Vectorization Cost Modeling for NEON, AVX and SVE*

Angela Pohl[a,*], Biagio Cosenza[b], Ben Juurlink[a]

[a] *Department of Electrical Engineering and Computer Science, Technische Universität Berlin, Einsteinufer 17, 10587 Berlin, Germany*
[b] *Department of Computer Science, University of Salerno, Via Ponte Don Melillo, 84084 Fisciano (Salerno), Italy*

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 three different instruction set architectures, including both traditional SIMD architectures with a defined fixed vector register size (AVX2 and NEON), and novel instruction set with scalable vector size (SVE). 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, as well as an SVE simulator. Results show that the fitted model significantly improves the correlation between predicted and measured speedup (AVX2: +52% for training data, +13% for validation data), reduces the average error of the speedup prediction (SVE: -43% for training data, -36% for validation data), as well as the number of mispredictions (NEON: -88% for training data, -71% for validation data) for more than 80 code patterns.

1. 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

*This article extends the paper *Portable Cost Modeling for Auto-Vectorizers* presented at IEEE MASCOTS 2019 [1]

*Corresponding author. *E-mail address:* angela.pohl@tu-berlin.de (Angela Pohl)

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 3 explains these algorithms in detail). An additional challenge to design accurate vectorization cost models comes from recent novel vector ISAs with scalable vector lengths. An example of such an ISA is the Scalable Vector Extension (SVE) from ARM [2]. In contrast to traditional vector ISAs, such as AVX2 or NEON with a predefined fixed length for vector registers, SVE allows for scalable vector length, which can be inferred at runtime through a register access. As a result, SVE allows for Vector Length Agnostic (VLA) programming [3], and its supported instructions differ from the traditional ones (an extensive discussion can be found in the SVE reference manual [4]).

We have assessed the accuracy of the vectorization cost models of GCC and LLVM on the TSVC benchmark [5], on an Intel Xeon E5-2697 with AVX2 SIMD hardware extensions and on an ARMv8 CortexA-53 with a NEON SIMD unit. In addition, we extended the analysis to an SVE platform which was simulated using the gem5 simulator, since as of today, no implementation of SVE is commercially available yet. 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 suitable to train and improve 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), and new ISAs with scalable vector length pose additional challenges for modeling.

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 three different SIMD ISAs (AVX2, NEON and SVE), 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 AVX2, NEON and SVE 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.

This article expands on previous work [1] by extending our analysis and modeling work to the novel SVE vector instruction set. We enhanced our accuracy analysis of state-of-the-art compilers to cover different SVE vector lengths and modified our proposed cost model to support the new instructions. Furthermore, we performed a thorough analysis on our feature representation, training data gathering, and model fitting.

The paper is organized as follows: Section 2 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 3 and 4. Section 5 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 6. The paper concludes in Section 7.

2. Related Work

Automatic vectorization has been extensively studied in literature [6, 7], 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 [7] or if-conversion [8] 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. [9] recognized the importance of correctly deciding when SIMDization is profitable in the XL compiler. Yuanyuan and Rongcai [10] 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. [11] 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. [12] 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. [13]: 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. [14] 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. [15, 16] 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. [17] 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 [18]. 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 a larger number of code features, including those that distinguish different memory access patterns.

While features representations of the codes are typically manually created, Cummins et al. showed an end-to-end approach that automatically extracts relevant code features from the source code [19]. Ithemal [20] uses a hierarchical multiscale recurrent neural networks for data-driven basic block throughput estimation, which is based on the opcodes and operands of instructions in a

basic block. In contrast with our work, which is portable among different vector ISAs, Ithemal focuses on modern x86-64 ISA.

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 [21], and Throttled SLP (TSLP) [22], a SLP model that forces vectorization to stop earlier whenever this is beneficial, therefore overcoming the limits of standard greedy algorithms. goSLP [23] shows how local heuristics only explore a limited space among all available vectorization opportunities, leading to suboptimal solutions, and propose a globally optimized SLP framework based on ILP an solver and dynamic programming.

3. 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.

3.1. 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 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

```
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
```

Figure 1: Pseudo-Code of LLV's cost calculation in LLVM

be compared. It is also possible that one pass deems a transformation beneficial, while another may not. An analysis on which of these slightly varying cost models is more accurate, has not been performed yet.

3.2. Cost Modeling in GCC

The GCC vectorizer combines SLP and LLV vectorization in one compiler pass [24]. 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.

3.3. Cost Modeling for SVE in GCC

When the vector length is unknown, the compiler needs to ensure that vectorization is beneficial for all vector sizes. To avoid having to check all potential vector lengths for profitability, GCC assumes that speedup is constant or strictly

increasing with vector size, i.e. a code that is beneficial to vectorize with VF=4 is also beneficial for VF=16. It therefore performs a single cost analysis for the smallest possible vector size, which is 128 bit for ARM SVE. This approach does not take into account, however, that speedup might not scale perfectly, and increasing vector lengths can have negative impact on speedups, such as compute-bound kernels becoming memory bound. Nonetheless, we proceed with GCC's assumption that speedup scales perfectly with increasing vector sizes. It enables us to extrapolate the estimated speedup S for larger vector lengths, e.g.

$$S_{VF=16} = 4 \cdot S_{VF=4}$$

when analyzing the accuracy of the cost model for increasing vector lengths.

4. 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.

4.1. Setup

In this analysis, we compared speedups estimated by the compilers with actual measured speedups of the TSVC benchmark [5]. 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. We set the default number of loop iterations to $32,000$, thereby not considering loops with small trip counts. 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.

The third hardware is a platform that supports ARM's Scalable Vector Extension (SVE), a vector length agnostic SIMD ISA. As no SVE hardware is available on the market yet—first products are expected in 2021—we used the gem5 simulator for our measurements [25]. gem5 is a modular platform for processor and system architecture simulation. It is actively supported by ARM and a branch to simulate hardware with SVE is publicly available [26]. Within this branch, there is a choice of three different CPU models: atomic, in-order,

and out-of-order. Since SVE is targeted for HPC applications, we chose the out-of-order CPU model, which resembles an ARM Cortex-A72. For SVE, vector lengths are variable between 128 and 2048 bit. We selected the smallest vector length of 128 bit and the vector length of the first product to hit the market, Fujitus's A64FX with 512 bit vector length [27].

For compilation, we used Clang/LLVM 6.0 and GCC 8.2.0. For SVE, however, auto-vectorization is currently supported in GCC only. We built 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, including the vectorizers. Furthermore, the cost model is either set to unlimited (GCC) or all instruction costs are forced to 1 (LLVM) to vectorize all codes regardless of the corresponding cost model prediction.

Running the test bench 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. Since the gem5 simulator does not produce execution times but cycle counts, we used these values to derive the measured speedups for the ARM SVE hardware.

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.

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. On the SVE platform, GCC was able to vectorize 79 kernels.

4.2. Accuracy Metrics

The accuracy of the speedup prediction of compiler cost models in auto-vectorizers has never been studied. In this section, we therefore introduce metrics to qualify their accuracy. We distinguish between three kinds of metrics: precision, classification, and impact metrics. The first class of metrics measures how well the predicted and the measured performance correlate, while the second assesses if the compiler made the correct decision in terms of *"To vectorize or not to vectorize?"*. The third class quantifies the impact that inaccuracies within the cost model have on code performance. All metrics serve as optimization goals for our cost model approach.

The first class of metrics are precision metrics. As mentioned when discussing the experimental setup, we can derive an estimated speedup S_{est} and a measured speedup S_{meas} for each loop kernel in the benchmark. For an accurate cost model, the measured speedup of a code S_{meas} must equal the estimated speedup S_{est}, i.e.

$$S_{est} = S_{meas}.$$

Based on this relation, we can define two precision metrics:

- the correlation ρ between a set of estimated and measured speedups and
- the average and maximum prediction error.

To determine the correlation between the two datasets, we apply Pearson's Correlation Coefficient. To measure the error, i.e. the distance of each plot point to the straight line, we used the Euclidian Distance L^2 as the second precision metric. Due to the varying set sizes of vectorized kernels on the different platforms, denoted with m, we then normalized the distance to obtain the average value, i.e.

$$L^2_{avg} = \frac{1}{m} \cdot \sum_{j=1}^{m} ||S_{j,meas} - S_{j,est}||$$

Similarly, L^2_{max} is the maximum value from the set of L^2 distances.

The second class of metrics concerns the classification results of the compiler. During vectorization, the compiler has to decide if a transformation is beneficial. It therefore classifies loops into two categories: beneficial and not beneficial. An inaccurate cost model will consequently produce two types of classification errors:

- false positives ($f\oplus$): the compiler deems a transformation beneficial, but the vectorized code exhibits no speedup or a slowdown.
- false negatives ($f\ominus$): the compiler does not deem the transformation beneficial, but the vectorized code would have exhibited a speedup.

To account for measurement inaccuracies, we imposed a 5% threshold, i.e. kernel slowdowns are classified as $S_{meas} < 0.95$, while kernel speedups are classified as $S_{meas} > 1.05$. Out of the three generated code versions, the vectorized code will

contain the loops where the compiler deemed the vectorization to be beneficial. This incorporates patterns that do not exhibit any speedups or even slowdowns, i.e. the *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, i.e. the *false negatives ($f\ominus$)*.

Inaccurate cost models with low precision produce classification errors that harm code performance. As a third class of metrics, we therefore measure the impact the classification errors have on the execution time of a dataset. To understand the impact, we need to evaluate three different execution times:

- t_{scl}: the normalized scalar execution time of a benchmark with m kernels. t_{scl} equals the set size of vectorized kernels and serves as the baseline execution time.
$$t_{scl} = \sum_{j=1}^{m} \frac{t_{j,scl}}{t_{j,scl}} = m$$
- t_{vec}: the normalized vectorized execution time of a benchmark with m kernels as produced by the compiler, including classification errors. The vectorized execution time of each kernel is normalized to its respective scalar execution time and the runtimes of all kernels are added up, i.e.
$$t_{vec} = \sum_{j=1}^{m} \frac{t_{j,vec}}{t_{j,scl}}$$
When a kernel exhibits a speedup, $\frac{t_{j,vec}}{t_{j,scl}} < 1$, while $\frac{t_{j,vec}}{t_{j,scl}} > 1$ for slowdowns.
- t_{opt}: the normalized vectorized execution time of a benchmark with n kernels without classification errors, overwriting the classification by the compiler.

t_{vec} determines the execution time of a benchmark deploying a non-perfect cost model, such as the state of the art, while t_{opt} gives the optimal execution time that can be achieved.

4.3. Results

An overview of the results of our accuracy analysis is given in Table 1, while a visual representation is 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 $\rho = 1$.

Overall, both compilers tend to overestimate the speedup gained by vectorization. Such 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

	AVX2		NEON		SVE (GCC)	
	LLVM	GCC	LLVM	GCC	128 bit	512 bit
Size	85	65	71	66	79	79
ρ	0.58	0.33	0.75	0.48	0.29	0.39
L^2_{avg}	0.28	0.48	0.26	0.21	0.35	1.01
L^2_{max}	4.58	6.30	4.41	3.38	8.53	15.00
$f\oplus$	4	2	0	0	0	0
$f\ominus$	9	0	17	2	0	0
t_{scl}	85	65	71	66	79	79
t_{vec}	53.53	33.16	40.34	31.83	34.87	25.82
t_{opt}	51.79	32.53	36.54	31.03	34.87	25.82

Table 1: Accuracy metrics for baseline analysis of cost model's speedup prediction

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. The following subsections discuss the architecture-specific results in more detail.

4.3.1. Intel AVX2

On the AVX2 platform, both compilers tend to overestimate the speedup gain. 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, range between 1x-3x for both compilers. The average L^2 distance is 0.28 for LLVM and 0.48 for GCC, with maximum distances L^2_{max} reaching up to 4.58 (LLVM) and 6.30 (GCC), respectively.

In regards to the classification errors, LLVM mispredicts 13 kernels ($f\oplus$: 4, $f\ominus$: 9), while GCC does not produce any false negatives, but two false positives ($f\oplus$: 2, $f\ominus$: 0). These mispredictions reflect in our impact metric, the execution time. Even though GCC only produces two false positives, the difference between vectorized and optimal optimal execution time is 0.63 time units, i.e. the vectorized code reaches 98% of maximum performance ($S_{vec} = 1.96$, $S_{opt} = 2.00$). This gap is larger for the vectorized code generated by LLVM due to the large number of false negatives, i.e. missed vectorization opportunities. The 13 mispredictions from LLVM cause an overall difference of 1.74 time units and the vectorized kernels reach 97% of maximum performance ($S_{vec} = 1.59$, $S_{opt} = 1.64$).

4.3.2. ARM NEON

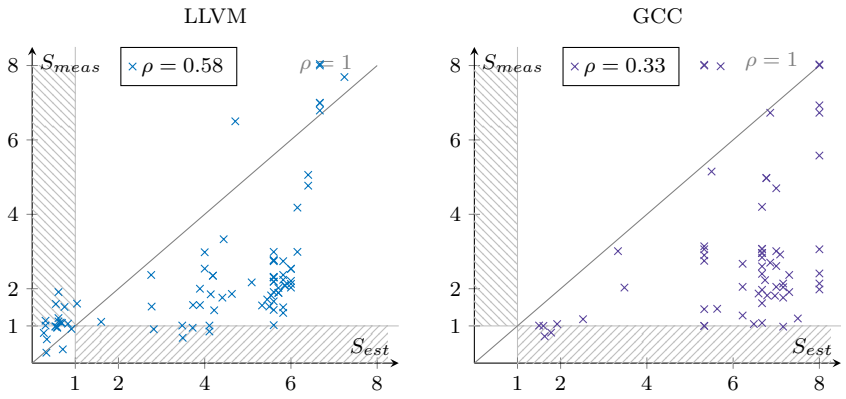
Compard to the AVX2 platform, correlations are higher on the NEON platform, with $\rho = 0.75$ (LLVM) and $\rho = 0.48$ (GCC). As can be seen in the plots,

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 ($S_{est} = VF$); there are kernels where both compilers even assume superlinear speedups ($S_{est} > VF$), too. Measured speedups are in the same range as on the AVX2 hardware, however, i.e. between 1x-3x. In regards to the average L^2 distance, both compilers achieve lower average errors than on the AVX2 platform (0.26 for LLVM, 0.21 for GCC). Taking the smaller vectorization factor and hence smaller target interval into account, on the other hand, shows a larger error relative to the vectorization factor.

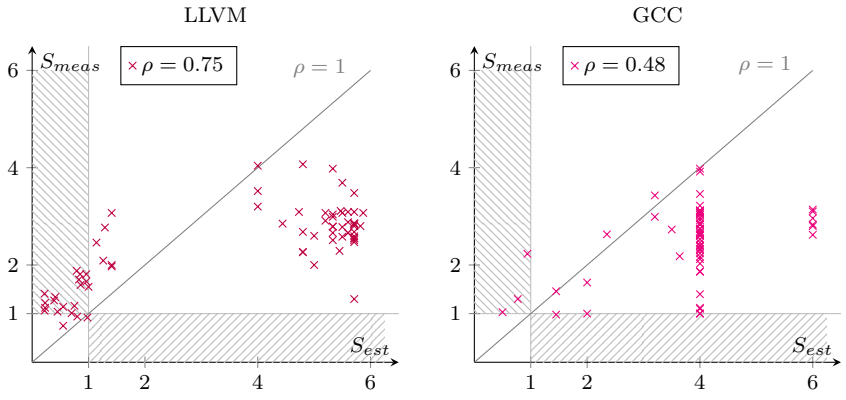
Contrary to the AVX2 platform, neither compiler produces false positives on the ARM NEON hardware. LLVM is conservative in its decisions, however, and produces 17 false negatives ($f\oplus : 0,\ f\ominus : 17$), while GCC only produces two ($f\oplus : 0,\ f\ominus : 2$). As a consequence, performance achieved by the code vectorized by GCC is close to its optimum ($S_{vec} = 2.07$, $S_{opt} = 2.10$). The code vectorized by LLVM, however, only achieves 91% of the optimum performance ($S_{vec} = 1.76$, $S_{opt} = 1.94$).

4.3.3. ARM SVE

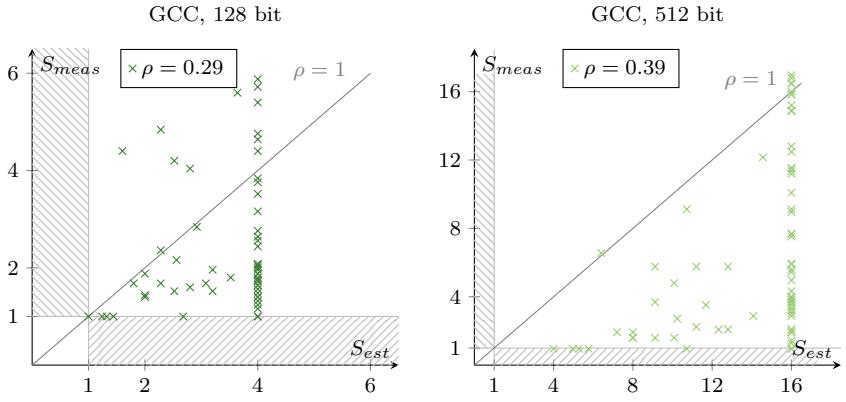
Figure 2c shows the analysis results for two different vector lengths, 128 bit and 512 bit, both executing the same vector length agnostic code on the simulated hardware. The estimated speedups are identical for both plots, scaled to their respective vectorization factors, as GCC performs a single benefit analysis to cover all potential vector lengths. The measured speedups are generated by two different simulation runs with respective hardware settings. For both vector lengths, the correlation between the estimated and the measured speedup is low, i.e. $\rho = 0.29$ for a vector length of 128 bit and $\rho = 0.39$ for a vector length of 512 bit. As on the ARM NEON platform, GCC estimates the majority of kernels to scale perfectly, but actual speedups are lower. There is a set of kernels, however, where the simulator assumes a perfect scaling of vector instructions, i.e. no overhead added by vectorization, or even super-linear speedups. This can be observed in particular on the 128-bit platform, where measured speedups can be significantly higher than the vectorization factor ($S_{meas} > 1.5 \cdot VF$). As a consequence, the average Euclidian distance ranges between 0.35 and 1.01, and the maximum distance for the 128-bit platform is as high as $L^2_{max} = 8.53$. Despite the low correlation, GCC does not produce any classification errors ($f\oplus : 0,\ f\ominus : 0$). This is coherent with the results from the ARM NEON platform, where GCC only produces false negatives, but no false positives. Due to the optimistic runtime estimation by the simulator, there are no false negatives on the ARM SVE platform.



(a) Intel AVX2



(b) ARM NEON



(c) ARM SVE

Figure 2: Analysis results for the TSVC benchmark for LLVM and GCC loop level vectorization passes on Intel AVX2 (2a), ARM NEON (2b), and ARM SVE (2c) hardware; SVE plots show GCC vectorization results for two different vector lengths; shaded areas mark false positive and false negative predictions, straight line marks perfect positive correlation of $\rho = 1$

5. 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. Furthermore, we discuss how to modify the existing model for the SVE architecture, i.e. adding ISA specific features, as well as avoiding overfitting of the model.

5.1. 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

$$S_{est} = \frac{c_{scalar}}{c_{vec}} = \frac{c_{scalar}}{\sum\limits_{\forall i \in I} n_i c_i}$$

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

$$S_{est} = \sum_{\forall i \in I} n_i w_i$$

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 $N = \sum_{\forall i \in I} n_i$, we account for different instruction mixes. The model thus becomes

$$S_{est} = \sum_{\forall i \in I} \frac{n_i}{N} w_i$$

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).

5.2. 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 modeling, 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 following two loops share the same representation on IR level:

```
for(i=0; i < LEN; i++){
   x[i] = y[i] + 1.0;
}
```

```
for(i=LEN-1; i>=0; i--){
   a[i] = b[i] + 1.0;
}
```

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.

The SVE instruction set architecture introduces a set of instructions to efficiently vectorize specific code patterns. The most notable instructions are the `first fault in register` memory operations and the `whilelt` instruction for loop management. These instructions were therefore added to the model,

although their use is not covered by existing testbenches [28]. In addition, instructions in the SVE instruction set are typically predicated and predication requires additional computing time, therefore impacting the speedup. Instead of specifically modeling this overhead, however, our approach already accounts for it indirectly by moving the model target from cost to speedup.

5.3. 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) [29] 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 [30] 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.

We were not able to extended the training data set for the SVE hardware, however. After removing kernels where the gem5 simulator estimates a superlinear speedup (-3 kernels) and those that share the same feature representation (-22 kernels), we analyzed the remaining feature coverage. To ensure a stable model, we discarded all kernels with a unique feature, leaving only 31 kernels in the training data set.

5.4. Dimensionality Reduction for SVE Hardware

Due to the small training data set for the SVE hardware, we chose to reduce the dimensionality of the feature set for the SVE models. Otherwise, keeping the large feature space will result in an overfitted model and a limited capability to predict kernel performance outside of the training data set. The decision which features to keep for the respective model was based on the contribution of each feature towards the overall error reduction in the model, i.e. its impact in minimizing the L^2 distance. We therefore conducted a greedy forward feature selection for each of our fitting approaches and selected the top ten features. For two of the three fitted models, even less than ten features were needed, as the model error did not decrease further with adding more features. The resuls of the greedy forward feature selection are discussed in detail in Section 6.3.

6. Experimental Results

Having defined all features and an extended training data set, the model is fitted to three 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.

6.1. 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 [31, 32, 33]. 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 4. 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. The models fitted with support vector regression, however, either predict the overall average speedup of $S = 2.01$ for the AVX2 and NEON platform, or are overfitted on the SVE hardware. While the high correlation of $\rho = 0.92$ seems promising, the model becomes unstable during cross validation and produces a significant amount of mispredictions. The fitting approach with SVR 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, from 0.75 to 0.88 (LS, + 17%) and 0.80 (NNLS, + 7%) on the NEON platform, and from 0.48 to 0.79 (LS, +65%) and 0.63 (NNLS, +31%) on the SVE 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, from 19.48 to 3.54 (LS, -82%) and 4.47 (NNLS,

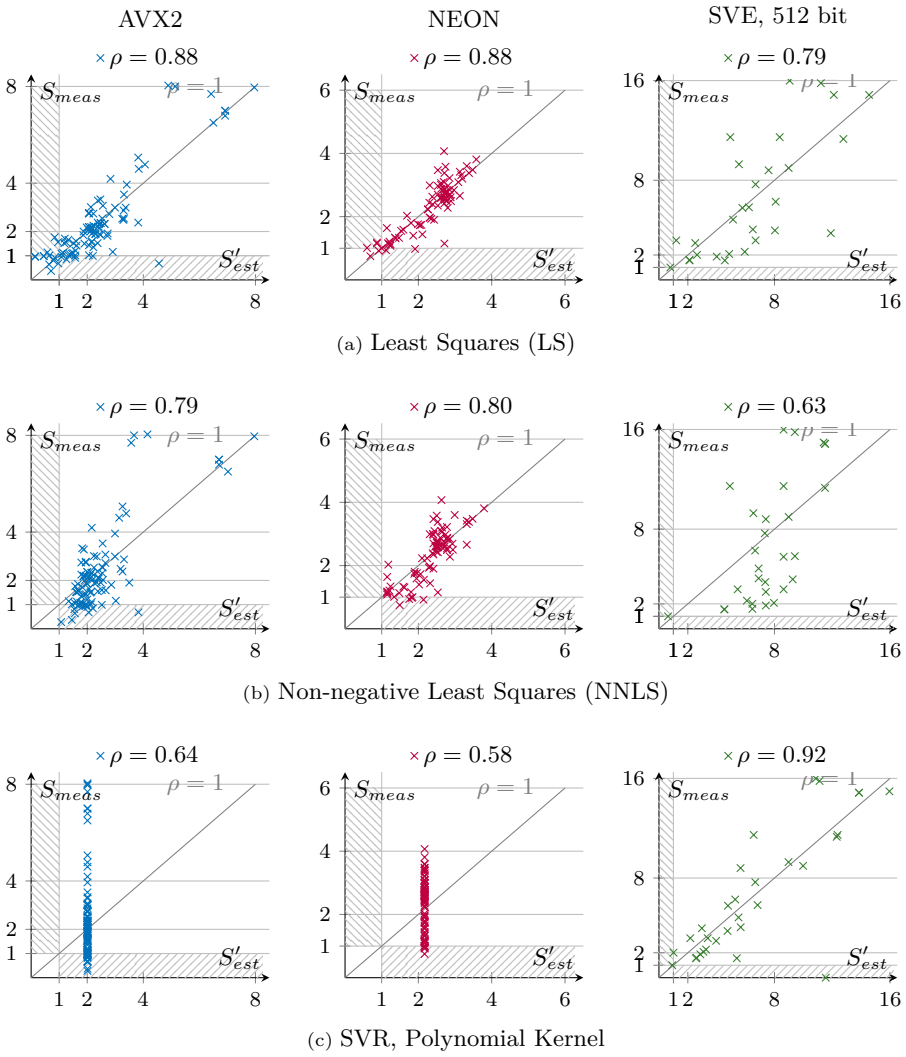


Figure 3: Correlation between estimated and measured speedups of training data after fitting for three fitting techniques; left column shows fitting for AVX2 platform, middle column for NEON platform, right column for SVE platform with 512 bit vector length

- 77%) on the NEON platform, and from 44.78 to 17.36 (LS, -61%) and 25.96 (NNLS, -42%) on the SVE platform.

The number of mispredictions was reduced as well for two of the three platforms. 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 this specific platform.

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.

On the SVE platform, the models do not introduce any mispredictions, and are therefore comparable to the GCC baseline. The overview of all model metrics can be found in the summarizing tables in Figure 4.

6.2. Model Validation

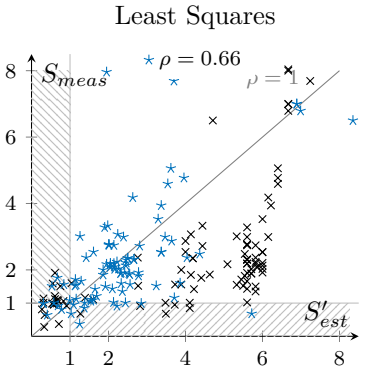
After fitting the model, we validated its stability and predictive ability using *Leave One Out Cross Validation* (LOOCV). LOOCV is equivalent to a leave-p-out cross-validation with $p = 1$. The choice of p, preferred to larger values, is motivated by the training data's sparse dataset, as all training 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 the summarizing tables in Figure 4. The figure also visualizes the results for the best fitting model, i.e. the LS-fitted models for the AVX2 and NEON hardware, and the NNLS-fitted model for the SVE platform.

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

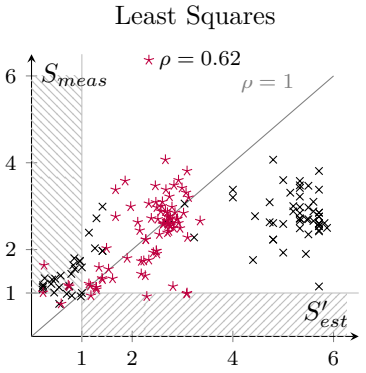
AVX2

		LS		NNLS	
	Baseline	Fitted	LOOCV	Fitted	LOOCV
Size	99				
ρ	0.58	0.88	0.66	0.79	0.53
L^2_{avg}	0.26	0.08	0.14	0.11	0.15
L^2_{max}	4.58	3.88	6.01	4.33	6.05
$f\oplus$	4	3	3	9	9
$f\ominus$	9	3	3	0	0
t_{scl}	99				
t_{vec}	60.35	58.61	58.61	63.43	63.43
t_{opt}	56.93				



NEON

		LS		NNLS	
	Baseline	Fitted	LOOCV	Fitted	LOOCV
Size	85				
ρ	0.76	0.88	0.62	0.80	0.37
L^2_{avg}	0.23	0.04	0.07	0.05	0.10
L^2_{max}	4.56	1.56	2.10	1.44	4.84
$f\oplus$	0	1	2	3	3
$f\ominus$	17	2	3	0	1
t_{scl}	85				
t_{vec}	47.24	43.02	43.49	43.14	44.04
t_{opt}	42.65				



SVE, 512 bit

		LS		NNLS	
	Baseline	Fitted	LOOCV	Fitted	LOOCV
Size	31				
ρ	0.48	0.79	0.47	0.63	0.48
L^2_{avg}	6.56	2.42	4.23	3.76	4.15
L^2_{max}	13.99	8.15	15.30	9.00	9.31
$f\oplus$	0	0	0	0	0
$f\ominus$	0	0	2	0	0
t_{scl}	31				
t_{vec}	14.72	14.72	16.16	14.72	14.72
t_{opt}	14.72				

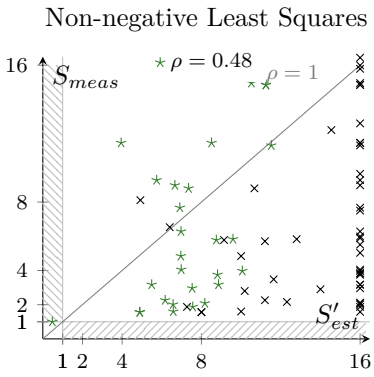


Figure 4: Results of Leave One Out Cross Validation on training data for AVX, NEON, and SVE hardware; plots show best fitted models, with black plot points marking the baseline, colored plot points marking results of Leave One Out Cross validation

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 6.1: 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, 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).

On the SVE hardware, both models maintain the same correlation as the baseline, while decreasing the average L^2 distances. The LS-fitted model reduces L^2_{avg} from 6.56 to 4.23 (-35%), whereas the NNLS-fitted model reduces it to 4.15 (-37%). The L^2_{max} distance increases for the LS-fitted model, however, from 13.99 to 15.30 (+9%). This is due to two mispredictions that the model introduces. These two false negatives further impact the overall execution time, increasing it from the optimal execution time of $t_{opt} = 14.72$ to 16.16 time units (+10%). The NNLS-fitted model, on the other hand, decreases L^2_{max} to 9.31 (-33%) and does not produce any classification errors. It therefore achieves the optimal runtime.

6.3. 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

	AVX2		NEON		SVE	
Rank	Feature	L^2	Feature	L^2	Feature	L^2
1	`getelementptr`	18.20	`getelementptr`	7.61	`icmp`	26.13
2	`shl`	17.25	`icmp`	6.86	`fadd`	24.27
3	`fptrunc`	16.45	`and`	6.51	`getelementptr`	22.76
4	`trunc`	15.85	`bitcast`	6.25	`phi`	20.92
5	`br`	15.42	`fmul`	6.08	`add`	19.82
6	`fdiv`	15.10	`or`	5.98	`LD_Widen`	19.11
7	`fmul`	14.97	`LD_VecReverse`	5.89	`br`	18.31
8	`lshr`	14.86	`sub`	5.80	`LD_Interleave`	17.80
Full Model		8.19		3.54		17.36
Baseline		25.45		16.94		44.78

Table 2: Top eight features chosen by greedy forward feature selection on training data; error metric is the Euclidean distance between modeled and measured speedups

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 2. It can be seen that on all hardware platforms, the `getelementptr` feature is selected among the top three most important features. It 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 accesses that are performed within the loop. The same coverage applies to the No. 1 ranked feature on the SVE platform; the `icmp` instruction is used within loops to determine if sufficient loop iterations have been executed. A model utilizing only their respective single best feature will already reduce the L^2 distance by 29% on AVX2, by 41% on NEON, and by 42% on SVE platforms compared to their respective baselines. However, such a model would still infer a significant number of mispredictions, impacting the normalized execution time negatively.

On the SVE platform, the greedy forward feature selection algorithm further determined that not all features might be critical to model the small training data set of 31 kernels. The LS approch reduces its error with each additional feature, although the L^2 distance decreases by a mere 2% only when adding more than 8 features. Even less features are needed for the NNLS approach. Here, the algorithm is not able to reduce the error further by adding more than 4 features to the model (`icmp, fadd, br, fcmp`). We assume, however, that more features will be needed in the future with more test cases (and hence a higher feature coverage) and more accurate performance measurements when the new hardware is available.

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

	AVX2		NEON		SVE	
	Feature	w_i	Feature	w_i	Feature	w_i
$\oplus$	`fdiv`	84.12	`ST_VecReversed`	15.41	`icmp`	169.37
	`icmp`	37.42	`ST_Interleave`	7.70	`LD_Interleave`	59.21
	`fcmp`	31.83	`ST_Vector`	5.92	`LD_Widen`	53.70
	`sub`	15.38	`fsub`	5.85	`fadd`	42.11
	`fadd`	12.13	`ST_Scalarized`	4.90	`br`	12.96
$\ominus$	`shl`	-42.91	`urem`	-20.94	`getelementptr`	-45.36
	`LD_VecReverse`	-23.53	`call`	-8.93	`add`	-43.05
	`fptosi`	-17.27	`LD_Scalarized`	-6.13	`phi`	-29.39
	`LD_Scalarized`	-13.25	`shl`	-5.95	`ST_Scalarized`	-21.92
	`br`	-10.90	`sext`	-3.41	`fmul`	-17.04

Table 3: Top five highest feature weights after fitting; positive weights contribute to speedup, negative weights diminish speedup

nature of our cost model. As each feature is multiplied by its weight and summed up to get the estimated speedup (see Section 5.1), 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 a code characteristic that impacts the speedup gain, e.g. due to additional overhead. Such a feature ranking can also be used to hint programmers what instructions to avoid on certain hardware. Results for the AVX2, NEON, and SVE platforms are shown in Table 3.

Interestingly, results vary significantly between the platforms. For positive weights, i.e. those instructions that benefit from vectorization, the top five on AVX2 are arithmetic instructions, while they are almost exclusively memory store accesses on NEON. It shows that vectorization success depends on 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 5.2.

On the SVE platform, a different pattern can be observed. Most notably, the `getelementptr` instruction is the instruction with the highest negative weight. This instruction is added to LLVM bitcode for every memory access. The model recognizes the negative impact memory instructions have on code vectorization for such large vectors. Exceptions are those `load` instructions that can be found on the list of instructions with high positive weights, e.g. `LD_Interleave`,

where a speedup can be obtained due to the model compensating the penalty imposed by the `getelementptr` feature. A similar pattern can be deducted for control flow. The `phi` instruction is weighted negatively, i.e. branching within vectorized code is considered harmful. However, this does not apply to the `icmp` instruction, which is used to determine the end of a loop, but instructions such as `fcmp` that are used for `if-else`-statements within loop bodies.

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.

7. 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 with respect to all three metric categories: precision, classification, and impact. Our modeling approach is independent of the data type used and can, therefore, be applied to programs using smaller data types and mixed precision techniques. Tested on three hardware platforms (based on AVX2, NEON and SVE 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, while maintaining an error-free prediction on the SVE platform. 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 platforms, 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. On the SVE hardware, we can observe that memory instructions in general impact performance negatively, with the exception of certain `load` instructions, e.g. `LD_Interleave`.

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. Regarding vector length agnostic architectures, it will

also become critical to understand if a transformation will become detrimental to speedup with larger vector lengths. An accurate cost model is therefore needed to correctly predict such vector length limits, which can then be enforced via runtime checks or enforcing smaller vector lengths. A further analysis will be needed, however, as soon as hardware is available.

References

[1] A. Pohl, B. Cosenza, B. H. H. Juurlink, Portable cost modeling for auto-vectorizers, in: 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2019, Rennes, France, October 21-25, 2019, 2019, pp. 359–369. doi:10.1109/MASCOTS.2019.00046.
URL https://doi.org/10.1109/MASCOTS.2019.00046

[2] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, P. Walker, The ARM scalable vector extension, IEEE Micro 37 (2) (2017) 26–39. doi:10.1109/MM.2017.35.
URL https://doi.org/10.1109/MM.2017.35

[3] A sneak peak into SVE and VLA programming, https://developer.arm.com/hpc/a-sneak-peek-into-sve-and-vla-programming, accessed: 2020-02-02.

[4] A-Profile Architecture Specifications, https://developer.arm.com/products/architecture/a-profile/docs, accessed: 2020-02-02.

[5] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, D. A. Padua, An Evaluation of Vectorizing Compilers, in: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT '11, IEEE Computer Society, 2011, pp. 372–382. doi:10.1109/PACT.2011.68.
URL http://dx.doi.org/10.1109/PACT.2011.68

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

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

[8] J. Shin, M. Hall, J. Chame, Superword-Level Parallelism in the Presence of Control Flow, in: Proceedings of the International Symposium on Code Generation and Optimization, CGO '05, IEEE Computer Society, 2005, pp. 165–175. doi:10.1109/CGO.2005.33.
URL http://dx.doi.org/10.1109/CGO.2005.33

[9] P. Wu, A. E. Eichenberger, A. Wang, P. Zhao, An integrated simdization framework using virtual vectors, in: Proceedings of the 19th Annual International Conference on Supercomputing, ICS '05, ACM, New York, NY, USA, 2005, pp. 169–178. doi:10.1145/1088149.1088172.
URL http://doi.acm.org/10.1145/1088149.1088172

[10] Z. Yuanyuan, Z. Rongcai, An open64-based cost analytical model in auto-vectorization, in: 2010 International Conference on Educational and Information Technology, Vol. 3, 2010, pp. V3–377–V3–381. doi:10.1109/ICEIT.2010.5608348.

[11] D. Nuzman, I. Rosen, 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, 2006, pp. 132–143. doi:10.1145/1133981.1133997.
URL http://doi.acm.org/10.1145/1133981.1133997

[12] U. Bondhugula, A. Hartono, J. Ramanujam, 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, ACM, New York, NY, USA, 2008, pp. 101–113. doi:10.1145/1375581.1375595.
URL http://doi.acm.org/10.1145/1375581.1375595

[13] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, 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, IEEE Computer Society, Washington, DC, USA, 2009, pp. 327–337. doi:10.1109/PACT.2009.18.
URL https://doi.org/10.1109/PACT.2009.18

[14] K. Stock, L. Pouchet, P. Sadayappan, Using Machine Learning to Improve Automatic Vectorization, TACO 8 (4) (2012) 50:1–50:23.

[15] E. Park, L.-N. Pouchet, J. Cavazos, A. Cohen, 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, IEEE Computer Society, Washington, DC, USA, 2011, pp. 119–129.
URL http://dl.acm.org/citation.cfm?id=2190025.2190059

[16] E. Park, J. Cavazos, L. Pouchet, C. Bastoul, A. Cohen, P. Sadayappan, Predictive Modeling in a Polyhedral Optimization Space, International Journal of Parallel Programming 41 (5) (2013) 704–750.

[17] A. Trouvé, A. J. Cruz, D. B. Brahim, H. Fukuyama, K. J. Murakami, H. A. Clarke, M. Arai, T. Nakahira, E. Yamanaka, Predicting vectorization profitability using binary classification, IEICE Transactions 97-D (12) (2014)

3124–3132. doi:10.1587/transinf.2014EDP7190.
URL https://doi.org/10.1587/transinf.2014EDP7190

[18] A. Trouvé, A. J. Cruz, K. J. Murakami, M. Arai, T. Nakahira, E. Yamanaka, Guide automatic vectorization by means of machine learning: A case study of tensor contraction kernels, IEICE Transactions on Information and Systems E99.D (6) (2016) 1585–1594. doi:10.1587/transinf.2015EDP7440.

[19] C. Cummins, P. Petoumenos, Z. Wang, H. Leather, End-to-end deep learning of optimization heuristics, in: 26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017, 2017, pp. 219–232. doi:10.1109/PACT.2017.24.
URL https://doi.org/10.1109/PACT.2017.24

[20] C. Mendis, A. Renda, S. P. Amarasinghe, M. Carbin, Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks, in: Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, 2019, pp. 4505–4515.
URL http://proceedings.mlr.press/v97/mendis19a.html

[21] J. Shin, J. Chame, 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, IEEE Computer Society, Washington, DC, USA, 2002, pp. 45–55.
URL http://dl.acm.org/citation.cfm?id=645989.674318

[22] V. Porpodas, 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, IEEE Computer Society, 2015, pp. 432–444. doi:10.1109/PACT.2015.32.
URL https://doi.org/10.1109/PACT.2015.32

[23] C. Mendis, S. P. Amarasinghe, goSLP: Globally optimized superword level parallelism framework, PACMPL 2 (OOPSLA) (2018) 110:1–110:28. doi:10.1145/3276480.
URL https://doi.org/10.1145/3276480

[24] I. Rosen, D. Nuzman, A. Zaks, Loop-aware SLP in GCC, in: GCC Developers Summit, 2007.

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al., The gem5 Simulator, ACM SIGARCH Computer Architecture News 39 (2) (2011) 1–7.

[26] gem5 SVE Branch, `https://gem5.googlesource.com/arm/gem5/+/sve/beta1`, accessed: 2019-04-05.

[27] T. Yoshida, Fujitsu high performance CPU for the Post-K Computer, in: Hot Chips 30 Symposium (HCS), Series Hot Chips, Vol. 18, 2018.

[28] A. Pohl, M. Greese, B. Cosenza, B. Juurlink, A Performance Analysis of Vector Length Agnostic Code, in: Proceedings of the 2018 International Conference on High Performance Computing & Simulation (HPCS), 2019.

[29] 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), IEEE, 2017, pp. 219–228.

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

[31] T. Oliphant, A Guide to NumPy, Vol. 1, Trelgol Publishing USA, 2006.

[32] T. Oliphant, SciPy: Open source scientific tools for Python, Computing in Science and Engineering 9 (2007) 10–20.

[33] 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 12 (Oct) (2011) 2825–2830.