Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 00 (2013) 000–000

Procedia
Computer Science

www.elsevier.com/locate/procedia

The Eighth International Workshop on Automatic Performance Tuning, iWAPT 2013

# Using Machine Learning in order to Improve Automatic SIMD Instruction Generation

Antoine Trouvé[a,*], Arnaldo Cruz[a,b], Hiroki Fukuyama[a,b], Jun Maki[b], Hadrien Clarke[a,b], Kazuaki Murakami[a,b], Masaki Arai[c], Tadashi Nakahira[c], Eiji Yamanaka[d]

[a]*Department of Informatics, ISEE, Kyushu University, Japan*
[b]*Institute of Systems, Information Technologies and Nanotechnologies, Japan*
[c]*Fujitsu Laboratories Ltd, Japan*
[d]*Fujitsu Ltd, Japan*

**Abstract**

Basic block vectorization consists in extracting instruction level parallelism inside basic blocks in order to generate SIMD instructions and thus speedup data processing. It is however a double-edged technique, because the vectorized program may actually be slower than the original one. Therefore, it would be useful to predict beforehand whether or not vectorization could actually produce any speedup. In this article, we propose to do so by using a machine learning technique called support vector machine. We consider a benchmark suite containing 151 loops, unrolled with factors ranging from 1 to 20. We do our prediction offline *after* as well as *before* unrolling. Our contribution is threefold. First, we manage to predict correctly the profitability of vectorization for 70% of the programs in both cases. Second, we propose a list of static software characteristics that successfully describe our benchmark with respect to our goal. Finally, we determine that machine learning makes it possible to significantly improve the quality of the code generated by Intel Compiler, with speedups up to 2.2 times.

*Keywords:* Machine Learning, Support Vector Machine, Compiler Optimization, Vectorization, SIMD

## 1. Introduction: Problem Definition

Single Instruction Multiple Data (SIMD) is a paradigm which enables to dramatically raise the peak performance of processors as well as their power efficiency. This is particularly important in high performance computing and embedded systems, in which higher performance is required within a constant power envelope. For short vector sizes (up to 8 or 16), software engineers can rely on vendor compilers (e,g. Intel Compiler) for automatically generating SIMD instructions: this is called automatic vectorization.

In this paper, we focus on one automatic vectorization technique called automatic-basic-block vectorization, referred to as ABBV in the rest of this paper. ABBV consists in leveraging the inherent instruction-level parallelism (ILP) inside basic blocks in order to generate SIMD instructions. The reader should keep in mind that ABBV is different from loop vectorization[1] and software pipelining, not considered in this paper [11]. ABBV is carried out in the compiler's backend and mainly relies on pattern matching. The quality of the results greatly depends on the input to the backend, that is, basic blocks should exhibit enough ILP. The amount of ILP is not

*Corresponding author. Tel.: +81-92-852-3460 ; fax: +81-92-852-3465; e-mail: trouve@isit.or.jp
[1]The reader should further notice that automatic vectorization is equivalent to loop unrolling followed by ABBV.

only an inherent property of the compiled program but also it is greatly affected by the front and middle ends of the compiler, for instance by unrolling loops. Still, sometimes no benefit can be obtained by using vector instructions and it may be more advantageous to only use traditional scalar instructions. This may happen because vector instructions in modern processors often involve some overheads like data packing and unpacking or very slow access to unaligned data in the memory subsystems. Hence, compilers that carry out ABBV (we call them *vectorizing compilers*) should be able to: (1) not vectorize if it is not profitable, that is, if the execution time of a program compiled with ABBV is higher than the same without it, and (2) transform the code ahead of the backend in order to feed the best possible input to the ABBV optimizer. We refer to these properties as **P1** and **P2** in the remainder of this paper.

However, modern compilers are far from optimal with respect to both P1 and P2. Let us consider Intel Compiler, which is well known for the high quality of its output. We report in section 4 that 44% of the programs vectorized by Intel Compiler using only ABBV are slower than without vector instructions. This blatantly shows its failure with respect to P1. Moreover, works from various teams at University of Illinois, USA [2], or INRIA, France [7], have shown that it is also weak with respect to P2 even on simple examples[2]. In this paper, we focus on one code transformation: loop unrolling. We have measured that if we do not assist Intel Compiler into unrolling the program upstream, it fails to generate profitable vectors for 90% of our benchmarks. This occurs despite the fact that there exists an unroll factor that allows profitable vectorization for 45% of it.

In this paper, we propose to determine ahead of the backend whether or not a program can be profitably vectorized downstream by the ABBV optimizer, the Intel Compiler in our case. To do so, we utilize the support vector machine (SVM) learning algorithm. We carry out two experiments that differ in their inputs and that address P1 and P2 respectively (see section 4.3). First, we consider a program *after* being transformed in order to reproduce the situation encountered in the backend of a vectorizing compiler, when it decides whether or not to vectorize. Second, we consider the program *before* being transformed, as a middle-end would do before deciding which code transformation to apply. In the remaining of this paper we will often refer to these experiments simply as **Exp1** and **Exp2**, respectively. We were able to obtain an accuracy of approximatively 70% for both, which is far above the current capabilities of the Intel Compiler. Our success not only comes from the power of SVM, but also from the software characteristics we consider in order to describe the input programs.

This paper is organized as follows. First we justify our choice of using machine learning in order to predict the profitability of vectorization (section 2), before explaining how we do so (section 3). Next, we present and analyze our experimental results (section 4). We conclude by presenting the software features that describe the input programs (section 5), before exploring related works that use machine learning in the context of program optimization (section 6).

## 2. Why Machine Learning ?

Vectorizing compilers should be as good as possible with respect to properties P1 and P2, described in section 1. P1 can be decided by actually running the program and discarding vectorization it proves not to be profitable. This would however obviously require too much time. Compilers use heuristics inside the backend instead. In order to address P2, modern compilers often leverage some feedback from the backend to the middle-end. This however complicates the implementation of compilers and may initiate some time-consuming back-and-forth communications inside the compiler. As a consequence, compiler designers tend to favor heuristics for P2 as well, which are implemented straight inside the middle-end. In summary, modern compilers tend to use heuristics to tackle both P1 and P2. However, this approach behaved poorly for our benchmarks (see section 4.2). Indeed, a vectorizing compiler is a complex piece of software whose behavior is very hard to predict, even for its own designers[3]. In this context, the design of heuristics to predict the profitability of vectorization amounts to dark magic.

In this paper, we decide to acknowledge this situation and we regard the compiler as a black box. We merely consider its inputs and outputs, and try to determine if we could guess some properties from them. This relates

[2]These works consider all kinds of vectorization, not only ABBV

[3]The reader may want to take a look at, for example, GCC's mailing list to verify this assertion

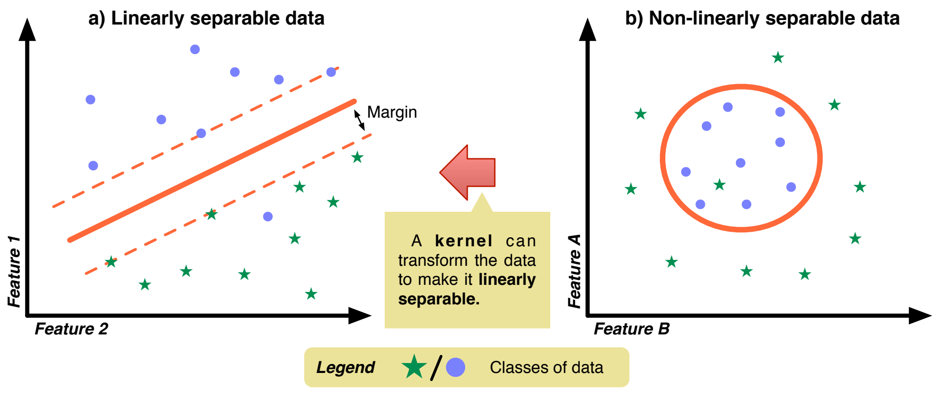
a) Linearly separable data
Margin
Feature 1
Feature 2
A kernel can transform the data to make it linearly separable.
b) Non-linearly separable data
Feature A
Feature B
Legend
Classes of data

Fig. 1. Lineary and non-lineraly separable data (resp. a and b, from left to right). The former can be separated by an SVM. The latter can be made linearly separable by means of kernels. The two kinds of shapes (circle, star) symbolize two classes of data.

to a problem of pattern recognition. The common way to solve such problem is to use machine learning [10] and this has motivated our approach. In this work, the output of the machine learning fabric is a binary value that corresponds to whether or not vectorization would be profitable. This is a classification problem in which we have to discriminate between two classes (profitable and non profitable). Support vector machine (SVM) is a natural approach to solve such a problem [10] and in consequence we focus on it.

## 3. Our Approach

We give in section 3.1 a short introduction on support vector machines (SVM). We will then explain how we use SVMs in section 3.2 in order to predict the profitability of vectorization for both Exp1 and Exp2 (respectively after and before transformation). Readers familiar with SVM may directly skip to section 3.2.

### *3.1. Introduction to Support Vector Machines (SVM)*

This section informally describes support vector machine (SVM) in general, and how it can be used to solve non-linear binary classification problems.

#### *3.1.1. Informal Definition of SVM*

An SVM is a type of linear classifier called *maximum margin classifier*. It is a decision machine, that is, it outputs a single binary value which represents one of two classes. This contrasts with other machine learning algorithms, such as neural networks, that output a continuous value that corresponds to the probability for the input to satisfy a given property. SVM can also be combined to perform multiclass classification, meaning the prediction of more than two classes.

In an n-dimension space, let us consider a set of data that may be of two classes, *0* or *1*. An SVM is a function that takes into input the coordinates of the data in the space and outputs its class:

$$SVM : \mathbb{R}^n \longmapsto \{0, 1\} \tag{1}$$

To do so, it simply determines the position of the data relatively to an hyperplane. The equation of the hyperplane is determined by solving an optimization problem on its coefficients using *training data*: this is *the training phase*. During the training phase the solver endeavors to maximize the margin between the training data of both classes[4]. This situation is illustrated in figure 1 *a* for two dimensions. Each dimension is called a feature and is a number that *describes* the input. Typical SVM problems use more than two features. For example, we utilize 12 features

[4]A rigorous definition of the margin can be found in Christopher M. Bishop's reference book [10]

in our experiments (please refer to sections 3.2 and 5). The choice of features is very important and constitutes one of the main challenges of using SVM. On the figure, the reader should further notice the presence of a circle in the middle of the stars; this does not prevent the SVM from finding an acceptable (that is, highly likely) frontier between both classes: SVM are said to be Bayesian classifiers.

The function defined by a trained SVM is called the *model*. The quality of the model is its propensity to successfully predict the class of new inputs after training. It is often assessed using its accuracy on data different from the training data:

$$accuracy(model) = \frac{correct\ predictions}{number\ of\ data\ samples} \tag{2}$$

When using SVM, we prepare a set of data vectors which include both the features and the expected class (that is, both the inputs and the output of the SVM). We call it the *examples dataset*. A part of the set is used for training, the remaining for testing to compute the accuracy. We call these sets the *training set* and the *test set*, respectively. To compute the accuracy, we use a technique called leave-one-out cross-validation (LOOCV). It consists of predicting with only one data in the test set while the others are used for training, and to repeat this process for each data vector. In this way the model's accuracy can be determined while training with the maximum number of examples.

We further graphically assess the quality of our model by means of a graph called the *learning curve*. Two examples of learning curves are given in figure 5. It shows the evolution of the accuracy of the model for a growing number of example data, for several different choices of training and test sets. We train the SVM with the training set and compute the accuracy for both the training and the test sets. For each horizontal value, we consider 10 randomly generated sets. The lines show the average of the accuracies related to both sets in order to show their respective tendencies. The line is expected to raise for the test set as the size of the datasets increases: this is because the SVM is more trained and therefore smarter. On the other hand, the accuracy for the training set should drop if the SVM does not overfit the data. Finally, both lines are expected to converge toward the same value: this is the *asymptotic accuracy* of our model.

### *3.1.2. About Non-Linearly Separable Data*

An SVM can discriminate between two classes if they are linearly separable. However, this is often not the case. In order to deal with non-linear problems, the common approach consists in transforming the features in order to make them fit this constraint. The transformation functions are called *kernels*. For instance, a kernel might be able to transform the data of figure 1 *b* into the one of figure 1 *a*. Some kernels have already been proposed by researchers specialized in the field. They transform the data by changing not only the meaning of the features, but also their number. However, the understanding of the underlying mathematics is out of the scope of this paper. Our data, presented in section 4, are not linearly separable and hence we make use such type of kernel.

The kernel type depends on the problem and should be decided case by case. In addition, these kernels often accept tuning parameters whose choice is again up the user. Fortunately, the literature provides some choices to start with. We rely on the approach suggested by Bishop [10]. In particular, we use the radial basis kernel and try for its parameters the values 1 and 3, multiplied by powers of ten from $10^{-4}$ to $10^{3}$ in order for the model's accuracy to be maximized.

## *3.2. Using Support Vector Machines*

As mentioned in section 1, we leverage SVM in order to determine the profitability of vectorization when only considering ABBV. We define that the output *0* corresponds to a program for which vectors are not profitable, and *1* otherwise. We consider two situations for which we want to determine the profitability of vectorization (Exp1 and Exp2 mentioned in section 1). The inputs and outputs of both experiments are shown in figure 2. In both cases, the output of the SVM is the same; they however differ from their inputs. Exp1 only accepts as feature the transformed input program, while Exp2 also accepts the parameters of the transformation to apply. In this paper we only consider loop unrolling to transform the program, hence the later boils down to the unroll factor.

In this work we select a set of 12 static, hardware-independent features to characterize the input programs (section 5). Our experimental flow to extract these features is shown in figure 3 (see section 4.1.1).

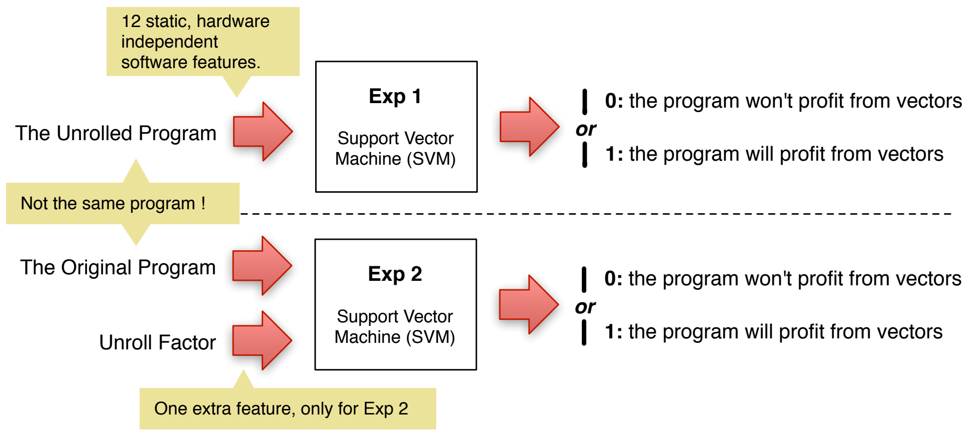
12 static, hardware independent software features.
The Unrolled Program
Exp 1
Support Vector Machine (SVM)
0: the program won't profit from vectors
or
1: the program will profit from vectors
Not the same program !
The Original Program
Unroll Factor
Exp 2
Support Vector Machine (SVM)
0: the program won't profit from vectors
or
1: the program will profit from vectors
One extra feature, only for Exp 2

Fig. 2. The inputs and outputs of the SVM used in both Exp1 and Exp2. They only differ by one extra input feature in the case of Exp2: the unroll factor to apply on the program.

Table 1. The options fed to Intel Compilers. We use SSE3 as supported by our test machine (Intel Core2Duo Extreme Merom@2.66GHz).

| | |
|---|---|
| With ABBV | -std=c99 -O3 -fno-alias -opt-report 3 -V -vec -vec-report5 -xSSE3 |
| Without ABBV | -std=c99 -O3 -fno-alias -no-vec |

## 4. Experiments

We present our experimental setup in sections 4.1, 4.1.1 and 4.2, before assessing our results in section 4.3. The software characteristics are discussed later on in section 5.

### *4.1. Benchmark and Scope*

Compiler optimizations are more efficient when applied to programs' hotspots, that is, innermost loop nests. That is why we consider a benchmark suite made of simple loops, representative of how an innermost calculation loop may look like. Our benchmark is called TSVC (which stands for test suite for vectorizing compiler), in its version provided by Maleki et al. [2]. It is composed of 151 simple loops and has been devised to assess the quality of compilers for vectorizing loops. In our work however, we do not consider loop vectorization; instead, we expect to mimic this transformation by expanding it into (1) loop unrolling and (2) ABBV. That is why we consider not only the 151 loops that constitute TSVC, but also the same with the innermost nest unrolled with a factor ranging from 2 to 20. The resulting benchmark counts 3020 programs ($151 \times 20$).

The programs are compiled into executables using Intel Compiler with and without ABBV, using the compilation options shown in table 1 (please refer to the experimental flow on figure 3). For Intel Compiler not to unroll, vectorize or pipeline loops, we annotate the innermost loops with the corresponding pragmas: `nounroll`, `novector` and `noswp`. The host machine is an Intel Core2Duo (Merom) at 2.66GHz, that supports up to SSE3 instruction set extension.

#### *4.1.1. Preparation of Data and Experimental Flow*

Our experimental flow is divided into two steps: the generation of known data and the training/validation of the SVM. The flow to generate the former is detailed in figure 3. It consists of 3 steps: (1) we prepare, compile and execute our benchmark with and without ABBV; (2) we determine the profitability of vectorization (3) we measure the software characteristics for the benchmark. The flow relies on three tools: an innermost loop unroller, a feature extractor, and the vendor compiler. For the first, we use a tool called PIPS[5]. For the third, we use the Intel Compiler (see section 4.2). For the second, we measure by means of custom tools the characteristics of programs

[5]Online: http://pips4u.org/

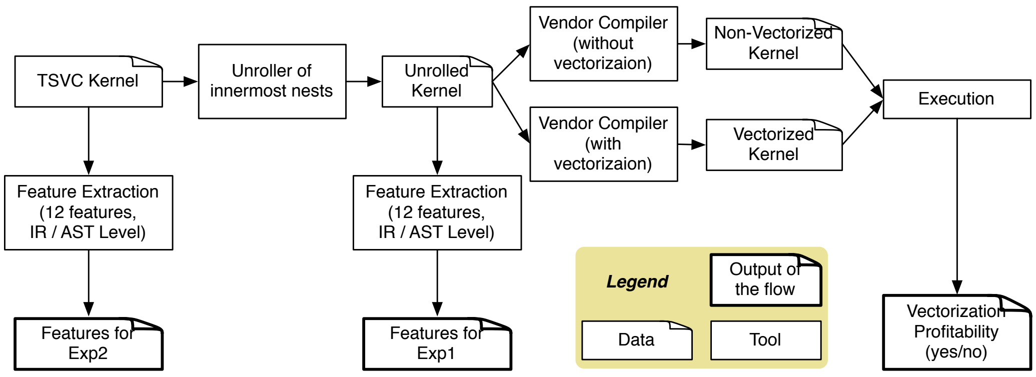
TSVC Kernel
Unroller of innermost nests
Unrolled Kernel
Vendor Compiler (without vectorizaion)
Vendor Compiler (with vectorizaion)
Non-Vectorized Kernel
Vectorized Kernel
Execution
Feature Extraction (12 features, IR / AST Level)
Feature Extraction (12 features, IR / AST Level)
Features for Exp2
Features for Exp1
Legend
Output of the flow
Data
Tool
Vectorization Profitability (yes/no)

Fig. 3. Our experimental flow to build our training and test sets. The outputs of the flow are fed to the SVM. The reader should notice that the features fed to Exp1 and Exp2 consists of the same items, but they differ from the timing of the flow they are extracted at (resp. before and after unrolled).

at two abstraction levels: LLVM's intermediate representation and Clang's abstract syntax trees[6]. We refer to both as **IR** and **AST** respectively.

After preparing the data, we use the SVM as explained in section 3.2: (1) we divide the examples dataset into the training set and the test set (respectively 80% and 20% of the initial set); (2) we train the SVM with the training set; (3) we assess the quality of our model by computing the LOOCV accuracy and by plotting the learning curve. To implement these experiments, we use libsvm[7], a stable and free library for SVM.

## *4.2. Our Baseline*

We compare our results against two baselines. The first one is Intel Compiler alone (see details below). The second one is the exact same experiments as the ones described in section 4.1.1, but using nearest neighbor (NN) instead of SVM. NN is a simple machine learning technique that considers for prediction the vectorization profitability of the closest training data with respect to the Euclidian distance in the space of features.

We compare against Intel compiler alone because it is considered by the community as one of the brightest available vendor compiler. With respect to P1, it has a success rate of 56%. This number is the ratio of profitable vectorization among the examples dataset used in section 4.3; we consider all the kernels and all the unroll factors, and Intel Compiler has unrolling, loop pipelining and loop vectorization off.

With respect to P2, it is impossible to reproduce Exp2 with Intel Compiler. Instead, we consider its success ratio to profitably generate SIMD instructions when it is possible. We consider the 151 non-unrolled programs, and we activate unrolling in Intel Compiler (by removing the `nounroll` pragmas). By doing so, we ask Intel Compiler to figure out by itself a correct transformation to apply (including unrolling) in order to generate profitable vectors. It succeeds for 13 programs. However, as explained in section 4.3, it is possible to find an unroll factor that enables profitable vectorization for 63 programs. Therefore, its success rate is 20.63% (13/63).

## *4.3. Experimental Results*

First of all, let us take a look at figure 4 *a*. It plots the number of times that vectorization was profitable for each benchmark. This graph confirms that our data are not skewed. Further analyses show that the ratio of the execution time without vectors and the same with vectors ranges from 0.46 to 39.25. In other words, ABBV may provide up to 39.25 time speedups, but failing to recognize non-profitable ABBV may slow down the compiled program up to 2.2 times ($0.46^{-1} = 2.2$). The latter occurs for the program *s116* with an unroll factor of 2.

We plot the accuracy of our method against the baselines in figure 4 *b*. We use LOOCV as explained in section 3.1. The prediction accuracy for Exp1 by using SVM is almost 70%. This is significantly better than the 56% of

[6]Online: respectively http://www.llvm.org and http://clang.llvm.org
[7]Online: http://www.csie.ntu.edu.tw/ cjlin/libsvm/

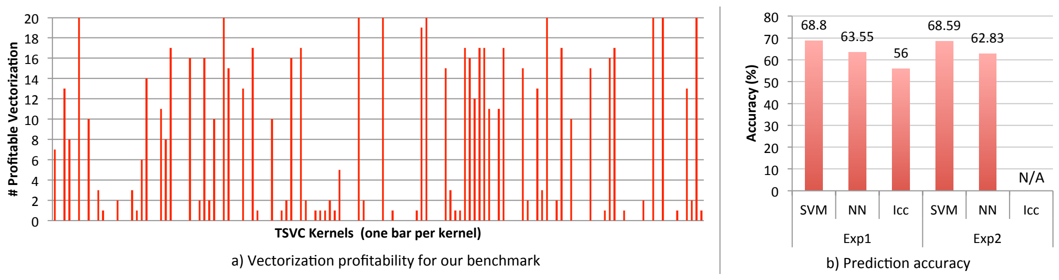

Fig. 4. a) Number of times the vectorization was profitable for each program; b) Our prediction accuracy

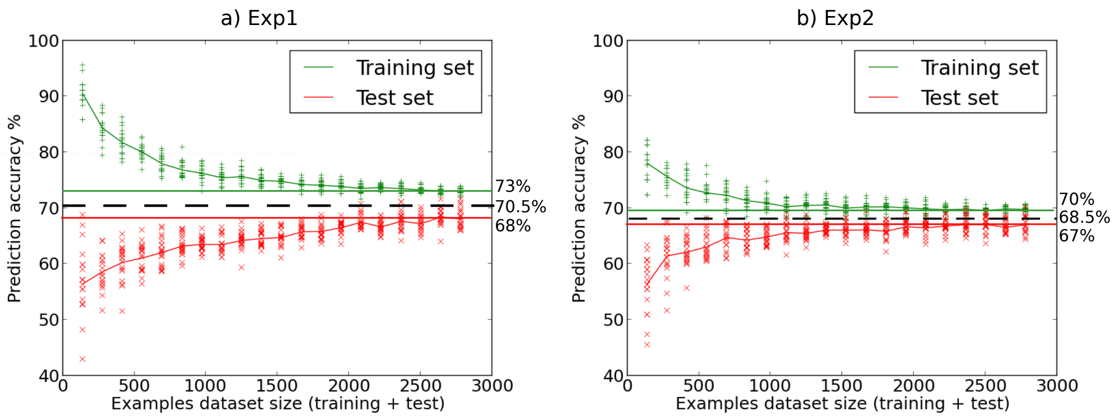

Fig. 5. Learning curves for Exp1 and Exp2 (resp. a and b, from left to right).

Intel Compiler (labeled *icc* in the plot). In particular, the SVM manages to correctly decide against vectorizing for the program *s116* with an unroll factor of 2, thereby providing a speedup of 2.2. Moreover, we achieve a similar, high precision accuracy in the case of Exp2. This is remarkable because we solve in Exp2 a problem more complex than in Exp1. No number is given for Intel Compiler for the reasons explained in section 4.2. Still, we can compute the success of SVM at finding at least one profitable vectorization for each program when possible: it is far above Intel Compiler at 73.01% (against 20.63%).

Figure 4 *b* further plots the prediction accuracy using NN: 63.55% and 62.83% respectively. These numbers are under those of SVM, but very close. This is coherent with the results published by Stephenson et al. [1]. Moreover, NN has the advantage over SVM of being a simpler algorithm. Therefore we believe there may be situations where it is preferable over SVM. Still, SVM achieves more than 5% better accuracy and it is the most successful technique in our experiments.

Figure 5 plots the learning curves for both Exp1 and Exp2 (from left to right). Please refer to section 3.2 for more information on how to interpret these graphs. The profile of the curve is ideal, as the accuracy for the testing and the training data are actually converging towards 70.5% and 68.5%, respectively. The maximum average accuracies measured for the test sets are respectively 68% and 67%[8]. These learning curves illustrate the convergence of our model and its high accuracy, that is, its high quality. In other words, we have shown that SVM can be successfully used to predict the profitability of vectorization on TSVC, and can be useful to address both P1 and P2.

## 5. About Software Characteristics

In this section, we present the software characteristics we used to model the benchmark to feed the SVM. There are several ways to classify the types of software characteristics. First, they may be measured statically

[8]These accuracy numbers are different (yet very close) from the ones computed using LOOCV, because the method to obtain them is different.

from sources, or dynamically at runtime. While the second makes it possible to gather far more information, it requires to actually execute the program: this is not something we can afford inside a compiler for obvious time-related constraints. Software characteristics may further be hardware dependent and hardware independent. The former suffers from lack of portability: we may rely on some hardware counters on a given machine that is not available on another one, for example because of differences in micro-architecture. For this reason we favor the latter, and our results in section 4.3 show that this is enough. In conclusion, we consider only *static*, *hardware independent* software characteristics.

We choose the characteristics empirically, while observing the behavior of Intel Compiler with our benchmark. At the end we have retained 12 of them; 6 of which are extracted at AST level, and 6 at IR level (see section 4.1.1). These characteristics constitute features of the SVM, together with the unroll factor for Exp2. They are detailed one by one in table 2, and plotted by pairs in figure 6. When more than one array is accessed in the innermost loop, AST2 and AST3 are measured for each arrays access, then we consider as a software characteristics their arithmetic mean (a real number between 0 and 1). IR2 and IR4 rely on the prediction of the dynamic behavior of our program provided by LLVM. This is convenient as it uses some placeholder when the values can not be computed. IR6 should be understood as a rough estimation of the number of registers consumed by our loop. Finally, in order to determine AST1, we not only analyze the *for* statement, but also the array indexes for a consistent, constant multiplier of the induction variable.

In order to further assess the amount of redundancy in our set of features, we have run principal component analysis on the set of known data presented in section 4.3. The results (not detailed here) show that the amount of variance is loosely concentrated, and that it requires half of the components to express 80% of it: the information is relatively already well spread into our original set of features. In other words, all our features are relevant to help the SVM to carry out accurate predictions.

## 6. Related Work: Machine Learning and Compilation

Using machine learning to improve compilers is no breakthrough, and the literature already contains several works that apply support vector machine (SVM) [1, 5], nearest neighbor (NN) [1, 13], artificial neural networks (ANN) [6], and logistic regression [12]. They however make different types of predictions: Stephenson et al [1], Park et al. [5] and Agakov et al. [13] determine parameters of code optimizations; Kulkarni et al. [6] order optimization passes in the middle end; Pekhimenko et al. [12] use machine learning to focus on search algorithms.

The work from Stephenson et al. [1] is the closest from ours. It uses multiclass classification[9] to determine for each program the unroll factor that yields the best performance. Their benchmark consists of 2500 loops extracted from several well-known benchmarks targeted at high-performance computing as well as embedded computing. They leverage both SVM and NN, and manage to correctly predict the best unroll factor with an accuracy of 65% and 62%, respectively. This proves to be far better than their baseline, Open Research Compiler[10]. These numbers are very close from the ones we obtain in section 4.3. The reason may be because they focus on the unroll factor as we do, and because they also consider only static software characteristics. Our work is still different from this one for two reasons. First the set of characteristics we consider is smaller than, and not included into theirs (see section 5). Second, we predict the profitability of vectorization instead of the best unroll factor. This has several fundamental consequences on the way we manipulate SVM. Kulkarni et al. [6] use machine learning to tackle the complex problem of the ordering of optimization techniques. They model an optimization scenario using a Markov process; then they construct optimization scenarios iteratively, one optimization technique at a time, using ANN at each step. They apply their method to the just-in-time compiler of a Java virtual machine, and achieve to reduce the execution time of compiled programs by up to 20%.

Our work, as well as all the previously detailed ones, endeavors to improve the quality of the output of the compiler by reducing the execution time of the compiled programs. From this point of view, the works from Agakov et al. [13] and Pekhimenko et al. [12] are original. They utilize machine learning in order to reduce the compilation time, that is, the execution time of the compiler itself. The objective of Agakov et al. is to reduce the

[9]As opposed to binary classification
[10]Now called Open64. Online: http://www.open64.net/

time required to find the best optimization sequence to apply to a given program. They adopt a technique based on NN to bias an existing search algorithm (random or genetic) and they manage to reduce the search time by one order of magnitude. On the other hand, Pekhimenko et al. [12] use logistic regression to determine the parameters of optimization techniques inside a fixed optimization scenario, with the aim of leveraging the fast execution time of logistic regression compared to the heuristics implemented into a commercial vendor compiler. They manage to reduce the compilation time by two orders of magnitude while at the same time slightly improving the execution time. The originality of our work lies on its target: we aim at predicting the profitability of vectorization. This is the first work to do so to best of our knowledge. It is therefore complementary to the related work, and can be use side-by-side in order to improve traditional compilers that do not use any machine learning.

## 7. Concluding Discussion

In this paper, we use SVM to predict the profitability of automatic basic-block vectorization for Intel Compiler on TSVC, a benchmark made of 151 simple yet representative loops, unrolled by a factor ranging from 1 to 20. We achieve a prediction accuracy of about 70%, even before actually unrolling the loops. This technique may be useful in compilers in two ways. First, it can enable the compiler to avoid generating vectors that actually slow down the program; this often happens with the Intel Compiler (44% of our benchmarks). Second, it can allow the compiler to better predict the existence of profitable unroll factors in the middle-end, as Intel Compiler fails to do in 79.37% of the programs of our benchmark. This is important because depending on the unroll factor, vectorization may provide up to 39 times speedup according to our measurements (benchmark *vpvts* with an unroll factor of 20).

We however see several limitations in our approach. First, support vector machines require fine tuning especially with respect to the kernels, and wrong assumptions may easily lead to poor predictions. Second, the choice of the set of software characteristics is critical and depends on the problem we are trying to solve. If we don't have enough features or if they are not representative enough, we are likely to obtain low accuracy; the same stands if we have too many software characteristics because the more the features to feed to the SVM, the more the number of training examples needed. This leads us to the main bottleneck of our approach: we need for training many benchmark programs, although they are often hard to find.

Still, our results show that machine learning makes it possible to significantly improve the quality of the code generated by Intel Compiler: we have measured speedups up to 2.2 times by successfully preventing it to carry out unprofitable vectorization. Some challenges remain, but we are eager to tackle them in the near future.

## References

[1] M. Stephenson and S. Amarasinghe: Predicting unroll factors using supervised classification, in *International symposium on code generation and optimization*, pp. 123-134, March 2005

[2] S. Maleki et al.: An evaluation of vectorizing compilers, *Parallel Architecture and Compilation Techniques (PACT)*, pp. 372-382, October 2011

[3] L. Van Ertvelde and L. Eeckhout, Benchmark synthesis for architecture and compiler exploration, *International Symposium on Workload Characterization*, pp. 1-11, December 2010

[4] S. Kamil and A. Fox, Bringing parallel performance to Python with domain-specific selective embedded just-in-time specialization, *10th Python for Scientific Computing Conference*, 2011

[5] E. Park et al., Predictive modeling in a polyhedral optimization space, *International Symposium on Code Generation and Optimization (CGO)*, pp. 119-129, April 2011

[6] S. Kulkarni and J. Cavazos, Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning, *OOPSLA*, 2012

[7] M.-W. Benabderrahmane et al., The polyhedral model is more widely applicable than you think, *ETAPS International Conference on Compiler Construction (CC'2010)*, pp. 283-303, March 2010

[8] K. Hoste and L. Eeckhout, Microarchitecture-independant workload characterization, *MICRO*, pp. 63-72, May / June 2007

[9] K. Hoste, Analysis, Estimation and Optimization of Computer System Performance Using Machine Learning, PhD Thesis, Ghent University, September 2010

[10] C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006

[11] R, Allen and K. Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann, 2002

[12] G, Pekhimenko and A. D. Brown, Efficient program compilation through machine learning techniques, *International Workshop on Automatic Performance Tuning (iWAPT)*, October 2009

[13] F. Agakov et al., Using machine learning to focus iterative optimization, *International Symposium on Code Generation and Optimization (CGO)*, pp. 295-305, March 2006

Table 2. The software characteristics we consider

| Identifier | Level | Range | Description |
|---|---|---|---|
| AST1 | AST | ℕ | The increment of the innermost `for` loop |
| AST2 | AST | $\{0, 1\}$ | Will the address of the first access to the array be always aligned with the machine's vector size ? |
| AST3 | AST | $\{0, 1\}$ | In array accesses, is the induction variables involved in the last dimension is the one of the innermost loop ? |
| AST4 | AST | ℕ | Number of array accesses in the body of the loop |
| AST5 | AST | ℕ | The number of arrays accessed inside the loop |
| AST6 | AST | $\{0, 1\}$ | Do the benchmark involve any restrict keyword ? |
| IR1 | IR | ℕ | The size of the dataset |
| IR2 | IR | ℕ | Estimation of the dynamic instruction count |
| IR3 | IR | ℕ | The depth of the innermost loop |
| IR4 | IR | ℕ | The estimated trip-count of the innermost loop |
| IR5 | IR | ℕ | Number of IR statements in the innermost loop |
| IR6 | IR | ℕ | The number of SSA variables used in the innermost loop |

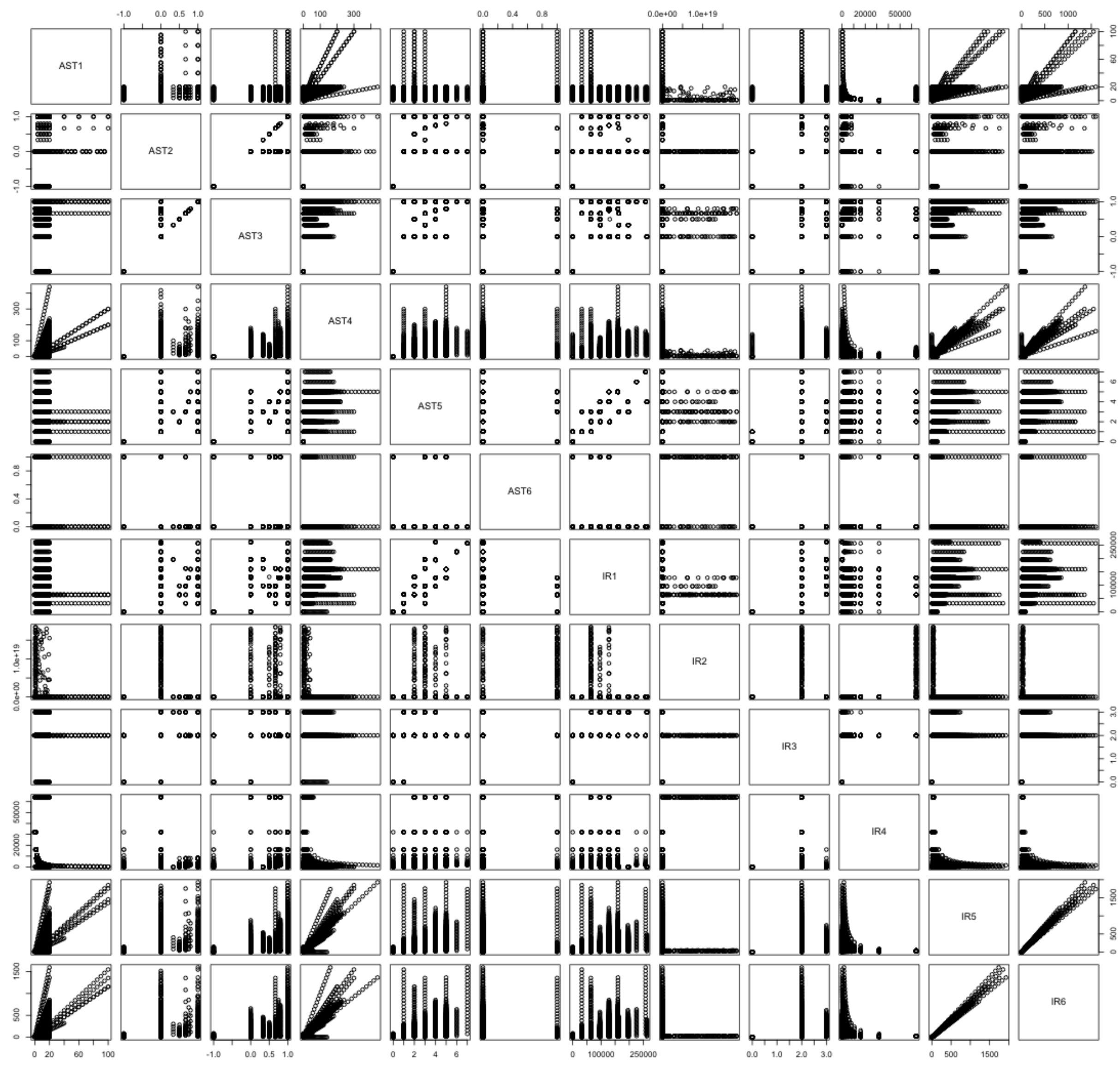

Fig. 6. Projection of the benchmarks in all 2D planes of characteristics (3020 dots on each graph)