evaluated the efficiency of automated vectorization, investigated its limitations and fundamental reasons, and enhanced the related technology. The original study classified programs into two categories: program features and transformation techniques. Following that, it assessed the efficacy of automated vectorization in three commonly used compilers (GCC, LLVM, and ICC), spanning numerous versions over the last five years. The benchmark was used in the assessment. The authors in [11] provide Autograph, an end-to-end system built for compiler auto-vectorization, in their paper. Autograph recognizes loops, builds dependency graphs, and learns structured representations that include both the structural dependencies of the computation graph and the semantics of the code. Autograph predicts vectorization factors and inserts vectorization pragmas with optimal VF/IF factors into the code using a deep reinforcement learning technique, resulting in increased performance. Autograph delivers an average performance boost of 2.47x for Polybench compared to neurovectorizer and 3.61x compared to the baseline after extensive testing and comparisons across different benchmark datasets. According to the above discussed studies, there are several key challenges associated with automatic vectorization. Current compilers frequently fail to identify situations where

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024

vectorization offers limited benefits, resulting in suboptimal performance compared to CPU peak performance. The increasing complexity of both compilers and architectures makes it difficult to accurately predict behavior and build reliable optimization methods. This complexity includes factors like transformation challenges from high-level loops to assembly code, reliance on heuristics in compilers for vector instruction construction, unique vector instructions in modern architectures, multiple levels of parallelism exploitation, and intricate memory hierarchies. Therefore, to improve the accuracy of identifying situations for useful vectorization and to reduce the risk of generating vectorized code that underperforms compared to the original code, this work proposes an ensemble learning model for binary classification to address the limitations of existing automatic vectorization approaches. It leverages three machine learning techniques including Random Forest (RF), Feedforward Neural Network (FNN), and Support Vector Machine (SVM) to predict whether a program can be profitably vectorized.

3. Methodology

Our overall approach focuses on ensemble learning where three machine learning algorithms are used to predict how programs will perform based on a set of input characteristics that describe the program. We start by using a dataset to extract features. These features are then used as input vectors to the ensemble learning model. In our research, we collect data from hardware event counters using a dataset created from compiled loops that have been performed and analyzed. This runtime information is then utilized to generate a feature set from which the classifier's features may be selected. The goal of estimating the profitability of vectorization is approached as a classification problem. We identify programs that profit from vectorization with a 1 and ones that do not with a 0 as illustrate in figure 1.

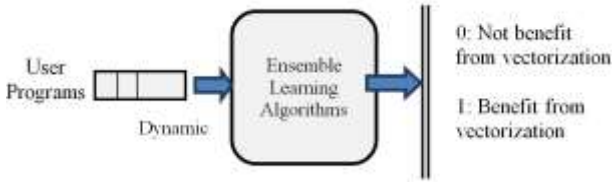


Figure 1: Proposed method

We then train a classifier to predict the class for new programs. This training process happens offline. The flowchart diagram of the proposed ensemble learning methodology for program vectorization is shown in Figure 2.

This ensemble learning methodology for program vectorization leverages three machine learning algorithms, trained on features extracted from hardware counters, to predict whether a program benefits from vectorization. Hardware events like branching and cycles are measured during non-vectorized code execution to generate features. Random Forest, Feedforward Neural Network, and Support Vector Machine then learn from these features to predict the binary classification of "vectorize" or "don't vectorize" for unseen programs. The final prediction is made through majority vote among the individual model predictions.

Data Collection

The test suite for vectorising compilers (TSVC) benchmark dataset was employed to train the model used in the study. The model comprised 151 loops written in C. The benchmark was devised chiefly to assess the auto-vectorisation abilities of different compilers. Specifically, the benchmark uses loops, each of which runs inside a control loop, facilitating precise processing time determination by iterating the run. Callahan, Dongarra, and Levine devised TSVC; Maleki et al. [11] [12] developed it further to assess the test collection for automatic vectorisation capabilities. This study uses the TSVC variant advanced by Maleki et al. to determine automatic vectorisation capability. The TSVC is a simple-to-understand application. It comprises nested loops that conduct several conventional processing steps and memory operations like unconditional jump, branch determination, and indirect array access, based usually on single-precision floating-point numbers. From an academic perspective, TSVC is typically employed to assess the efficacy of automated vectorisation. Recently, there has been increased interest in TSVC by the open-source industrial compiler community. TSVC has evolved and has now become the LLVM benchmark test set [1].

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024

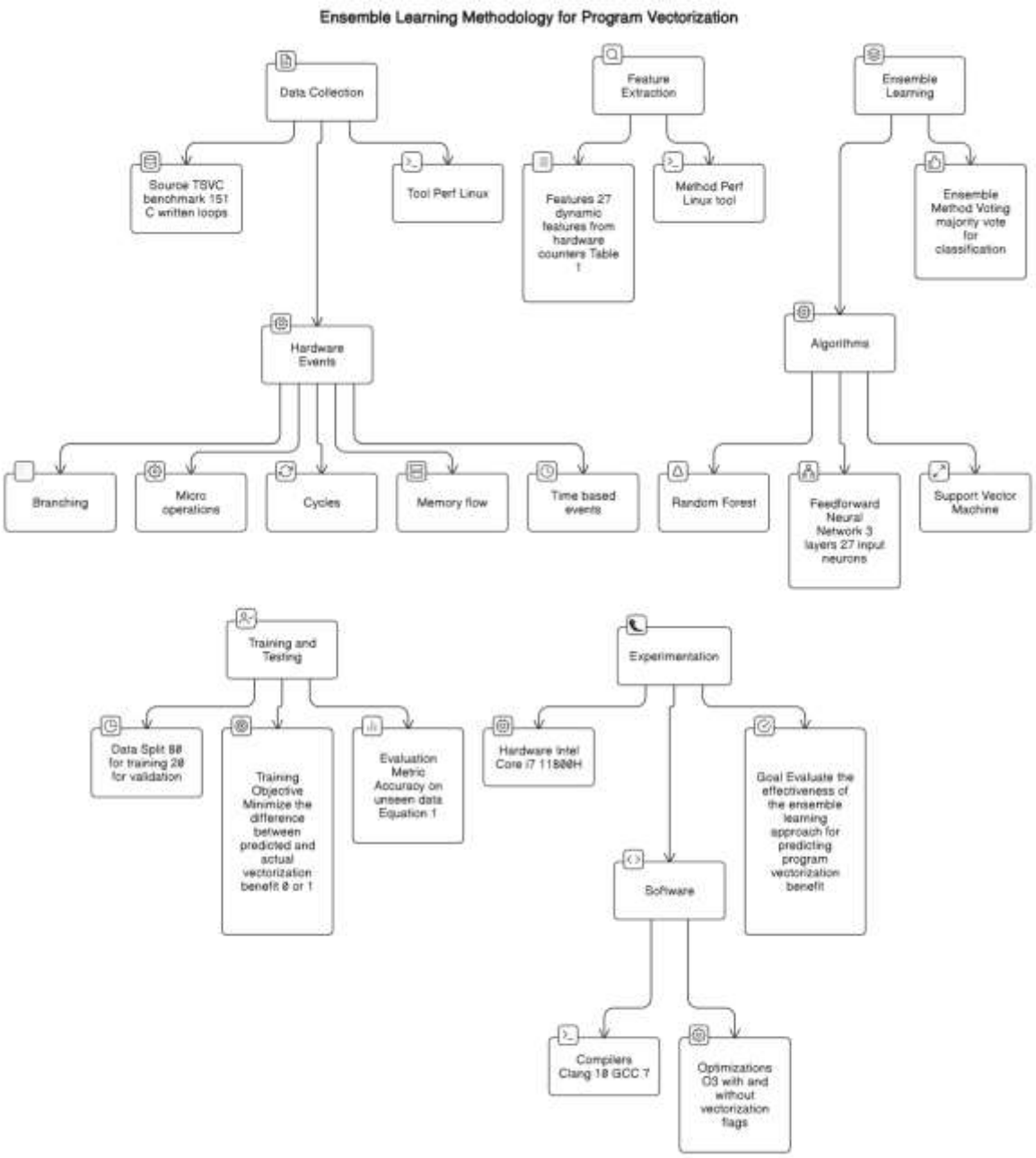


Figure 2: Ensemble learning methodology diagram for program vectorization.

Features Extraction

Performance characteristics are used to derive the inputs fed to the ML model. Considering that the study objective is to forecast vectorisation, the data is obtained using non-vectorised programmes where vectorisation is turned off but -O3 optimisation is on. The data-collection process comprised numerous hardware events being logged. Such events comprised memory flow, micro-operations, cycles, branching, and time-based activities. Table 1 lists the specific performance occurrences we collected. When processing the first feature set, it is vital to assess and remove elements representative of noise or the ones that do not relate to the model output.

Integrating such elements may trigger falls in the estimation framework. The feature set used for this research is dynamic because the elements are gathered by programme execution; these values change as the system and runtime configuration is modified. A framework based on dynamic features implicitly indicates the environment in its performance properties.

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024

Table 1: The Dynamic Features

Cpu-cycles or cycles, instructions, Cashe- references, Cashe-misses, Bus_cycles	Hardware events
Cpu_clock(msec), Task_clock(msec), Page- faults OR faults, Context-switches, Cpumigrations, Alignment-faults, Emulationfaults	Software events
L1-dcashe-loads, L1-dcashe-loads misses, L1-dcashe-stores, L1-dcashe-storesmisses, L1-icashe-loads, L1-icashe-loadsmisses, L1-icashe-loads, L1-icashe-loads misses, L1-icashe-prefetches, L1-icasheprefetches –misses, LLC-load, LLC-loadsmisses, LLC-strores, LLC-strores misses, LLC-prefetch-misses, Dtlb-loads,Dtlb-loadsmisses, Dtlb-store, Dtlb-store-misses, Dtlb prefetches, Dtlb-prefetches -misses, Itlb -loads, Itlb –loads -misses, Branch-loads, Branch-loads-misses	Hardware cache events

For the automatic collecting of performance counters during execution, we utilized the Perf Linux tool [13][14]. We changed a single, shared header file across all benchmarks to incorporate the performance counter choices, therefore no benchmark programs needed to be changed.

Ensemble Learning

To handle the same set of problems ensemble learning employs combined homogeneous as well as heterogeneous learning techniques, resulting in a number of learners. We apply heterogeneous learning methods in our work. The voting approach is then used to combine predictions by getting a majority vote (for classification) of the outputs of the learning algorithms. As a result, the final decision is obtained according to the voting procedure.

Training and Testing

This research uses ensemble learning to determine a binary outcome, suggesting whether vectorisation would benefit the programme. This case is representative of a classification problem where it is vital to differentiate the two sets (profitable and non-profitable). Model training is conducted as specified: several benchmark programmes are used, and the associated performance counter occurrences are used to gather the corresponding feature vectors. This study employs the Perf/Linux tool. Model training is based on a 27-feature vector. The following three classification techniques were used for model training:

Random Forest (RF): This approach diminishes overfitting and is less affected by noise in data. The trees are built using random feature subsets. The estimated value is calculated based on the average of the trees selected for training. Different tree counts like 5, 7, 9, 11, 13, and 15 were evaluated.

Feedforward Neural Network (FNN): It is a supervised machine learning technique used to train artificial neural networks. It is a basic neural network where node linkages are acyclic. It is typically presented as a set of linked layers: an input layer, one or several hidden layers, and an output layer. Network learning and performance enhancement are based on minimising the difference between the actual and predicted values. Figure 3 depicts the developed network comprising three layers having 27 input neurons and a single output.

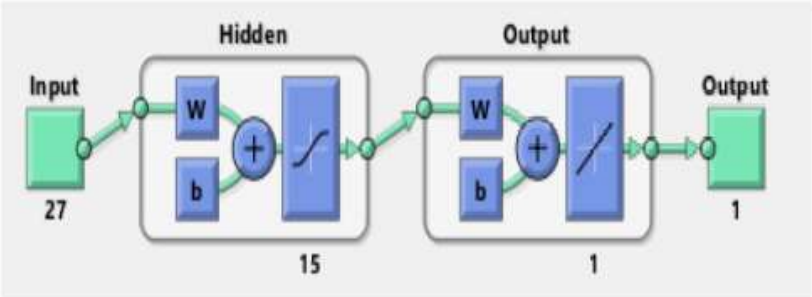


Figure 3: Feedforward Neural Network

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024

Support Vector Machine (SVM): It operates using the hyperspace, which is an imaginary multidimensional space. Each element is indicated as a point in that hyperspace where features serve as coordinates. A hyperplane is then computed to split the data, and support vectors help delineate class boundaries. When a fresh data element is evaluated, the framework assigns the predicted class by evaluating the side at which the element is situated relative to the hyperplane. It can filter features, and it is a very efficacious discriminative classifier. Moreover, there is a high degree of versatility since the user can alter the dimensionality of the kernel the SVM will process. The study model was validated using a distinct validation set developed by a random 80/20 split of the original dataset (80% for training and 20% for validation). Model quality is gauged by its tendency to classify fresh inputs provided after training. Accuracy is often evaluated using data points not present in the training dataset, as expressed in Equation 1:

$$accuracy(model) = \frac{correct\ predictions}{number\ of\ test\ samples} \quad (1)$$

Experiments

The first test is devised to determine the correlation between compiler flags from an auto-vectorisation perspective (indicating if a compiler successfully vectorises a specified loop). The test was performed on a system based on the Intel® Core™ i7-11800H architecture. The other test evaluated the use of ensemble learning integrated with hardware counter instances. Programme feature training for dissimilar compilers was based on three ML techniques. The suggested technique is devised for architectural independence. We performed a suite of tests based on an Intel® processor testbed to evaluate its effectiveness. Table 2 lists the hardware configurations used for these tests. The architectural configuration comprised TSVC dataset compilation and execution based on Clang and GCC 5.4; the -O3 optimisation comprising vector-specific flags was on (-fno-tree-vectorise option for GCC and -fno-vectorise for Clang). The test objective was to gather hardware counters that lacked vector manipulations. Further, the dataset was re-executed after compilation, integrating the -O3 optimisation flag and vector flags (-ftree-vectorise option for GCC and -fvectorise for Clang).

Table 2: Hardware configurations

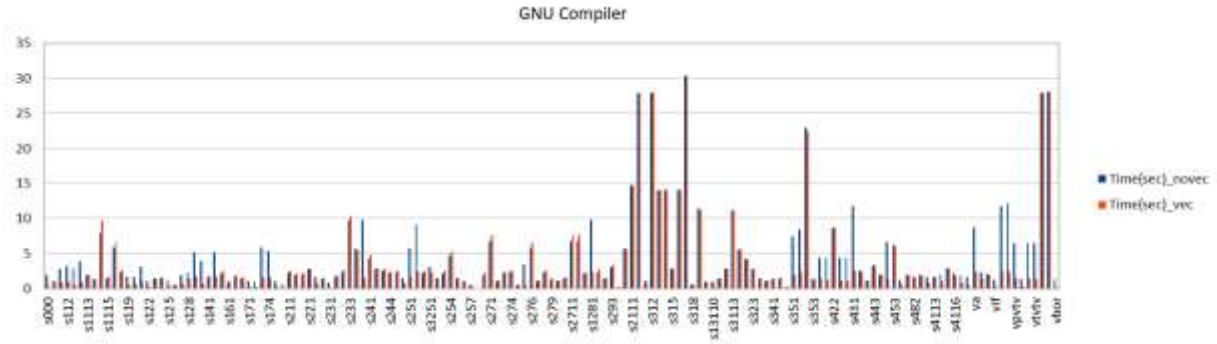
Hardware setup	
Microarchitecture	Tiger Lake
Operating System	Ubuntu 20.
Processor	Core i7-11800H
Vector Instruction Set Extensions	sse, sse2, ssse3 sse4_1, sse4_2, avx avx2, avx512
Processor Frequency	2.30 GHz
RAM	16.0 GB
Compiler	
Clang	Clang 10
GNU compiler collection	GCC 7

4. Results and discussion

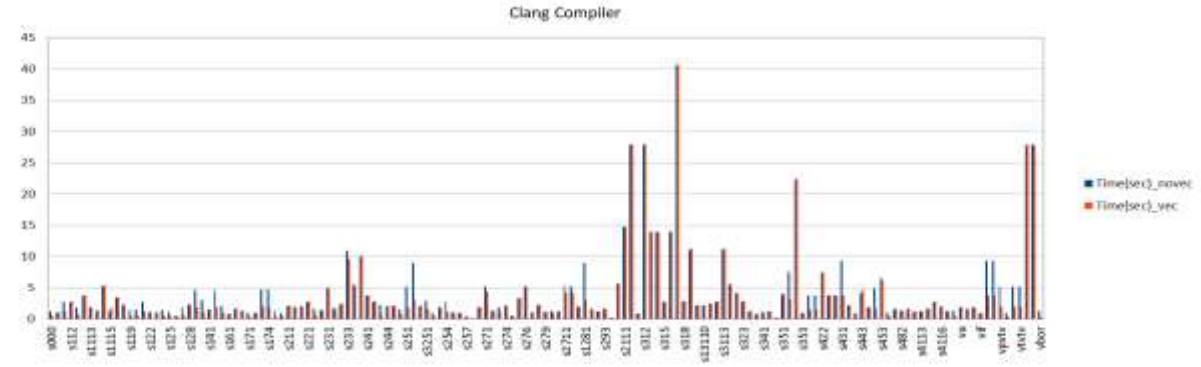
Evaluation of the Compile Time

This study assesses the time needed by compilers to process TSVC. The study dataset comprises 151 TSVC kernels. The initial test does not consider vectorisation profitability; the emphasis is solely on the possibility of vectorising the loop. Figures 4 (a) and (b) depict a statistical plot of GCC and Clang time needed to compile TSVC when automated vectorisation was on and off. The use of -O3 optimisation allowed GCC to vectorise 55 loops, while Clang vectorised 49 for the Intel® Core™ i7-11800H architecture. It highlights the speed of optimisation and compilation specific to TSVC programmes for a set compiler version. The horizontal axis depicts the loop names, while the vertical axis comprises the compilation time (seconds) required by GCC and Clang compilers.

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024



(a)



(b)

Figures 4 (a), and (b) represent statistical graphs for TSVC when it is compiled with GCC and Clang.

The experiment analyzed the compilation time of 151 TSVC kernels by GCC and Clang compilers, focusing solely on whether the loops were vectorized or not. The results are displayed in Figure 4 (a) and (b). With optimization level -O3 enabled, GCC vectorized 55 loops compared to Clang's 49 on a Core i7-11800H architecture. These graphs offer insights into the compilation and optimization speed of TSVC programs for specific compiler versions, where loop names are on the horizontal axis and compilation time in seconds is on the vertical axis. Overall, the experiment highlights compiler differences in handling loop vectorization and its impact on compilation time.

Different compiler

This experiment evaluates two compilers, GCC and Clang, using a performance prediction model with three machine learning algorithms implemented in MATLAB. Separate datasets are built for each compiler, trained with RF, FNN, and SVM algorithms. As shown in Table 3, both RF and FNN achieve an accuracy of 85.7%, while SVM lags behind at 71.4%.

Table 3: Accuracy of Clang compiler

Classifier	Accuracy
RF	90.4
FNN	76.19
SVM	71.4
EL	90.4

We first present the results of the Random Forest (RF) classifier. It takes 27-dimensional feature vectors representing benchmark programs as input and outputs a class label (0 or 1). The best accuracy of 90.4% was achieved with 11 trees.

Next, the Feedforward Neural Network (FNN) with 27 input nodes and one output node achieved an accuracy of 76.19% using 15 hidden nodes. Finally, the Support Vector Machine (SVM) achieved 71.4% accuracy. Ensemble learning significantly improves the final result, reaching 90.4% accuracy as shown in Figure 5.

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024

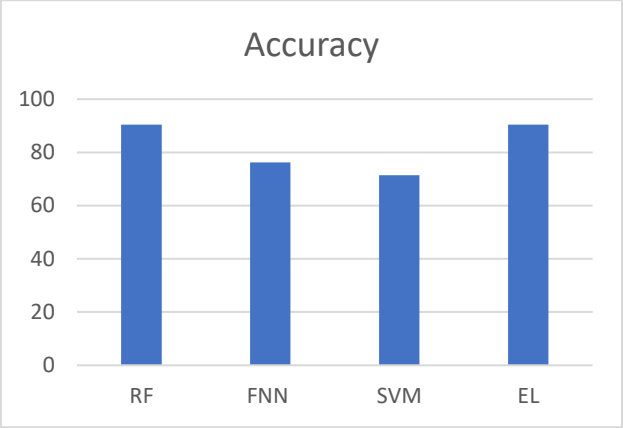


Figure 5: The accuracy of the Clang compiler

The individual machine learning models, including Random Forest (RF), Feedforward Neural Network (FNN), and Support Vector Machine (SVM), achieved varying levels of accuracy in predicting program vectorization benefit. The RF classifier performed the best with an accuracy of 90.4%, highlighting its effectiveness in this task. The FNN and SVM models achieved lower individual accuracies of 76.19% and 71.4%, respectively. However, the crucial finding is that by leveraging ensemble learning, which combines the predictions of these individual models, the overall accuracy significantly improved, reaching the same 90.4% achieved by the best performing individual model (RF). This demonstrates the effectiveness of ensemble learning in potentially enhancing the prediction accuracy compared to relying on solely one model. It's important to note that Figure 5 seems irrelevant to the discussion of model performance and might be referencing a different aspect of the research that needs clarification.

Result with GCC compiler

In Table 4, the classifier has 95.7 accuracy for RF and 90.4 for FNN, while the SVM the accuracy was 85.5.

Table 4: Accuracy of GCC compiler

Classifier	Accuracy
RF	100
FNN	90.4
SVM	85.5
EL	100

We use the same previous structure for all classifiers. For the RF we got the best result with number of trees equal to 9. For the second classifier FNN we got the best result with a number of hidden nodes equal to 15. Finally, with SVM we got accuracy equal to 80.9%. After applying the ensemble learning the accuracy of the final result improves. It reaches 100% as illustrated in Figure 6.

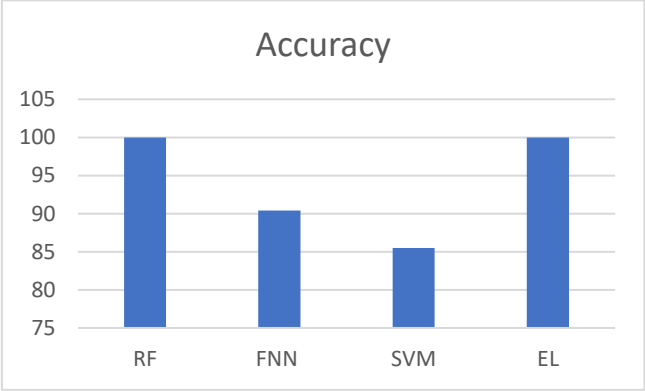


Figure 6: The accuracy of GCC compiler

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024

Comparison with other methods

Our proposed technique, which utilizes ensemble learning and dynamic features, is compared to the single-learner algorithms presented in [19]. While their approach achieves an accuracy of 0.88 with a Random Forest and 0.82 with a Support Vector Machine ([19]), our method demonstrates improved classification accuracy. Additionally, we compare our work to the ensemble learning methods using logistic regression, decision trees, linear regression, and artificial neural networks in [1]. While their ELAV approach achieves 88% and 87% accuracy on two different systems, our method offers advantages.

5. Conclusions

In this paper, we present an ensemble learning-based automated vectorization performance optimization strategy for predicting the profitability of automatic vectorization for GCC and Clang Compilers. We employ TSVC, a benchmark consisting of 151 basic typical loops. The usefulness of leveraging hardware performance data in supervised machine learning models to predict compiler auto-vectorization (for clang) was investigated. We identify situations where vectorization is feasible with a validation accuracy of 100% for the GCC compiler and 90% accuracy for the Clang compiler. By utilizing the resilience of various machine learning algorithms through ensemble learning, this work contributed to more efficient and robust automatic vectorization, specifically for e-business microcontroller applications. While the proposed ensemble learning approach offers promising advancements in predicting profitable vectorization opportunities, it is important to acknowledge its limitations and future research directions. One limitation lies in the generalizability of the model. We need to investigate more benchmarks and their effectiveness for broader architectures, or hardware counter features needs further investigation.

Acknowledgments

The researchers would like to thank the University of Babylon (Iraq) / College of Science for Women for supporting this work.

Conflict of interest statement

The authors have no conflicts of interest to declare.

Refrences

[1] K. Stock, L. Pouchet, and P. Sadayappan, "Using machine learning to improve automatic vectorization," *ACMTrans. Archit. Code Optim*., Vol. 8, No. 4, pp. 123, 2012.

[2] A. Haj-Ali, N. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica.,"Neurovectorizer:End-toend vectorization with deep reinforcement learning",*In Proceedings of the 2020 International Symposium on Code Generation and Optimization, CGO 2020.*ACM, pp. 242–255, 2020.

[3] H. Amiri, A. Shahbahrami, A. Pohl and B. Juurlink, "Performance evaluation of implicit and explicit SIMDization", *Microprocessors and Microsystems*, Vol. 63, pp. 158-168, 2018.

[4] R. Barik, J. Zhao, and V. Sarkar. "Automatic vector instruction selection for dynamic compilation",*In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT'10,* New York, NY, USA . ACM, pp. 573–574, 2010.

[5] S. Siso, W. Armour, and J. Thiyagalingam, "Evaluating auto-vectorizing compilers through objective withdrawal of useful information," *ACM Trans. Archit. Code Optim*., Vol. 16, No. 4, 2019.

[6] C. D. Rickett, S.E. Choi, B.L. Chamberlain, "Compiling high-level languages for vector architectures" . In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing (LCPC'04)*,pp.224–237, 2005

[7] A. Trouve, A. J. Cruz, D. B. Brahim, H. Fukuyama, K. J. Murakami, ´ H. A. Clarke, M. Arai, T. Nakahira, and E. Yamanaka, "Predicting vectorization profitability using binary classification," *IEICE Transactions*, Vol. 97-D, No. 12, pp. 3124–3132, 2014.

[8] N. Watkinson and et. al. "Using hardware counters to predict vectorization",*In Languages and Compilers for Parallel Computing (LCPC 2017).* Springer, 2017.

[9] H.Liu, R. Zhao, K. Nie, "Using Ensemble Learning to Improve Automatic Vectorization of Tensor Contraction Program", *IEEE Access* ,Vol. *6*, pp.47112–47124, 2018.

[10] J. G. Feng, Y. P. He, and Q. M. Tao, "Evaluation of compilers' capability of automatic vectorization based on source code analysis",*Scientific Programming*, Vol. 2021, 2021.

[11] S. Maleki, Y. Gao, M. J. Garzaran et al., "An evaluation of vectorizing compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT*), TX, USA, pp. 372–382, 2011.

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024

[12] W. Gao, R. C. Zhao, L. Han, J. M. Peng, and R. Ding, "Research on SIMD automatic vectorization compiling optimization," *Ruan Jian Xue Bao/Journal Of Software*, vol. 26, no. 6, pp. 1265–1284, 2015.
[13] M. Almohammed, A. Fanfakh, and E. Alwan, "Parallel genetic algorithm for optimizing compiler sequences ordering", *In Proc.CCIS*, pp. 128-138, 2020.
[14] L. H. Alhasnawy, E. H. Alwan, and A. B. M. Fanfakh, "Using machine learning to predict the sequences of optimization passes" , in Proc. CCIS, pp.139-156, 2020.
[15] H. Liu, J. Xu, S. Chen, T. Guo, "Compiler Optimization Parameter Selection Method Based on Ensemble Learning", *Electronics*, Vol.11, No.15, pp.2452, 2022.
[16] R. Kruppe, J. Oppermann, L. Sommer, A. Koch. "Extending LLVM for lightweight SPMD vectorization: Using SIMD and vector instructions Easily from any language", *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 278–279, 2019.
[17] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr. "A simd optimization framework for retargetable compilers". *ACM Trans. Archit. Code Optim.*, Vol. 6, No.1,pp. 2:1–2:27, 2009.
[18] D. S. McFarlin, V. Arbatov, F. Franchetti, andM. Püschel, "Automatic simd vectorization offastfourier transforms for the larrabee and avx instructionsets". In *Proceedings of the International Conference onSupercomputing*, ICS'11, New York,NY, USA, ACM. pp. 265–274, 2011.

Doi: https://doi.org/10.54216/FPA.150203
Received: August 10, 2023 Revised: November 22, 2023 Accepted: March 16, 2024