Portable Cost Modeling for Auto-Vectorizers

Angela Pohl, Biagio Cosenza and Ben Juurlink
Embedded Systems Architecture
Technische Universität Berlin
Berlin, Germany
{angela.pohl, cosenza}@tu-berlin.de

***Abstract*—Compiler optimization passes employ cost models to determine if a code transformation will yield performance improvements. When this assessment is inaccurate, compilers apply transformations that are not beneficial, or refrain from applying ones that would have improved the code. We analyze the accuracy of the cost models used in LLVM's and GCC's vectorization passes for two different instruction set architectures. In general, speedup is over-estimated, resulting in mispredictions and a weak to medium correlation between predicted and actual performance gain. We therefore propose a novel cost model that is based on a code's intermediate representation with refined memory access pattern features. Using linear regression techniques, this platform independent model is fitted to an AVX2 and a NEON hardware. Results show that the fitted model significantly improves the correlation between predicted and measured speedup (AVX2: +52% for training data, +13% for validation data), as well as the number of mispredictions (NEON: -15 for training data, -12 for validation data) for more than 80 code patterns.**

I. Introduction

Optimizing compilers identify code transformations that improve a program in regard to a given goal, e.g. higher performance, lower energy consumption, or smaller memory usage. However, code transformations are not always beneficial and it is therefore important to understand if they may infer severe overheads. For this reason, modern compilers, such as GCC and LLVM, typically perform a profit analysis to determine whether a transformation is beneficial, i.e. yielding an improvement over to the code's baseline version.

An accurate profitability analysis is particularly important for vectorization. Auto-vectorizers work on either loops or basic blocks, trying to group together multiple instructions in order to replace them with a vectorial one. This process requires code transformations such as instruction replacement or code re-writing, and it can introduce expensive overheads, e.g. because of complex memory access patterns or vector shuffling.

State-of-the-art compilers use a cost model to understand whether applying vectorization is beneficial. However, such cost models are relatively simple: the cost is determined on individual instruction level, and the cost of a transformed block is the sum of all of its individual instruction costs (Section III explains these algorithms in detail). We have assessed the accuracy of the vectorization cost models of GCC and LLVM on the TSVC benchmark [1], on an Intel Xeon E5-2697 with AVX2 SIMD hardware extensions and on an ARMv8 CortexA-53 with a NEON SIMD unit. Experimental results show that: (a) existing vectorization cost models are only weakly or moderately correlated with the actual cost; (b) there are mispredicted codes where the error in cost modeling results in wrong choices, i.e. either in vectorization with slowdowns (*false positives*) or cases where vectorization would have been beneficial but it is not applied (*false negatives*); (c) mispredictions have an impact on the final performance in terms of execution time. Therefore, the heuristics used by today's production compilers are not sufficiently accurate.

The design of an accurate vectorization cost model is challenging for several reasons. First, an accurate cost model should consider those code features (e.g. instruction patterns) that impact the performance of the vectorized code. For example, it should explicitly distinguish whether memory accesses are interleaved, reversed or scalarized, as these patterns have a different impact on the speedup of the generated code. Secondly, typical vectorization benchmarks are not build for accurate cost modeling: while they focus on diverse vectorization cases, they do not cover a variety of *cost modeling patterns*. For example, the whole TSVC benchmark (151 loops) contains only two reversed loops. Finally, approaches should be portable between the different SIMD instruction set architectures (ISAs).

Based on these insights, we propose a novel modeling methodology that increases the accuracy of auto-vectorizers' cost models. Our approach models the cost using a machine learning technique with accurate code feature representation, fitted on speedup, and using extended training data. It does not depend on a specific SIMD instruction set and can be easily ported to any target hardware. The resulting cost model can be implemented as a pluggable extension to the LLVM cost model, and can be used by all of the compiler's auto-vectorizers.

We make the following contributions:

- An analysis of the accuracy of the cost models of GCC and LLVM's auto-vectorizers perfomed on the TSVC benchmark on two different SIMD ISAs (AVX2 and NEON), which shows the correlation between predicted and actual speedup, the number of mispredictions, and their impact on performance.
- A new portable cost model which improves state-of-the-art auto-vectorizers on both AVX2 and NEON in terms of cost prediction correlation, number of mispredictions and execution time. The proposed model predicts the speedup of a vectorized code based on an accurate code

feature representation, carefully tuned regression analysis, and an extended training data. Results are cross-validated on TSVC and selected Polybench loops, and fitted with different fitting techniques.
- An accurate feature analysis characterization based on both error- and model-based techniques that highlights the portability of the model by showing how different target-dependent code features are exploited on multiple SIMD ISAs.

The paper is organized as follows: Section II provides an overview of related work in the area of cost modeling in compilers. Existing cost models and their experimental assessment are described, respectively, in Section III and IV. Section V describes the different components of our proposed model. Experimental results showing cost model fitting, feature analysis, and improvement on cost prediction correlation, misprediction and performance impact, are presented in Section VI. The paper concludes in Section VII.

II. Related Work

Automatic vectorization has been extensively studied in literature [2], [3], and multiple techniques have been proposed that exploit vectorial parallelism either at loop level (LLV) or on straight-line code (SLP). This section focuses on related work that investigates the cost modeling of those techniques, rather than the proposed vectorization algorithms.

The ability to decide if vectorization is profitable is an important part of modern optimizing compilers. Code transformation techniques such as loop distribution and interchange [3] or if-conversion [4] can positively impact the profitability of vectorization, and it is therefore critical to provide an accurate cost model that correctly predicts whether overheads overcome the benefit of vectorization. Wu et al. [5] recognized the importance of correctly deciding when SIMDization is profitable in the XL compiler. Yuanyuan and Rongcai [6] have proposed an analytical cost model for the Open64 compiler, which, however, shows many cases where it cannot evaluate the right cost. Nuzman et al. [7] proposed a cost model for vectorization of strided-accesses; however, it does not consider other overheads or patterns.

Polyhedral compilers often include loop vectorization as part of a broader loop optimization framework. Bondhugula et al. [8] applied inner-loop vectorization after a tiling heuristic and selected the inner loops interchange transformation that is vectorizable; however, their method does not consider vectorization overheads. A polyhedral vectorization cost-model has been introduced by Trifunovic et al. [9]: their approach focuses on scheduling metrics, but does not cover code generation dependent metrics exploited in this work.

Machine learning models have gained interest in the compiler community and have been used to define vectorization cost model as well. Stock et al. [10] introduced a machine learning approach to improve automatic vectorization of tensor contraction kernels and stencil computations. Their cost model assists the generation of vectorized code by selecting the one with the best performance, after applying permutation and unroll-and-jam. It operates on assembly code and is not portable to non-Intel architecture; instead, our model is based on features extracted from LLVM's bitcode, and its portability is shown on both Intel AVX2 and ARM NEOM ISAs. Park et al. [11], [12] used a model based on logistic regression and support vector machine to narrow the set of candidate polyhedral loop optimizations, including vectorization; their approach is based on iterative search and, in contrast with our fully static approach, requires to execute the transformed variants on the target machine. Trouvé et al. [13] formulated vectorization profitability as a classification problem, predicted using a support vector machine (SVM). They also used a similar classification model to predict a compiler's command-line options that choose the most profitable vectorization in tensor contraction kernels [14]. Their SVM model is based on only twelve features (six extracted from the abstract syntax tree, six from the intermediate representation), resulting in a high number of mispredictions. In contrast, our approach defines 72 code features, including those that distinguish different memory access patterns.

Cost modeling is also critical in the context of straight-line code vectorization. Two examples are: realignment and data-reuse considered together with loop unrolling [15], and Throttled SLP (TSLP) [16], a SLP model that forces vectorization to stop earlier whenever this is beneficial, therefore overcoming the limits of standard greedy algorithms.

III. Cost Modeling in Auto-Vectorizers

In this section, we provide an overview of the currently implemented cost models in LLVM's and GCC's auto-vectorizers.

A. Cost Modeling in LLVM

The LLVM compiler applies multiple vectorization passes to the code, i.e. Loop Level Vectorization (LLV) and Superword Level Parallelism (SLP); the optional Basic Block (BB) vectorizer has been deprecated in the latest compiler versions. Both of the active passes use a similar approach to assess the cost of a vectorization. They determine the block cost of the transformed loop body or basic block (BB) and compare it to the scalar block cost. For this purpose, a cost is assigned to each instruction, based on the instruction type, the underlying hardware platform and the vectorization factor (VF). The vectorization factor denotes the maximum number of elements that fit into one vector, e.g. VF = 8 for single precision floating point numbers and a vector width of 256 bit. In the compiler, there are lookup tables for a variety of instruction set architectures and SIMD extensions defining these individual instruction costs. This block cost analysis is then performed for all possible vectorization factors. Since the same vectorization factor is applied to all instructions in one BB, the maximum possible vectorization factor is derived from the largest data type loaded/stored in the BB. Afterwards the minimum cost is chosen. If this minimum is the scalar block cost, no vectorization is applied, although other optimization techniques, such as unroll-and-jam, might be performed. The

$c_{min} = c_{scalar}$
$VF_{min} = 1$
for all Vectorization Factors **do**
  for all BBs in Loop **do**
    for all Instructions in BB **do**
      $c_{bb} += getInstrCost(Instr, VF)$
    end for
    $c_{vec} += c_{bb}$
  end for
  if $(c_{vec} < c_{min})$ **then**
    $c_{min} = c_{vec}$
    $VF_{min} = VF$
  end if
end for
return c_{min}, VF_{min}

Fig. 1. Pseudo-Code of LLV's cost calculation in LLVM

complete algorithm is shown in pseudo-code in Figure 1. To take overhead inferred by vectorization into account, a loop trip count threshold is added to avoid vectorization of "tiny" loops (trip count ¡ 16). For such tiny loops, vectorization is allowed only if no overhead is added outside of the loop.

Despite both passes using the same underlying lookup tables, their cost estimation varies due to slight differences in their respective lookup functions. For example, one pass assigns individual costs to the `getelementptr` instruction, while another merges this cost with the `load` /`store` instructions' costs. In addition, all passes use a different baseline, i.e. scalar block cost, to assess the transformation benefit. The results of the passes' cost analysis therefore cannot be compared. It is also possible that one pass deems a transformation beneficial, while another may not. An analysis which of these slightly varying cost models is more accurate, has not been performed yet.

B. Cost Modeling in GCC

The GCC vectorizer combines SLP and LLV vectorization in one compiler pass [17]. This pass utilizes a similar approach to cost modeling as described for LLVM. It also determines a BB's cost based on its individual instruction costs and the vectorization factor. However, it also accounts for vectorization overhead outside of a loop body. As the overhead, such as a scalar loop tail, becomes less significant in terms of cost with increasing loop iterations, the cost model tries to solve the following inequation to determine the minimum number of profitable loop iterations n:

$$n \cdot c_{scalar} + c_{s,out} > (n - n_{out}) \cdot \frac{c_{vec}}{VF} + c_{v,out}$$

When a number of loop iterations n can be found where the cost of the vectorized code c_{vec} and the overhead outside of the loop $c_{v,out}$ is less than the scalar cost c_{scalar} and the scalar overhead $c_{s,out}$, the loop is vectorized. In addition, a runtime check is added to avoid execution when the number of loop iteration is smaller than n. This has the side effect that vectorization is possible even for small iteration counts. If the inequation cannot be solved, the loop is deemed to be unvectorizable. The underlying cost prediction thus impacts the decision to vectorize a loop, as well as the minimum number of profitable iterations.

IV. Baseline Accuracy Analysis

The lookup tables used to determine cost in LLVM and GCC are based on latency and/or throughput numbers of individual instructions. However, cost is considered an abstract value in a sense that it does not translate into code performance directly, but must be interpreted relative to other cost values. The accuracy of these cost relations, i.e. the predicted speedup, has not yet been studied.

A. Setup

In this analysis, we compared speedups estimated by the compilers with actual measured speedups of the TSVC benchmark [1]. The benchmark consists of 151 loop patterns that test different vectorization challenges, such as dependence testing, statement reordering, or control flow. Contrary to other popular benchmarks, the TSVC kernels typically incorporate only one loop or one set of nested loops. This allows us to attribute a kernel's speedup directly to the speedup of its innermost loop, without further code instrumentation or annotation. To get accurate measurements of the vectorization only, further loop optimizations, such as interleaving and automatic unrolling, were disabled. The first test hardware is an Intel E5-2697 processor with AVX2 extensions, which corresponds to a vectorization factor of 8 for single precision floating point calculations. The second hardware is an ARMv8 CortexA-53 with 128-bit NEON extensions, which corresponds to a vectorization factor of 4 for single precision floating point numbers. For compilation, we used Clang/LLVM 6.0 and GCC 8.2.0, and build three different code versions:

- **scalar**: all optimizations are turned on, except for the vectorizers
- **vectorized**: all optimizations are turned on, including the vectorizers
- **forced vectorization**: all optimizations are turned on, inlcluding the vectorizers. Furthermore, the cost model is either set to unlimited (GCC) or all instruction costs are forced to 1 (LLVM).

The vectorized code includes all loops where the compiler deemed the vectorization to be beneficial. This incorporates patterns that do not exhibit any speedups or even slowdowns, so called *false positives* ($f\oplus$).

The forced vectorization adds patterns where the compiler previously did not apply vectorization due to the cost model predicting no benefit. This includes patterns that would have shown a speedup, so called *false negatives* ($f\ominus$).

Running the testbench provides the measured speedup S_{meas} for each loop kernel by calculating

$$S_{meas} = \frac{t_{scalar}}{t_{vec}},$$

where t_{vec} can be the result of regular or forced vectorization. To account for measurement inaccuracies, we imposed a 5%

AVX2

	LLVM	GCC
Set Size	85	65
ρ	0.58	0.33
L^2_{avg}	0.28	0.48
L^2_{max}	4.58	6.30
$f\oplus$	4	2
$f\ominus$	9	0
t_{scl}	85.00	65.00
t_{vec}	53.53	33.16
t_{opt}	51.79	32.53

NEON

	LLVM	GCC
Set Size	71	66
ρ	0.75	0.48
L^2_{avg}	0.26	0.21
L^2_{max}	4.41	3.38
$f\oplus$	0	0
$f\ominus$	17	2
t_{scl}	71.00	66.00
t_{vec}	40.34	31.83
t_{opt}	36.54	31.03

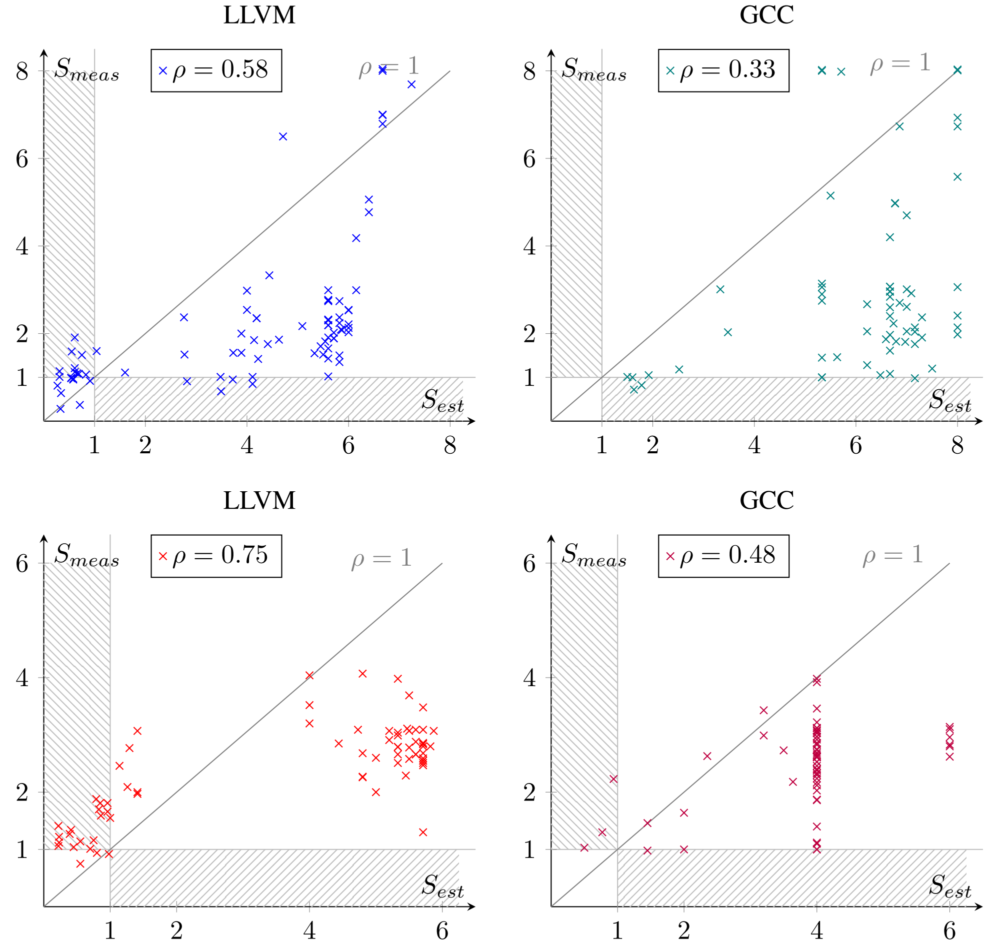

Fig. 2. Analysis results for LLVM and GCC loop level vectorization passes for the TSVC benchmark; shaded areas mark false positive and false negative predictions, straight line marks perfect positive correlation of $\rho = 1$

threshold, i.e. kernel slowdowns are classified as $S_{meas} < 0.95$, while kernel speedups are classified $S_{meas} > 1.05$

To obtain the speedup estimated in the compiler S_{est}, we analyzed the vectorization reports. The detailed reports provide the scalar loop body cost c_{scalar}, as well as the vectorized loop body cost c_{vec}. The predicted speedup can thus be derived as

$$S_{est} = \frac{c_{scalar}}{c_{vec}}$$

As described in the previous section, GCC also accounts for outside loop costs, i.e. prologue and epilogue cost, that have to be added to the loop body cost. This applies to scalar and vectorized loops. For large iteration counts, however, the cost calculation converges to the formula above, which is the case for the TSVC benchmark.

With the estimated and measured speedup, it is now possible to determine a correlation between the two quantities. Ideally, $S_{est} = S_{meas}$, which corresponds to a perfect linear correlation of $\rho = 1$ for the complete dataset.

For our results, we determined the estimated and measured speedups for LLVM's LLV pass, as well as GCC's vectorizer. We omitted LLVM's SLP pass due to the loop based kernels in our benchmark, which are not suitable for SLP vectorization. In fact, only three kernels out of the 151 are vectorizable with SLP by both compilers. We then removed those kernels from the analysis where the cost model was not used. This applies to codes where optimization techniques such as pattern substitution or reductions are applied; in these cases, vectorization is always deemed beneficial and no further assessments are performed. After further removing identical kernels, the evaluated dataset consisted of 85 kernels for LLVM and 65 kernels for GCC on the AVX2 platform, as well as 71 kernels for LLVM and 66 kernels for GCC on the NEON platform.

B. Results

The results for the analyzed kernels are displayed in Figure 2. In these scatter plots, each plot point corresponds to one of TSVC's analyzed kernels. Shaded areas either mark false positives ($f\oplus : S_{est} > 1, S_{meas} < 0.95$) or false negatives ($f\ominus : S_{est} < 1, S_{meas} > 1.05$), while the straight line indicates the perfect positive correlation of $\rho = 1$.

Both compilers tend to overestimate the speedup gain. On the AVX2 platform, this results in moderate-to-weak correlations of $\rho = 0.58$ (LLVM) and $\rho = 0.33$ (GCC). LLVM estimates a speedup of around 6x for a large number of kernels,

while GCC estimates speedups to range between 6x-8x. The measured speedups, on the other hand, typically range between 1x-3x for both compilers.

On the NEON platform, correlations are higher with $\rho = 0.75$ for LLVM and $\rho = 0.48$ for GCC. As can be seen in the plots, both compilers show distinct clusters in their respective performance prediction, resulting in a clear classification whether to vectorize or not. This is due to the assumption that the majority of kernels scales perfectly ($S_{est} = VF$) or even exhibits super-linear speedups ($S_{est} > VF$). However, measured speedups are in the same range as on the AVX2 platform, i.e. between 1x-3x.

For all vectorization passes, the over-estimations of speedup imply that there is no or little penalty added in the cost calculation for vectorization. As an example, LLVM tends to assume perfect scaling of memory operations, i.e. the load/store costs are the same for scalar and vectorized code. With vectorization, however, the memory bandwidth demand grows, including a kernel becoming memory bounded due to vectorization. Such side effects as in this example cannot be modeled with today's cost models, since they only analyze cost at instruction level, regardless of other code properties such as arithmetic intensity.

When determining the correlation factor, it must be noted that its Pearson's product-moment coefficient is sensitive to outliers, such as the significantly over-estimated speedups that fall into the false positive range for LLVM on the AVX2 platform. We therefore calculated the Euclidean distance as well by determining the L^2 norm of the vector difference. Due to the varying set sizes of vectorized kernels, we then normalized the distance per kernel to obtain a comparable value, i.e.

$$L^2_{avg} = \frac{||S_{meas} - S_{est}||}{setSize}$$

Our measurements show an average distance of $L^2_{avg} = 0.28$ for LLVM, while GCC exhibits a higher average distance with $L^2_{avg} = 0.48$ on the AVX2 platform. On the NEON hardware, average distances are $L^2_{avg} = 0.25$ for LLVM and $L^2_{avg} = 0.21$ for GCC. Looking at each kernel individually, we see that maximum distances L^2_{max} range between 4.58 - 6.30 for AVX2 and 3.38 - 4.41 on NEON. We classified all kernels by their respective L^2 distance values to quantify the distribution of the error. An overview of all results is presented in the tables in Figure 2.

In addition to the analysis of the mathematical correlation and the Euclidean distances, we investigated two other properties of the cost models: the number of false predictions and their impact on execution time. For the AVX2 platform, LLVM predicts 12 codes wrong ($f\oplus : 3, f\ominus : 9$), while GCC does not produce false negatives and exhibits 2 false positives ($f\oplus : 2, f\ominus : 0$). For all compilers, mispredictions are spread out evenly across TSVC's code patterns with one exceptions: on both hardwares, four of LLVM's $f\ominus$ predictions are test patterns with array indirections, i.e. code such as `a[i] = b[c[i]]`. Here, speedup is underestimated due to the scalarized memory loads and shuffle operations that are needed to create the vectors.

Besides the absolute number of mispredictions, it is also important to understand their *impact*, i.e. how much slowdown is inferred due to a false positive and how much speedup is lost due to a false negative. For this measurement, we evaluated the normalized execution time of the kernels before and after vectorization and compared it to the optimal execution time. Since all kernels have different run times, they are normalized to their scalar execution time. All scalar kernels consequently have an individual run time of one time unit, e.g. the total normalized scalar execution time $t_{scl} = 85.00$ for LLVM on the AVX2 hardware. The total vectorized execution time t_{vec} is calculated by adding the normalized vectorized execution times for all vectorized kernels and the normalized scalar execution times for all other kernels. For example, a kernel that exhibits a speedup of 4x after vectorization will contribute 0.25 time units, while a non-vectorized code will contribute its normalized scalar execution time of 1.0. The optimal execution time t_{opt} can be determined by adding the normalized execution times of a perfect vectorization, i.e. a vectorization without mispredictions.

Because of the low number of mispredictions for GCC, the difference between the vectorized and optimal execution time is limited for both hardware platforms and ranges between 0.63-0.80 time units. The difference is larger for LLVM due to the high number of mispredictions, however, and ranges between 1.74 - 3.8 time units. Especially on the NEON hardware, performance is lost due to the large number of $f\ominus$ kernels.

V. Improving the Modeling

Based on the analysis insights, we sought an improved method to create more accurate performance predictions. In this section, we describe the transition from modeling an abstract cost to predicting individual loop speedup, explain how the features for the new cost model were chosen and present enhancements to the existing TSVC benchmark to ensure sufficient feature coverage.

A. Targeting Speedup Instead of Cost

As described previously, a BB's vectorized cost c_{vec} is calculated as the sum of all its individual instruction costs c_i. This vectorized cost c_{vec} is then compared to the block cost of the scalar code to determine a code transformation's profitability. We have used this predicted speedup as the accuracy measure in the previous analysis section.

Based on this approach, we can model the predicted speedup directly, instead of the indirect method of determining individual instruction costs c_i from which the speedup will be derived. As a consequence, rather than calculating

$$S_{est} = \frac{c_{scalar}}{c_{vec}} = \frac{c_{scalar}}{\sum n_i c_i}$$

with n_i denoting the number of occurrences of a specific instruction type, we model a weight w_i that contributes to the predicted speedup

$$S_{est} = \sum n_i w_i$$

In this context, w_i can be positive, zero, or negative. A positive weight indicates that an instruction scales well with vectorization, while a negative weight indicates an added overhead, i.e. a slowdown.

As an additional refinement, we incorporated a metric for the *block composition* into our model. As of today, compilers look at each instruction cost c_i individually, regardless of the BB's other instructions. However, code characteristics such as the arithmetic intensity impact the maximum achieved speedup. By normalizing the individual instruction counts n_i to the total number of instructions in the BB, we account for different instruction mixes. The model thus becomes

$$S_{est} = \sum \frac{n_i}{\sum n} w_i$$

Our modeling approach has the advantage that it is no longer tied to a scalar baseline cost c_{scalar}, which can also introduce error. Especially with the value of a block cost being restricted by its integer data-type only, small relative errors can result in large absolute errors. As an example, $c_{vec} \in (1, 3876)$ and $c_{scalar} \in (0, 170, 068)$ in GCC for the TSVC benchmark on the AVX2 platform. Furthermore, confining our dependent variable, i.e. the target speedup S_{est}, to an interval of $(0, VF)$ will help in model fitting later.

An additional benefit of this approach is that it allows the comparison of different vectorization options. Since the performance estimation is no longer tied to a certain baseline, but predicts a block speedup, the results can be compared to other predictions. As a use case example, our cost model enables the comparison of LLV and SLP vectorization results to select the better option, since there are codes where SLP outperforms LLV in LLVM (e.g. kernel s128).

B. Feature Representation and Extraction

An important aspect of our modeling approach is keeping the model abstract and hardware agnostic. We therefore chose to use LLVM's Intermediate Representation (IR) as a baseline feature set. In its latest release, the LLVM IR instructions can be classified into five different categories: terminator instructions, binary instructions, bitwise binary instructions, memory instructions, and others. In total, there are 62 instruction types. To understand if this abstract code representation is sufficient for speedup modelling, we grouped together all TSVC code patterns that share the same representation in LLVM IR and compared the achieved speedups within each feature group. From this analysis, we were able to see that a further differentiation for memory operations was needed. For example, the loops

```
for (i = 0; i < LEN; i++){
       x[i] = y[i] + 1.;
}
```

and

```
for (i = LEN-1; i >= 0; i--){
       a[i] = b[i] + 1.;
}
```

share the same representation on IR level. However, speedup varies by 10% due to the reverse loop iteration. This difference stems from the fact that for the reverse loop, two half vectors of `b` are loaded and assembled instead of the one contiguous load operation used for `y` in the loop with the positive stride. This difference in code generation is not yet visible at IR level, since it will be performed later in the backend.

We therefore replaced the `load` and `store` features with more fine grain memory access pattern features. These access patterns were taken from the current cost model implementation and enable the differentiation between `Unknown`, `Vector`, `VecReverse`, `Interleaved`, `Gather/Scatter` and `Scalarized` for both, `load` and `store` accesses. This leaves 72 features to model the code. Not all of these featues are used to model loops, however. For example, out of the terminator instruction category, only the `branch` instruction is utilized. Nonetheless we decided to keep all features in our model to preserve the flexibility to use our cost modeling approach for other optimization passes, such as SLP vectorization.

C. Enhancing the Training Data

For the initial baseline analysis, the TSVC benchmark was used to get the vectorization results for over 150 test kernels. However, when training a model, a great number of different codes is desirable to ensure a decent feature coverage and sufficient code variety. This is especially true when trying to apply machine learning algorithms. In this spirit, a Loop Repository for Vectorizing Compilers (LORE) [18] has been created by a consortium of compiler researchers. At the time of writing, however, these codes were not yet readily accessible.

To enhance the initial dataset of TSVC kernels, we therefore compiled Polybench [19] and extracted those kernels that LLVM was able to vectorize with forced compilation. The extraction was necessary due to the fact that we need single loops or a single set of nested loops in our kernels. In Polybench, kernels can have more than one set of (nested) loops, however.

In total, 14 more kernels were added to the baseline setup. It results in training dataset of 99 vectorizable kernels on the AVX2 hardware and 85 vectorizable kernels on the NEON platform. Overall, the training codes cover a set of 31 features on AVX2 and 29 features on NEON.

VI. Experimental Results

Having defined all features and an extended training data set, the model is fitted to two different hardware platforms to demonstrate the portability of the approach. In this section, we present the results of the fitted model, including validation and a detailed feature analysis.

A. Cost Model Fitting

To create platform specific cost models out of our abstract code representation, we applied different fitting techniques to determine the most suitable one. With S_{meas} as the dependent variable and the instruction weights w_i as independent variables, three different approaches were tested:

- **Least Squares (LS):** This method determines the w_i that minimize the L^2 norm $||S_{meas} - S_{est}||^2$.
- **Non-negative Least Squares (NNLS):** This approach also minimizes the L^2 norm, but imposes an additional restriction on the resulting w_i, as they must not be negative.
- **Support Vector Regression (SVR) with Polynomial Kernel:** For this approximation, a support vector machine is used for regression instead of classification. The machine can utilize different kernels, such as linear, polynomial, or sigmoid kernels to approximate data. In this experiment, we used polynomial approximation to understand if a non-linear kernel is more suitable for our problem than the linear techniques.

All models were fitted using Python's `NumPy`, `SciPy`, and `scikit-learn` libraries [20], [21], [22]. For the SVR implementation, a grid search was conducted to find the most suitable parameter values for the error range ϵ, the error penalty C, and the polynomial degree. The parameter set with the least number of mispredictions was chosen, i.e. $(C, \epsilon) = (1, 1)$ and a polynomial degree of 2. All results can be seen in Figure 3.

Compared to the LLVM baseline in Figure 2, all three fitting methods were able to reduce the over-estimation of speedup significantly. However, the SVR fitted model is predicting the overall average speedup $S = 2.01$ for all training kernels. It is therefore not suitable for creating an accurate cost model and will not be discussed further. The linear fitting methods are able to increase the correlation from 0.58 to 0.88 (LS, +52%) and 0.79 (NNLS, + 36%) on the AVX2 platform, and from 0.75 to 0.88 (LS, + 17%) and 0.80 (NNLS, + 7%) on the NEON platform. At the same time, L^2 distances are decreased from 25.45 to 8.19 (LS, - 68%) and 10.9 (NNLS, -57%) on the AVX2 platform, and from 19.48 to 3.54 (LS, -82%) and 4.47 (NNLS, - 77%) on the NEON platform.

The number of mispredictions was reduced as well. On the AVX2 platform (baseline: $f\oplus : 4, f\ominus : 9$), the LS model is able to reduce both, the number of false positives and false negatives ($f\oplus : 3, f\ominus : 3$). All false positives were also mispredicted in the baseline model, while the false negative codes are a completely different set of kernels. The kernel that was removed from the baseline's set of false positives is a kernel with heavy control flow statements (kernel s279) that the LS model now predicts correctly. As a consequence of the overall reduction in false predictions, the normalized execution time decreases from 60.35 to 58.61 time units (-3%).

The NNLS model reduces the overall number of mispredictions from 13 to 9 ($f\oplus : 9, f\ominus : 0$). However, all mispredictions are false positives. This is due to the model's non-negative weights w_i, as an inaccurate weight will likely add on to the predicted speedup and thus cause false positives rather than false negatives. Since false positives are more harmful for performance due to the inferred slowdowns, the overall execution time consequently increases from 60.35 to 63.43 time units (+5%). It hints that the NNLS fitting method is not suitable for the presented modeling approach.

On the NEON platform (baseline: $f\oplus : 0, f\ominus : 17$), both fitted models decrease the number of mispredictions and achieve a reduction in execution time. The LS-fitted model eliminates 15 false negative predictions, while introducing only one false positive ($f\oplus : 1, f\ominus : 2$). The false positive code contains array indirections (kernel s4116) and is predicted to have a speedup $S_{est} = 1.12$, while it exhibits a small slowdown of $S_{meas} = 0.96$. Despite this slowdown, the overall execution time is reduced from 47.24 to 43.02 time units (-9%).

The model fitted with NNLS removes all false negatives, but introduces three false positives at the same time ($f\oplus : 3, f\ominus : 0$). The impact of these false positives is limited, however, and the model achieves a reduction in execution time from 47.24 to 43.14 time units (-9%) due to the eliminated false predictions. The overview of all model metrics can be found in Table I.

B. Model Validation

After fitting the model, we validated its predictive ability using *Leave One Out Cross Validation* (LOOCV). LOOCV is equivalent to a leave-p-out cross-validation with $p = 1$. The choice of p, preferred to larger values, is motivated by the training data's sparse dataset, as all training codes have been designed to tackle diverse individual patterns.

To run the LOOCV analysis, a model is trained leaving out one kernel. The speedup of the left-out kernel is then predicted using that trained model. This process is repeated for each kernel in the training dataset. Results for the LS- and NNLS-fitted models can be found in Table I. Plots to visualize the results for the most accurate model, the LS-fitted model, are presented in Figure 4.

As expected, the error of the LOOCV results is generally larger than when a model is trained on the whole data set.

On AVX2, the correlation drops from 0.88 on fitted data to 0.66 on LOOCV data for the LS-fitted model, which is still higher than the baseline of 0.58 (+13%). For the NNLS-fitted model, however, the correlation between estimated and measured speedup drops below baseline to 0.53 (-9%). Nonetheless, the average L^2 distances are still significantly lower than baseline for both models (LS: -47%, NNLS: -43%). In terms of mispredictions, neither model introduces new errors. They are consistent at ($f\oplus : 3, f\ominus : 3$) for LS and ($f\oplus : 9, f\ominus : 0$) for NNLS, with both models still mispredicting the same codes as previously. As a consequence, normalized execution times do not change and still present the results discussed in section VI-A: the LS-fitted model exhibits a speedup, while the NNLS-fitted model presents a slowdown.

On the NEON hardware, the correlation between estimated and measured speedup drops below baseline for both models,

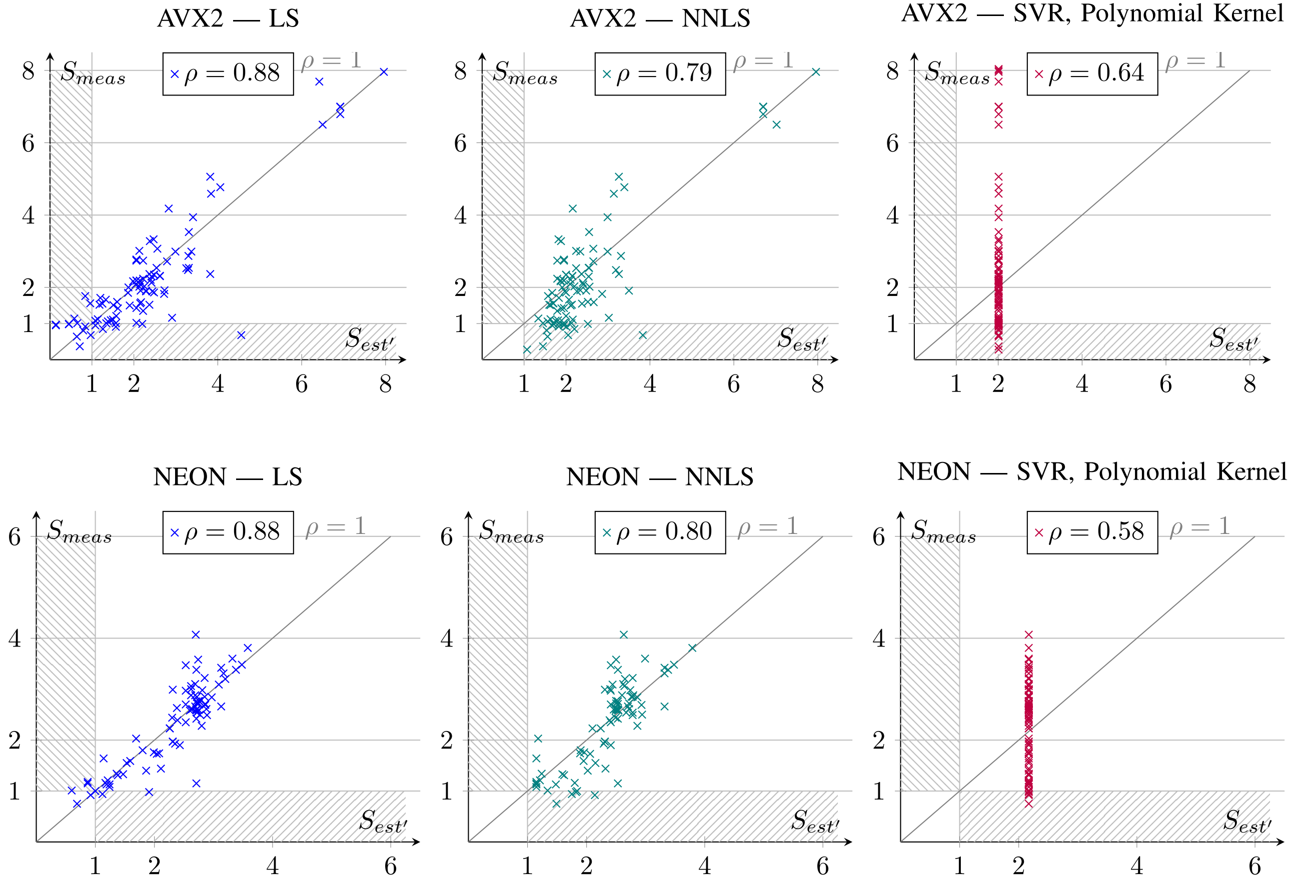

Fig. 3. Correlation between estimated and measured speedups of training data after linear fitting

from 0.76 to 0.62 (LS, -18%) and 0.37 (NNLS, -51%). Despite this drop in correlation, both models still outperform baseline in terms of L^2 distances (LS: -70%, NNLS: -57%). Furthermore, the baseline is exceeded in terms of number of mispredicted kernels and execution times. The LS-fitted model introduces one extra false positive and one extra false negative prediction ($f\oplus : 3, f\ominus : 2$). Regardless of these two additional mispredictions, the normalized execution time is still 8% below baseline at 43.49. For the NNLS-fitted model, one additional false negative is introdued ($f\oplus : 3, f\ominus : 1$), which increases the normalized execution time slightly to 44.04 time units (7% below baseline).

Based on the presented fitting and validation results, we assume that the Least Squares method is the most suitable one to fit our proposed cost model. We therefore focus on the analysis of the LS-fitted model and will not discuss the NNLS-based model further.

C. Feature Analysis

Having a fitted and validated model to predict code speedup, we can generate insight into what features are the most important for an accurate prediction on a specific target hardware. For this purpose, two different metrics were analyzed. First, a greedy forward feature selection was performed to understand which features are critical to reduce modeling error. Second, the obtained weights w_i were ranked, indicating which features contribute the most to code speedup and which features impact the speedup negatively.

Greedy forward feature selection is an algorithm that ranks a given feature set based on training data. It produces a list that indicates which features are the most essential in reducing model error. The algorithm starts with an empty feature set. It then selects the feature that produces the smallest model error when the model is trained with only one feature. This denominates the single best feature of the model. In its next iteration, the algorithm determines a second feature, which, combined with the already selected single best feature, produces the smallest model error for a model trained with two features. The algorithm then continues selecting features in this manner until a pre-determined number of features is selected or the model error is not reduced further.

For our proposed cost model, we chose the L^2 distance between estimated and modeled speedup as the error metric. The results of the greedy forward feature selection on our training data are listed in Table II. It can be seen that on both hardware platforms, the `getelementptr` feature is selected as the best feature. This is a feature that is present in all of our training data kernels, i.e. it has the best possible coverage. Furthermore, it is correlated to the total number of memory

	AVX2	LS		NNLS		NEON	LS		NNLS	
	Baseline	Fitted	LOOCV	Fitted	LOOCV	Baseline	Fitted	LOOCV	Fitted	LOOCV
Set Size	99					85				
ρ	0.58	0.88	0.66	0.79	0.53	0.76	0.88	0.62	0.80	0.37
L^2_{avg}	0.26	0.08	0.14	0.11	0.15	0.23	0.04	0.07	0.05	0.10
L^2_{max}	4.58	3.88	6.01	4.33	6.05	4.56	1.56	2.10	1.44	4.84
$f\oplus$	4	3	3	9	9	0	1	2	3	3
$f\ominus$	9	3	3	0	0	17	2	3	0	1
t_{scl}	99.00					85.00				
t_{vec}	60.35	58.61	58.61	63.43	63.43	47.24	43.02	43.49	43.14	44.04
t_{opt}	56.93					42.65				

TABLE I
IMPROVEMENTS ON TSVC AND POLYBENCH KERNELS AFTER DATA FITTING AND LEAVE ONE OUT CROSS VALIDATION

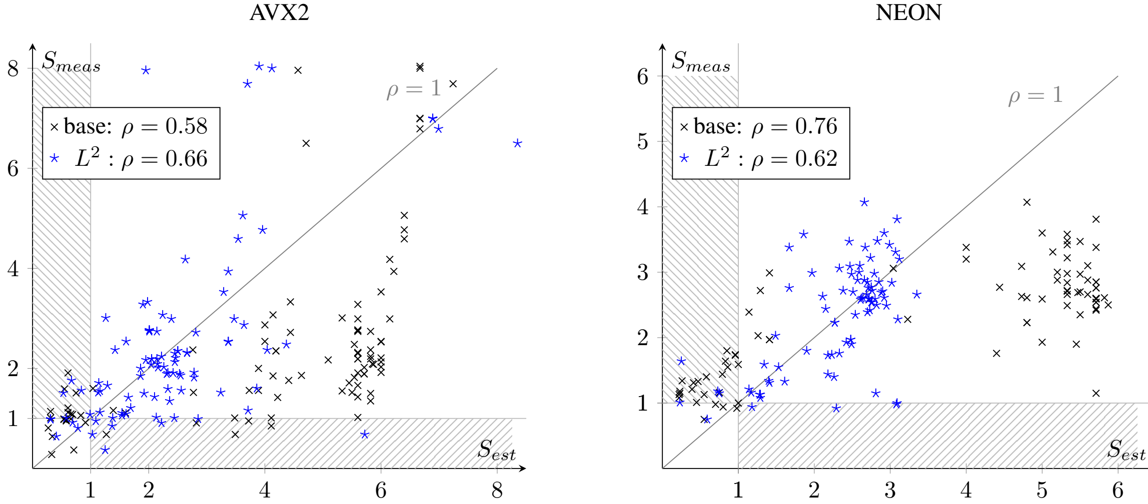

Fig. 4. Leave One Out Cross Validation on training data for LS-fitted model

	AVX2		NEON	
Rank	Feature	L^2	Feature	L^2
1	`getelementptr`	18.20	`getelementptr`	7.61
2	`shl`	17.25	`icmp`	6.86
3	`fptrunc`	16.45	`and`	6.51
4	`trunc`	15.85	`bitcast`	6.25
5	`br`	15.42	`fmul`	6.08
6	`fdiv`	15.10	`or`	5.98
7	`fmul`	14.97	`LD_VecReverse`	5.89
8	`lshr`	14.86	`sub`	5.80
Baseline		25.45		18.48
Full Model		8.19		3.54

TABLE II
TOP EIGHT FEATURES CHOSEN BY GREEDY FORWARD FEATURE SELECTION ON TRAINING DATA; ERROR METRIC IS THE EUCLIDEAN DISTANCE BETWEEN MODELED AND MEASURED SPEEDUPS

accesses that are performed within the loop. A model utilizing only this single best feature will alredy reduce the L^2 distance by 29% on AVX2 and by 41% on NEON platforms compared to their respective baselines. However, such a model would still infer a significant number of mispredictions, impacting the normalized execution time negatively.

Besides investigating which features are critical to obtain a small error in the model, it is also possible to analyze the feature values to understand how much each contributes to the estimated speedup. This is possible due to the linear nature of our cost model. As each feature is multiplied by its weight and summed up to get the estimated speedup (see Section V-A), the weights signify the impact on speedup. In this context, a positive weight means that the instruction will benefit from being vectorized; the higher the value, the higher the performance gain due to vectorization. A negative weight indicates overhead that is inferred due to the vectorization, i.e. it is not beneficial to vectorize this instruction. Such a feature ranking can also be used to hint programmers what instructions to avoid on certain platforms. Results for AVX2 and NEON hardware are shown in Table III. Interestingly, results vary significantly between the two hardwares.

For positive weights, i.e. those instructions that benefit from vectorization, the top five on AVX2 are arithmetic instructions,

	AVX2		NEON	
	Feature	w_i	Feature	w_i
⊕	fdiv	84.12	ST_VecReversed	15.41
	icmp	37.42	ST_Interleaved	7.70
	fcmp	31.83	ST_Vector	5.92
	sub	15.38	fsub	5.85
	fadd	12.13	ST_Scalarized	4.90
⊖	shl	-42.91	urem	-20.94
	LD_VecReverse	-23.53	call	-8.93
	fptosi	-17.27	LD_Scalarized	-6.13
	LD_Scalarized	-13.25	shl	-5.95
	br	-10.90	sext	-3.41

TABLE III
TOP FIVE HIGHEST FEATURE WEIGHTS AFTER FITTING; POSITIVE WEIGHTS CONTRIBUTE TO SPEEDUP, NEGATIVE WEIGHTS IMPACT SPEEDUP

while they are almost exclusively memory store accesses on NEON. It shows that vectorization success depends on very different code characteristics on the two platforms. It also emphasizes the importance to add code characteristics such as block composition/arithmetic intensity to the cost model. For negative weights, i.e. those instructions that are not beneficial to vectorize and might add overhead, results are more similar. On both platforms, the feature representing a scalarized load (`LD_SCALARIZED`) can be found. In this case, the impact on peformance stems from the inferred overhead that is needed for vector assembly. On AVX2, the `LD_VecReverse` is another load feature in the top five and is used for reverse loops. This is in line with our observation in Section V-B.

The feature analysis highlights the portability of the approach: despite our model being based on high-level features from LLVM bitcode, our proposed methodology is able to distinguish those code features that impact vectorization, independent of the target SIMD ISA.

VII. CONCLUSION

Compiler optimizations, such as vectorization, rely on cost modeling to assess the benefit of code transformations. To understand how accurate these cost models are, we analyzed the vectorization profitability prediction in LLVM's and GCC's auto-vectorizers. By comparing the correlation between predicted and measured speedup on more than 85 kernels, we are able to show that the current assessment over- estimates vectorization benefits. This leads to a weak-to-moderate correlation between estimated and actual speedup, mispredictions, and a loss in execution time.

We therefore propose a novel cost modeling approach that is platform independent and improves the state of the art of LLVM's performance prediction. Based on LLVM's intermediate representation, refined memory access features, and basic block composition, the resulting cost model is able to improve the prediction accuracy on all three metrics. Tested on two hardware platforms (based on AVX2 and NEON SIMD ISAs), the average Euclidean distance between the predicted and measured speedups is reduced by at least 65%. At the same time, the number of mispredictions decreases from 13 to 6 on AVX2 and from 17 to 5 on the NEON hardware. Consequently, the normalized execution time of the validation dataset is reduced by 3% on AVX2 and 9% on NEON.

By analyzing all features and their weights, we are furthermore able to generate platform specific insight. Due to the linear nature of our model, a feature correlates directly with its impact on vectorization, be it positive or negative. On our test hardwares, for example, we are able to identify that on AVX2, arithmetic instructions such as `fdiv` or `icmp` benefit the most from vectorization, while the same is true for `store` instructions on NEON.

In future work, we would like to apply our cost model to other optimization passes, such as the SLP vectorizer, to enable a single aligned cost model infrastructure in the compiler.

REFERENCES

[1] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An Evaluation of Vectorizing Compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pp. 372–382, IEEE Computer Society, 2011.

[2] M. J. Wolfe, *High Performance Compilers for Parallel Computing*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[4] J. Shin, M. Hall, and J. Chame, "Superword-Level Parallelism in the Presence of Control Flow," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pp. 165–175, IEEE Computer Society, 2005.

[5] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao, "An integrated simdization framework using virtual vectors," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, (New York, NY, USA), pp. 169–178, ACM, 2005.

[6] Z. Yuanyuan and Z. Rongcai, "An open64-based cost analytical model in auto-vectorization," in *2010 International Conference on Educational and Information Technology*, vol. 3, pp. V3–377–V3–381, Sept 2010.

[7] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pp. 132–143, 2006.

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 101–113, ACM, 2008.

[9] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, (Washington, DC, USA), pp. 327–337, IEEE Computer Society, 2009.

[10] K. Stock, L. Pouchet, and P. Sadayappan, "Using Machine Learning to Improve Automatic Vectorization," *TACO*, vol. 8, no. 4, pp. 50:1–50:23, 2012.

[11] E. Park, L.-N. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, (Washington, DC, USA), pp. 119–129, IEEE Computer Society, 2011.

[12] E. Park, J. Cavazos, L. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, "Predictive Modeling in a Polyhedral Optimization Space," *International Journal of Parallel Programming*, vol. 41, no. 5, pp. 704–750, 2013.

[13] A. Trouvé, 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.

[14] A. Trouvé, A. J. Cruz, K. J. Murakami, M. Arai, T. Nakahira, and E. Yamanaka, "Guide automatic vectorization by means of machine learning: A case study of tensor contraction kernels," *IEICE Transactions on Information and Systems*, vol. E99.D, no. 6, pp. 1585–1594, 2016.

[15] J. Shin, J. Chame, and M. W. Hall, "Compiler-controlled caching in superword register files for multimedia extension architectures," in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, (Washington, DC, USA), pp. 45–55, IEEE Computer Society, 2002.

[16] V. Porpodas and T. M. Jones, "Throttling Automatic Vectorization: When Less is More," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pp. 432–444, IEEE Computer Society, 2015.

[17] I. Rosen, D. Nuzman, and A. Zaks, "Loop-aware SLP in GCC," in *GCC Developers Summit*, 2007.

[18] Z. Chen, Z. Gong, J. J. Szaday, D. C. Wong, D. Padua, A. Nicolau, A. V. Veidenbaum, N. Watkinson, Z. Sura, S. Maleki, *et al.*, "Lore: A loop repository for the evaluation of compilers," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 219–228, IEEE, 2017.

[19] L.-N. Pouchet, U. Bondhugula, *et al.*, "The polybench benchmarks," *URL: http://web. cs. ucla. edu/pouchet/software/polybench*, 2017.

[20] T. Oliphant, *A Guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[21] T. Oliphant, "SciPy: Open source scientific tools for Python," *Computing in Science and Engineering*, vol. 9, pp. 10–20, 2007.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.