Using Graph-Based Characterization for Predictive Modeling of Vectorizable Loop Nests

William Killian
killian@udel.edu

Department of Computer and Information Science
University of Delaware
Newark, Delaware, USA

January 6, 2015

Abstract

Newer architectures continue to expand vector sizes and increase the different number of vector instructions. Because optimizing compilers have to be continually updated to take advantage of new vectorization capabilities of hardware being released, a good vectorizing compiler is typically released long after new hardware has been on the market. Obtaining the best possible performance on multicore SIMD architectures mandates the use of vectorization. Unfortunately, compilers are not always able to generate optimal code for the hardware; detecting and generating vectorized code is extremely complex. Programmers can use a number of tools to aid in development and tuning, but most of these tools require expert or domain-specific knowledge to use. In this work, we aim to provide automated techniques for determining the best way to optimize certain codes, with an end goal of guiding the compiler to generate highly optimized code without requiring expert knowledge from the developer. Existing work is inefficient to our goal because they can only work on entire programs when optimizations may need to be granular enough for individual loop nests. First, we choose a subset of optimizations which control vectorization and exhaustively search the optimization space to find the optimal speedup in this optimization search space for each loop nest. Then we exhaustively apply those optimizations on different loop nests from two sets of benchmarks. Our exhaustive search space lead to an average speedup of over 50% over the most aggressive setting in the Intel Compiler. Finally, we construct a graph-based machine learning prediction model, which takes as input the control flow graph of a loop and the vectorization that could be performed on this loop and and predicts the speedup obtained by applying that vectorization to the loop over a default vectorization setting. We show that for leave-one-out cross validation, our prediction model can select optimizations achieving up to an average of 88% of optimal for a given architecture.

Contents

1 Introduction **2**

2 Intel Compiler Optimizations **3**

3 Kernels **4**

4 Version Generation **5**

5 Vectorization Analysis **6**

6 Graph-Based Speedup Predictor **12**

7 Threats to Validity **17**

8 Related Work **23**

9 Future Work **24**

10 Conclusion **24**

Keywords: SIMD, Vectorization, Vectorizer Heuristics, Code Optimization, Graph-Based Learning, Machine Learning

1 Introduction

Obtaining the best possible performance on multicore SIMD architectures mandates the use of vectorization. The complexity of SIMD support through increased vector size and instructions only makes it more difficult to target applications for these superscalar architectures. Unfortunately, compilers are not always able to generate optimal code for the hardware; detecting and generating vectorized code is extremely complex. The Intel Compiler has its own internal heuristics used to determine whether or not code should be vectorized, but they do not provide direct access for the developer to see why the compiler chooses one optimization over another. Vectorization reports can help relay information from the compiler to the programmer. Some compilers also provide the capability for the programmer to aid the compiler with vectorization with the use of compiler-specific pragmas.

The number of tools a programmer can use to aid in development continues to grow, but most of these tools require expert or domain-specific knowledge to use (e.g. vectorization reports). We aim to provide an exhaustive search compilation method of determining the best way to optimize certain codes. Intel offers a Vectorization Optimization Guide [16] for their architecture and development tools, but it cannot directly carry over to other platforms. Stock et al. [28] propose using machine learning techniques to improve automatic vectorization. This existing work is inefficient to our goal because they can only work on entire programs when optimizations may need to be granular enough for individual loop nests. Graph-based learning techniques have also been applied at a loop-nest granularity, but not targeting vectorization optimization search spaces [8, 24].

Our experiments consisted of exhaustive search space compilation of a benchmark suite designed for the evaluation of vectorizing compilers [20] and another benchmark suite designed for polyhedral compilers [27]. To aid with version generation and compilation, we created two utilities. One is a source-to-source compiler which translates a simplified directive language to a specific directive language supported by a given compiler (e.g. Intel Compiler). The other utility performs version generation among a set of vectorization optimizations. The creation of these utilities can help non-experts in the generation, execution, and analysis of programs. Finally, we used our speedup predictor to choose the optimization sequence to apply that yields the best predicted speedup. Experiments show that our classier is able to predict the optimizations that give a speedup on our loop nests on average up to 88% of optimal speedup on our particular optimization search space.

The contributions of this work can help programmers tailor their applications to better exploit vectorization. The iterative compilation utilities can be especially useful when exploring tuning options during the code optimization phase. The speedup predictor can be used as part of a general-purpose auto-tuning strategy that would require minimal human intervention.

In this work we make the following contributions:

1. `VALT` – a directive compiler used to simplify code generation across different compiler backends
2. `autovec` – an iterative compiler utility to generate different versions of the same code

3. A graph-based speedup predictor specifically designed to predict the best vector optimizations to apply to a particular loop-nest.

The rest of the report is outlined as follows: Section 2 goes over the command-line and directive-based compiler optimizations which can be applied to a program. Section 3 introduces the kernels used in the evaluation of our work. Section 4 describes our `VALT` and `autovec` tools used for version generation. Our vectorization analysis is illustrated in Section 5. The speedup predictor is introduced and evaluated in Section 6. We go over threats of validity, related work, and future work in Sections 7, 8, and 9, respectively. Section 10 highlights the conclusion of this work.

2 Intel Compiler Optimizations

The Intel C++ Compiler version 15.0 offers many optimization options for the programmer. Some of these are restricted to command-line flags, while others must be inserted as directives into the original code. This section addresses the relevant optimization flags and directives used in this work.

2.1 Command-Line Flags

All configurable programmer-passed global optimizations happen through command-line flags. When performing code optimization, optimization-level flags (`-O2`,`-O3`) tend to be the most commonly used. In addition to specifying a set of optimizations to apply, the programmer is also able to specify the target architecture and modify the default code generation rules. Listed below are relevant command-line flags used in this work.

- `-O3` - apply all `-O1` and `-O2` optimization sets in addition to a new set. A full list of these optimizations are viewable from the Intel C++ Compiler reference guide [17].
- `-xHOST` - specifies that the architecture we are optimizing for is the same architecture we are compiling on. This is equivalent to specifying the native architecture on your system. For our experiments, `-xAVX2` would be applicable to the Haswell (HSW) microarchitecture and `-xSSE4.2` would be applicable to the Nehalem (NHM) microarchitecture.
- `-vec / -no-vec` - whether or not any generated code should be vectorized. `-vec` permits any amount of generated code to be vectorized while `-no-vec` prevents any code from being vectorized.

We used the command-line flags to drive native code generation for the target platform of execution.

2.2 Source Code Directives

In addition to command-line flags, the Intel Compiler is also capable of processing programmer-placed directives (in C/C++ written as `#pragma <opt>`). Many of these are specific to the Intel Compiler; however, some of them are gaining support in other compilers such as GCC [1].

Here, we will look at a subset of directives offered in the Intel Compiler, specifically those that aid with vectorized code transformations and generation.

- `#pragma vector always` - compiler will ignore the speed-up factor predicted by its internal model when considering vectorization. If the compiler believes the loop will execute slower with vectorization, it will still vectorize the code.
- `#pragma ivdep` - compiler will ignore all *unproven* inter-loop dependences. All *proven* dependences will not be vectorized. Unsafe code may be generated.
- `#pragma simd [vectorlength(n)]` - compiler will ignore all dependences and reductions. Everything related to vectorization is left for the programmer to manage. An optional argument for `simd` is to specify a vector length. This value is passed to the compiler to state how many safe iterations can be done at once. Unsafe code may be generated.

Such as in previous works [10], we used source code directives to drive the optimization selection and modification to show that we can guide the vectorization heuristics to improve performance.

3 Kernels

To evaluate the Intel Compiler's built-in vectorization heuristics, a set of benchmarks were used to determine performance improvement. Two sets of benchmarks were used for this work. The first set, the *Test Suite for Vectorizing Compilers* [20] is an extension and modification of a test suite for vectorizing Fortran compilers in the late 1980's [5]. The second set of kernels are the *PolyBench* kernels, which are kernels used in various publications on polyhedral compilers [27].

3.1 Test Suite for Vectorizing Compilers

The Test Suite for Vectorizing Compilers contains 151 different loop nests which iterate over different access patterns, computations, and memory access types (e.g. single value and (un)aliased pointers). The test suite was designed to evaluate how well compilers were able to recognize patterns which could be vectorized. For this work, we were using this test suite to evaluate and see which built-in heuristics in the Intel compiler may not enable the best performance while maintaining numerical correctness. By issuing varying directives, we were able to relax the Intel Compiler's built-in heuristics and instead see how applying optimizations, perhaps unsafely by the compiler's view, adjusts the performance of the loop nest. For sake of code re-use and to simplify generation and execution, each loop nest was placed in its own file.

autovec directive	Intel-specific pragma
`permute`	*generate each version*
`vl(x)`	`simd vectorlength(x)`
`always`	`vector always`
`ivdep`	`ivdep`
`none`	<nothing>

Table 1: `autovec` directive support and translation. `autovec` is capable of automatically expanding its directive clauses to Intel-specific pragmas.

3.2 PolyBench

PolyBench/C 3.2 contains 30 different static control-flow micro-benchmarks deriving from several scientific domains (e.g. linear algebra, machine learning, image processing). As with the test suite for vectorizing compilers, we have used these kernels to explore the potential increase in performance. We made minor source code changes for PolyBench to adapt the larger kernels to our machine learning model. First, for our graph-based characterization, only the relevant source code was used to generate our control flow graph (CFG). This was done manually by only specifying a single loop nest in a source code file. Second, we created n different versions of each PolyBench kernel, where n is determined by the number of individual loop nests which we elected to look at. Otherwise, no source code changes were made to the benchmark suite.

4 Version Generation

We developed two utilities in order to help with generation of different optimized versions of the code. `autovec` is a source-to-source compiler which translates a simplified directive language to a specific directive language supported by a given compiler (e.g. Intel Compiler). `VALT` performs exhaustive search space compilation among a set of optimizations to apply more than once in a given program. These utilities can help non-experts in the generation and analysis of programs, and have been used here to understand the inner workings of the compiler's vectorization strategies.

4.1 autovec

For version generation of each kernel, we used scripts with placeholder directives to drive which optimization was to be used on a per-loop basis. We defined our own directive language, we call `autovec`, to aid with this task. Table 1 shows the possible directives we can parse and generate with `autovec`. When `autovec` is given a *permute* argument before a loop, the tool will permute through all loop optimizations and generate a different version. As multiple loops are issued with this command, the number of possible versions of a single kernel grows exponentially. Table 4 shows versions of code which are (a) safe and adhere to the original implementation, (b) unsafe and cause a variance in result, and (c) invalid code which could not even compile.

VALT directive	Intel-specific pragma
vector(default)	<no code emitted>
vector(none)	novector
vector(always)	vector always
vector(ignore)	ivdep
vector(aligned)	vector aligned
vector(temp)	vector temporal
vector(nontemp)	vector nontemporal
vectorsize(x)	simd vectorlength(x)
loop(unroll(x))	unroll(x)
loop(jam(x))	unroll_and_jam(x)
loop(nofusion)	nofusion
loop(dist)	distribute_point

Table 2: VALT directive language translation for Intel-specific pragmas

	Nehalem (NHM)	Haswell (HSW)
Processor	i7-950	i7-5930K
Clock Rate	3.06GHz	3.50GHz
L3 Cache	8MB	15MB
Shared Memory	24GB DDR3-1333	32GB DDR4-2133
Vector Width	128 bit	256 bit
Extension Support	SSE 4.2	AVX 2

Table 3: Machine Configurations

4.2 VALT

An extension to autovec was also created with additional optimizations permitted. VALT (vectorization and loop transformation) enables a developer to quickly specify which vectorization and loop transformation directives to apply to a given loop. Backends exist for both Intel compiler (Intel-specific pragmas) and CAPS Compiler [6] (hmppcg directive support). Table 2 shows how VALT directives directly translate to Intel-specific pragmas. For this work we limited our use of VALT to only be used for vectorization.

5 Vectorization Analysis

5.1 Execution and Evaluation

Intel Compiler v15.0 was used for all experiments throughout this research. Table 3 shows the machine configuration used for all experiments. All time measurements were performed in either cycles or gettimeofday. Additionally, dynamic frequency scaling was disabled on all test platforms to mitigate timing errors. Speedups were measured as compared to the "default" optimization configuration: no added directives and compiled with -O3 -xHOST -vec. Intel vectorization reports (-vec-report6) were also obtained to aid in classification and to see why the compiler chose not to

⟨*directive*⟩ ::= '`#pragma`' '`VALT`' ⟨*clauselist*⟩

⟨*clauselist*⟩ ::= ⟨*clauselist*⟩ [[,]] ⟨*clause*⟩
  | ⟨*empty*⟩

⟨*clause*⟩ ::= '`vector`' '`(`' ⟨*vectorlist*⟩ '`)`'
  | '`depend`' '`(`' ⟨*dependopts*⟩ '`)`'
  | '`vectorsize`' '`(`' ⟨*number*⟩ '`)`'
  | '`loop`' '`(`' ⟨*looplist*⟩ '`)`'

⟨*vectorlist*⟩ ::= '`none`'
  | ⟨*vectorlist*⟩ '`,`' ⟨*vectoritem*⟩
  | ⟨*vectoritem*⟩

⟨*looplist*⟩ ::= ⟨*looplist*⟩ '`,`' ⟨*loopitem*⟩
  | ⟨*loopitem*⟩

⟨*vectoritem*⟩ ::= '`default`'
  | '`none`'
  | '`always`'
  | '`aligned`'
  | '`unaligned`'
  | '`nontemp`'
  | '`temp`'

⟨*loopitem*⟩ ::= '`unroll`' [['`(`' ⟨*number*⟩ '`)`']]
  | '`jam`' [['`(`' ⟨*number*⟩ '`)`']]
  | '`dist`'
  | '`nofusion`'

⟨*dependopts*⟩ ::= '`ignore`'
  | '`default`'

⟨*number*⟩ ::= [1-9][0-9]*

Figure 1: `VALT` directive language grammar

```
(17): loop was not vectorized: existence of vector dependence
(19): vector dependence: assumed ANTI dependence a(19) and a(18)
(18): vector dependence: assumed FLOW dependence a(18) and a(19)
(18): vector dependence: assumed FLOW dependence a(18) and a(19)
(19): vector dependence: assumed ANTI dependence a(19) and a(18)
(15): loop was not vectorized: not inner loop
(13): loop was not vectorized: not inner loop
```

Figure 2: A sample vectorization report

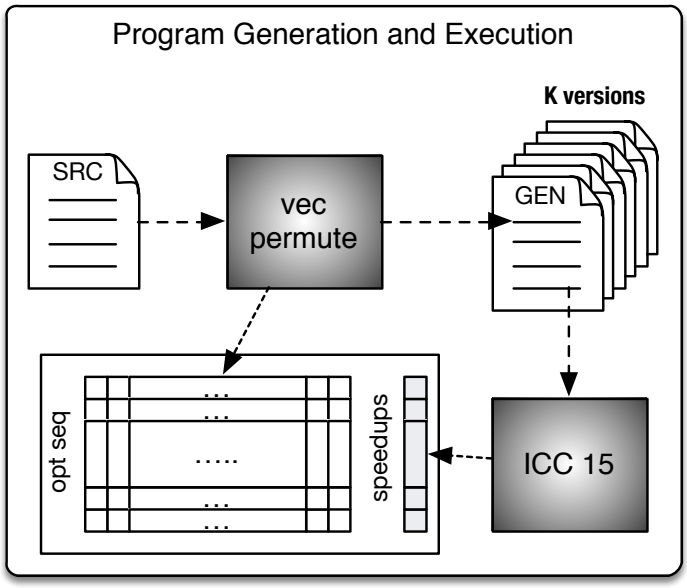


Figure 3: Workflow overview for version generation and compilation. `vecpermute` performs automatic optimization selection and version generation

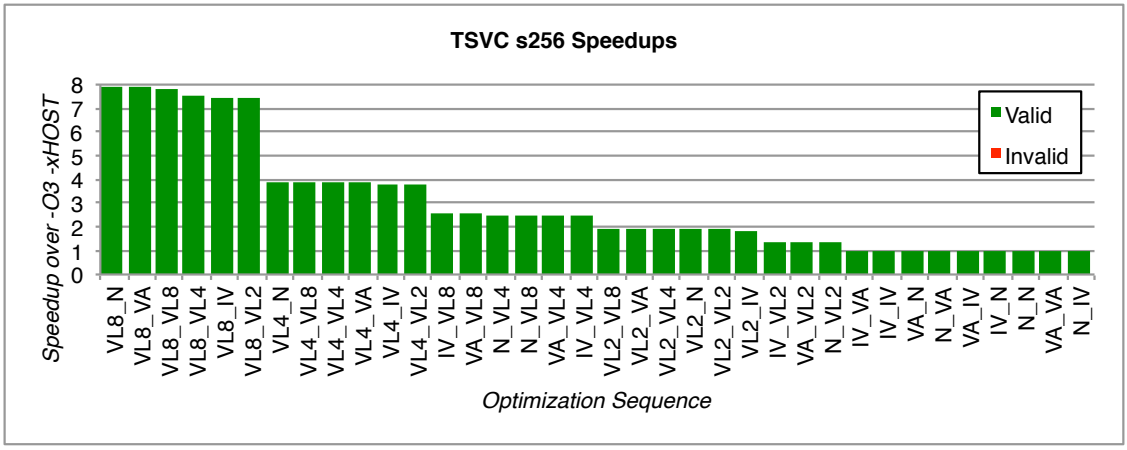
vectorize some loop nests. Figure 2 shows a sample vectorization report where the Intel Compiler did not vectorize the loop because of assumed vector dependences and being unable to vectorize non-inner loops. The original benchmark's vectorization report was compared to each new version's vectorization report.

We ran each program 10 times on a processor with dynamic frequency scaling turned off. In the case of TSVC loop nests, each loop nest was surrounded by an additional iteration loop ranging between 10000 and 20000 iterations. This was to help eliminate timing error. Execution time was recorded via `gettimeofday` for all TSVC loops. This was the default timing method used in the original benchmark suite. For PolyBench, the timing method was the total number of CPU clock cycles during kernel execution. This was the default timing method used in the original PolyBench suite. For cross-architecture analysis, we used correlation to evaluate how similar the speedup for a given benchmark running on one architecture carried over to another architecture. The equation for correlation is:

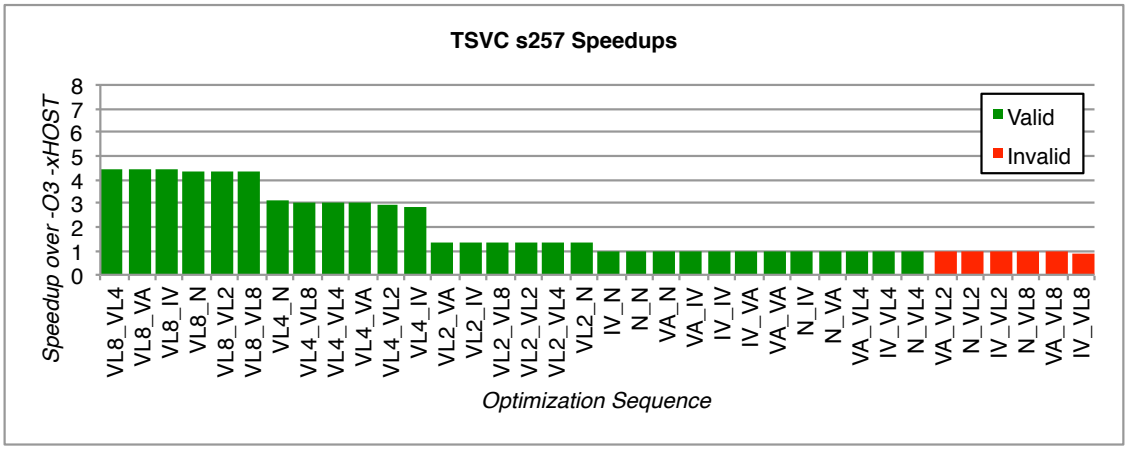
$$Correl(X,Y) = \frac{\sum (x-\bar{x})(y-\bar{y})}{\sqrt{\sum (x-\bar{x})^2 \sum (y-\bar{y})^2}}$$

5.2 Version Generation and Correctness

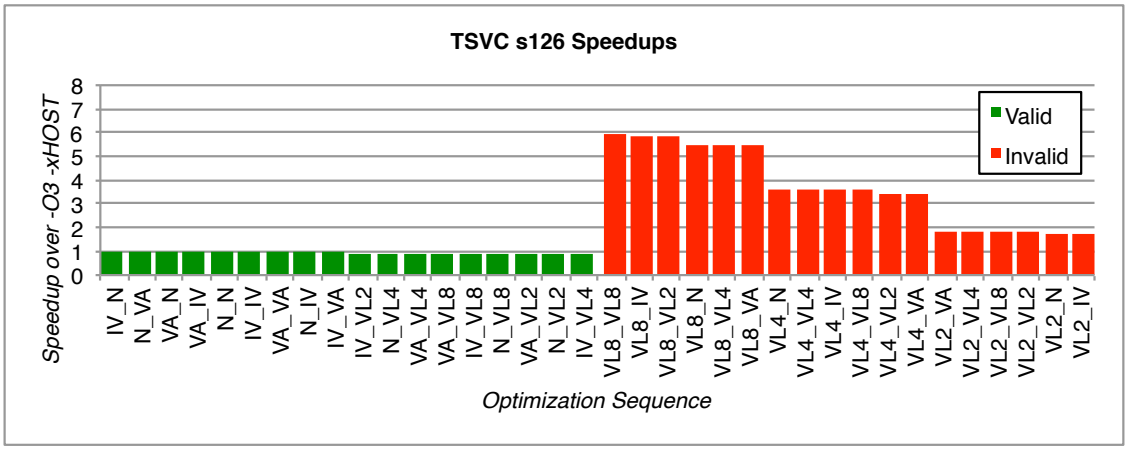
Figure 3 provides a workflow overview for version generation. Figure 5 shows sample performance analysis of three different loop nests and demonstrate varying levels of correctness. Figure 4a shows a loop nest where good speedup was observed with no invalid code generation. This is the ideal



(a) Good speedup observed; no invalid code generation. This is the ideal case. All searched optimizations produced valid code.



(b) Good speedup observed; invalid code generated. This case isn't ideal, but the advantage to this case is all invalid code generated is slower than the valid code.



(c) Minor speedup observed; invalid code faster. This is the worst case. All invalid code is faster than valid code.

Figure 4: TSVC loop nest optimizations and speedup comparison

Benchmark	Architecture	Valid	Invalid	Error
TSVC	Nehalem	1832	151	3
TSVC	Haswell	1826	155	5
PolyBench	Nehalem	5204	3826	0
PolyBench	Haswell	5204	3826	0

Table 4: Code Generation statistics for TSVC and PolyBench. Valid programs are those that produce the correct result. Invalid programs compile and run, but produce incorrect results. Programs classified under error either produce run-time errors or failed to compile.

Architecture	Valid Faster	Invalid Faster
Nehalem	140	11
Haswell	143	8

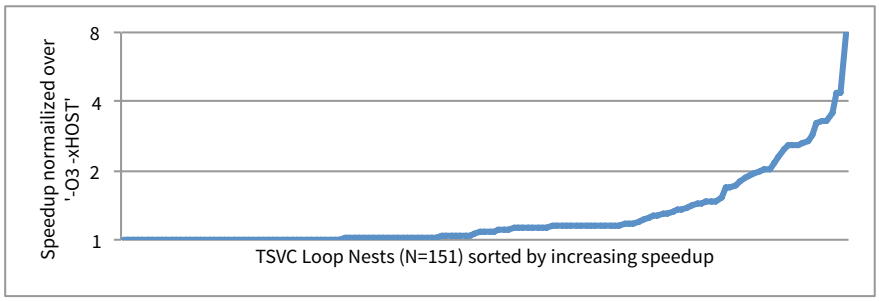
Table 5: Exhaustive search speedup overview for TSVC. For each loop nest, the fastest speedup observed produced correct or incorrect results. This is to illustrate the significance that valid code generation is important.

case. Figure 4b shows a loop nest with good speedup observed, but code with a speedup observed can still be invalid. Likewise, invalid code generation can occur with no speedup observed with valid code generation as shown in Figure 4c. Sometimes no variation with valid code generation produced a speedup. For loops which saw low (or no) speedup improvements, there were a few varying reasons why. First, the compiler may have already internally applied the best vectorization optimizations that were in our search space. Second, there was cache line contention for some of the memory-intensive loop nests. Third, some loops are only able to be vectorized through manual code modifications.

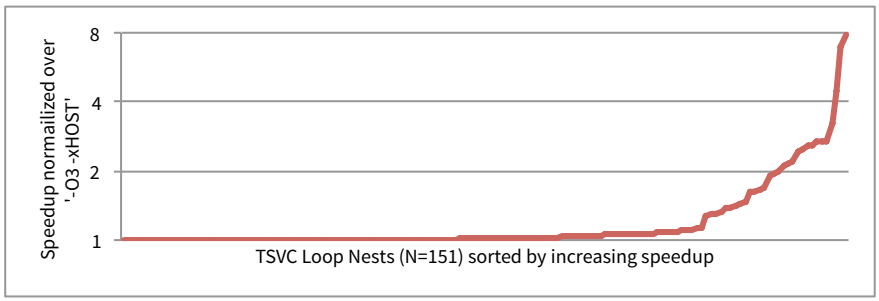
5.3 TSVC

Figure 5 shows the results of exhaustive exploration for speedup for PolyBench loop nests running on the Nehalem and Haswell architectures. For each graph shown, each x-coordinate corresponds to a different loop nest, while the y-coordinate corresponds to the best speedup observed for the loop nest. Figure 5a and Figure 5b depict speedup curves for each architecture. Nehalem was able to see improved speedup for more benchmarks than Haswell; however, the maximum speedup for each architecture were nearly the same. Figures 5c and 5d depict cross-architecture curves for the TSVC loop nests.

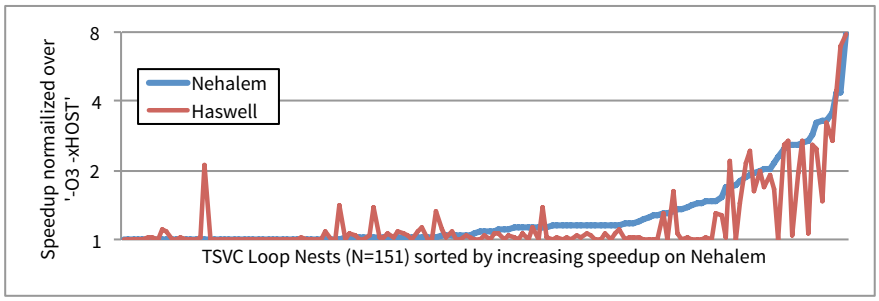
The speedup curves for both Nehalem and Haswell appear similar when each are sorted from lowest to highest (Figures 5a and 5b. However, when performing cross-architecture comparison, there are a few observations that can be noted in Figures 5c and 5d. First, the best speedup observed for each architecture is not only from the same benchmark, but also with a similar speedup. Second, Although there are discrepancies with speedups across the different architectures, we see some correlation between the two curves, particularly within the first 40 benchmarks. The correlation coefficient between the TSVC loop nests across Nehalem and Haswell architectures is 0.8945.



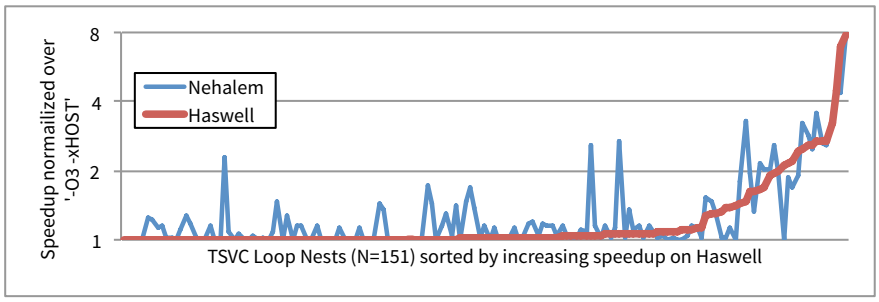
(a) Nehalem



(b) Haswell



(c) Cross Architecture sorted by Nehalem



(d) Cross Architecture sorted by Haswell

Figure 5: Speedup of TSVC loop nests on Nehalem and Haswell.

Bit Configuration	Optimization
000000	No Loop
100000	No Optimization
110000	`#pragma vector always`
111000	`#pragma ivdep`
111100	`#pragma simd vectorlength(2)`
111110	`#pragma simd vectorlength(4)`
111111	`#pragma simd vectorlength(8)`

Table 6: Optimization bit vector configuration

5.4 PolyBench

Figure 6 shows the results of exhaustive exploration for speedup for PolyBench loop nests running on the Nehalem and Haswell architectures. For each graph shown, each x-coordinate corresponds to a different loop nest, while the y-coordinate corresponds to the best speedup observed for the loop nest. Figure 6a and Figure 6b depict speedup curves for each architecture. Similarly to the TSVC loop nests, the PolyBench loop nests have improved speedup for more benchmarks than Haswell; however, there were a few notable differences. First, Haswell saw a much higher peak speedup than Nehalem. Second, for loop nests that had a speedup, those running on Haswell saw a much higher speedup compared to Nehalem.

Figure 6c and Figure 6d depict cross-architecture curves for the PolyBench loop nests. We observe that both architectures had some benchmarks achieving over an 800% speedup over the default `'-O3 -xHOST'` baseline, though half of the benchmarks had a speedup less than 10%. The 11 top benchmarks running on each architecture exhibited similar trends with regard to speedup. Results from Nehalem act as a good overpredictor for potential speedup observed on Haswell, as indicated in Figure 6c. The correlation coefficient between the PolyBench loop nests across Nehalem and Haswell architectures is 0.8894.

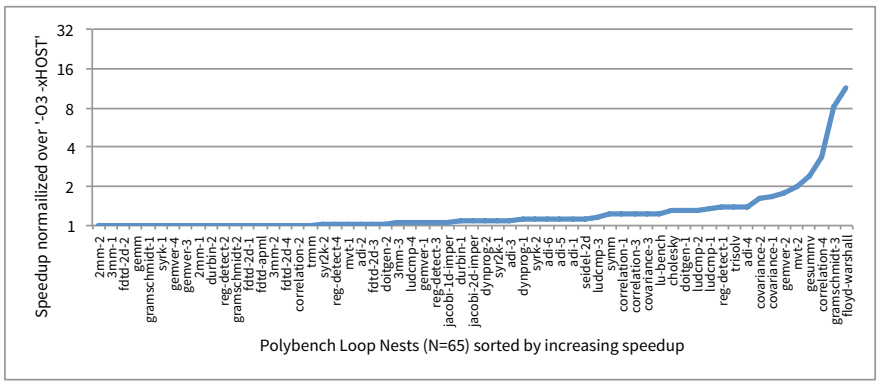
6 Graph-Based Speedup Predictor

Given the speedup information based on different optimizations for a collection of loop nests and kernels, an end goal would be to automate the prediction of the speedup of a benchmark for a fixed input and the optimization sequence targeted to a particular code.

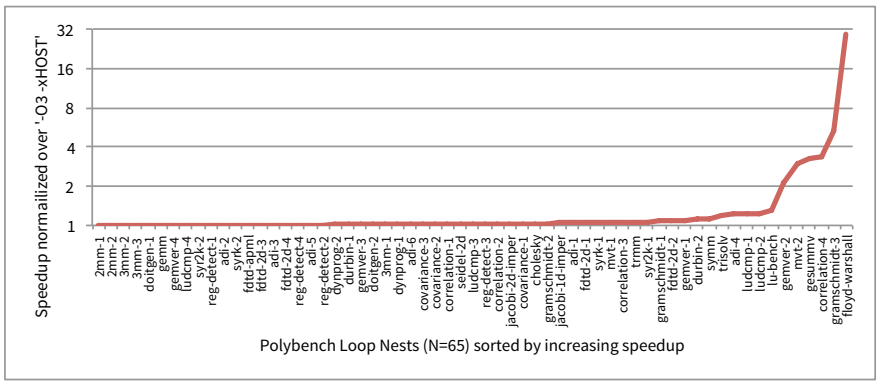
6.1 Experiment Configuration

There are two main inputs feeding into our graph-based predictor. The first is an optimization sequence which is represented as an optimization bit vector. The second input is a graph-based representation of the code being optimized.

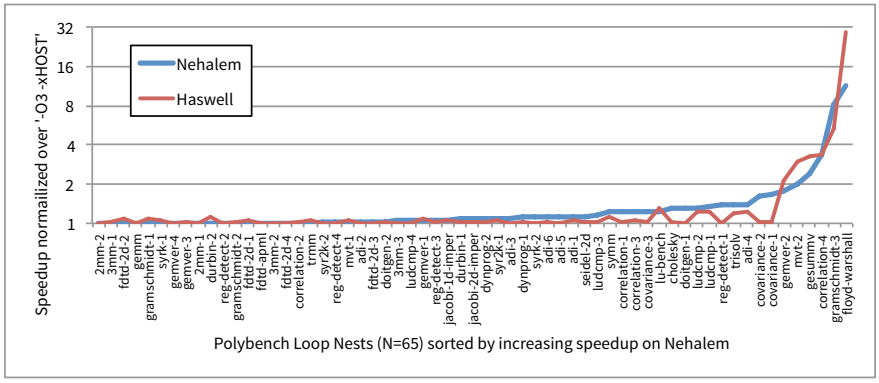
An optimization bit vector is a vector (or array) that has different field values that can be set or unset (0 or 1). The length of this vector determines how many possible combinations of



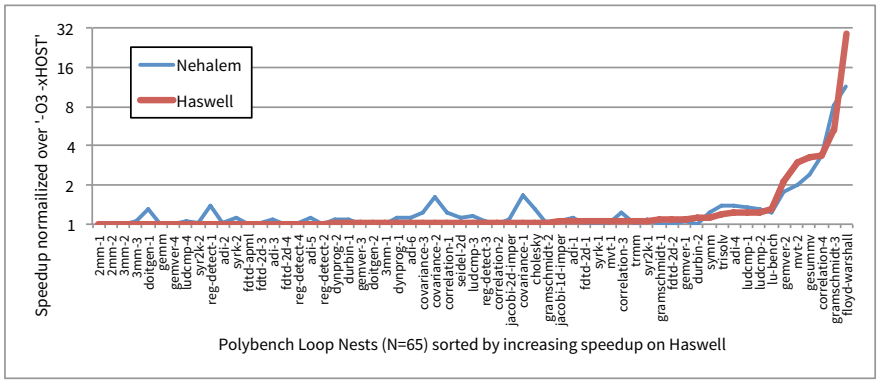
(a) Nehalem



(b) Haswell



(c) Cross Architecture sorted by Nehalem



(d) Cross Architecture sorted by Haswell

Figure 6: Speedup of PolyBench loop nests on Nehalem and Haswell.

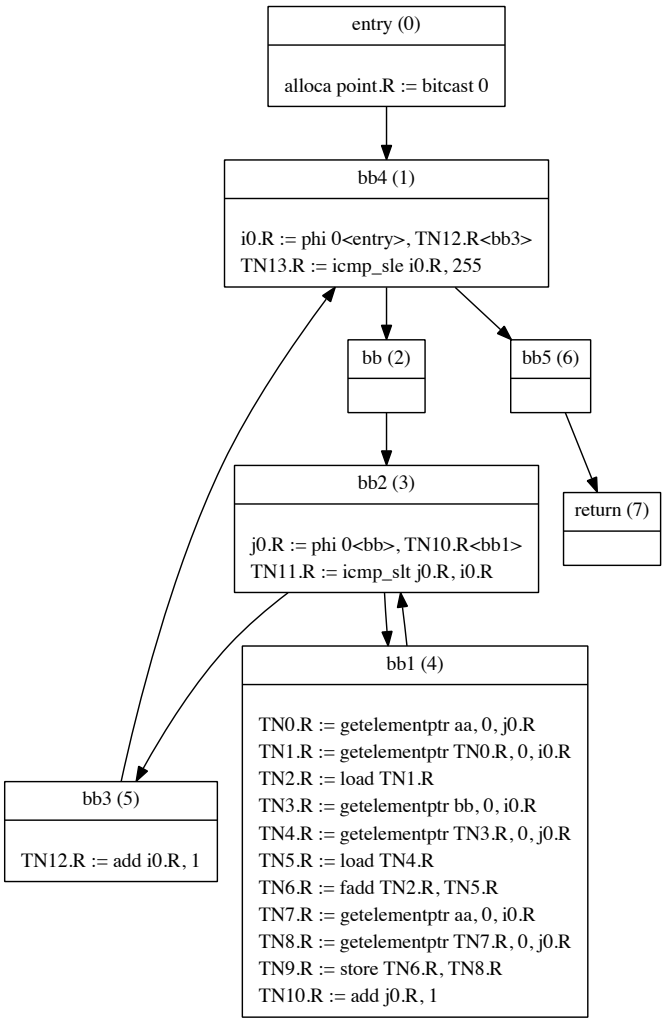


Figure 7: Sample control flow graph with LLVM IR basic blocks

optimizations can exist. For this work, our bit vector has a fixed length of six. Table 6 describes the optimization bit vector values and which optimization they represent. The zeroth bit indicates if there is a loop present. The first through sixth bits can additionally be set to indicate the level of optimization to apply. The optimizations are ordered in such a way that as you apply an additional optimization, the potential for generating incorrect code can increase.

Multiple loops for a given code would increase the number of optimization bit vectors. Because of potential ambiguity, we elected to have the optimizations listed in source code line order. For the case of TSVC loop nests and Polybench loop nests, no loop nest exists for size greater than 4. To simplify training and classification, any loop nests of size less than 4 were padded with 0 indicating no further optimization to be applied. Table 3 presents the machine configuration we used to collect all training data for prediction.

Feat. #	Feature Description
ft1	Number of Instructions
ft2	Number of Add instruction
ft3	Number of Sub instruction
ft4	Number of Mult instruction
ft5	Number of Div instruction
ft6	Number of Load instruction
ft7	Number of Store instruction
ft8	Number of Comparisons
ft9	Number of Conditional Branches
ft10	Number of Unconditional Branches

Table 7: Feature Vector description used for graph-based characterization

Figure 8a shows the feature extraction workflow used to construct our training data. We use MinIR [2] to extract control flow graphs (CFGs) from each of our programs. A sample CFG from the program s114 from the TSVC benchmark suite can be seen in Figure 7. From the CFG we generate a graph-based characterization which includes a feature vector for each basic block in the CFG and a list of directed edges in the graph. The feature vector we generate is shown in Table 7.

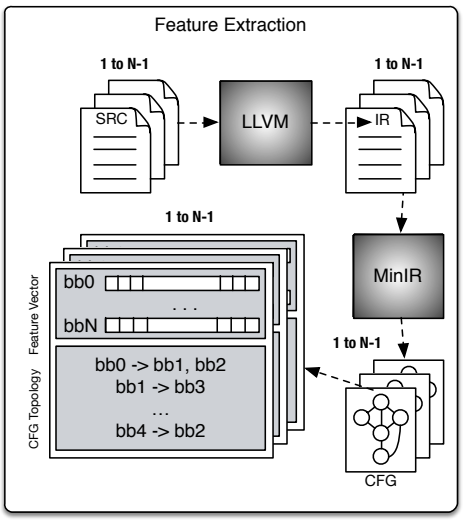
Once the training data is constructed, it can be fed to a learning algorithm. This learning algorithm produces a machine learning model. The workflow of producing the predictor is shown in Figure 8b. We used support vector machines (SVMs) to construct our predictive models. SVMs are a class of machine learning algorithms that can be used for both classification and regression. In our work, we use SVMs for regression as we are trying to predict speedups. Our SVM model uses *graph kernel* functions to transform the training data (control flow graphs) into a different, linearly-separable feature space. Then a linear classifier is constructed that separates the points into multiple classes. For this work we used the same shortest path graph kernel used by Park et. al. [24]

Finally, once we have a predictor constructed, we can evaluate it on unseen programs. Figure 8c shows the workflow of how the predictor can emit speedup predictions by providing as input the graph-based representation of a program with an optimization. The output of our predictor is a speedup value.

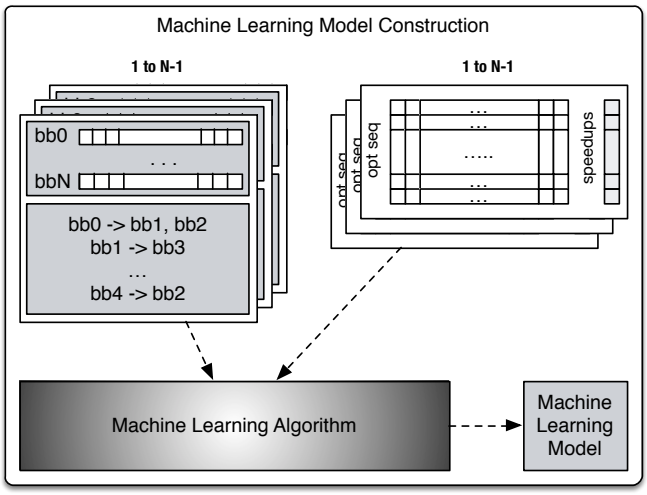
6.2 Prediction Results and Evaluation

We tested our trained models with leave one out cross validation (LOOCV). For a given loop nest found in the TSVC set, we constructed a model based on the other 150 loop nests and compared the model's prediction to the actual speedup observed for the model. The same procedure was done for all other loop nests. For a given loop nest found in the PolyBench loop nests set, we constructed a model based on the other 64 loop nests and compared the model's prediction to the actual speedup observed for the model.

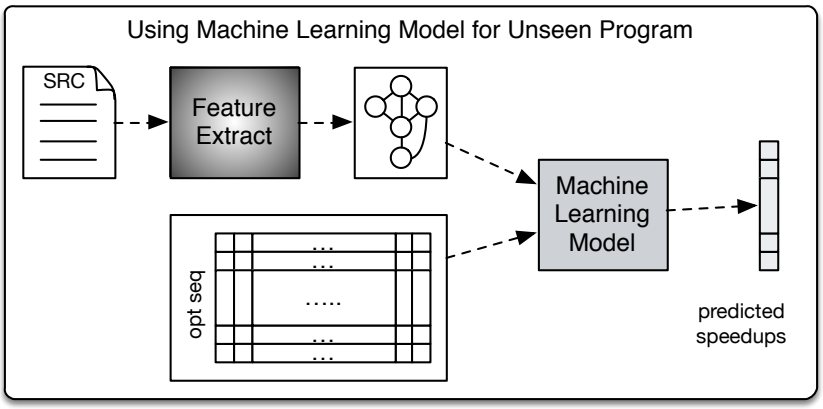
We have two different evaluation methods for our speedup predictor: 1-shot and 3-shot evalu-



(a) Workflow overview for graph-based feature extraction. MinIR is used to generated a control flow graph (CFG). Basic blocks are then annotated with additional features.



(b) Workflow overview for predictor model construction. We use the graph-based characteristics generated earlier as training data for a support vector machine.



(c) Workflow overview for speedup prediction using the constructed predictor model. We pass in the graph-based representation along with an optimization bit vector to predict a speedup.

Figure 8: Overall Predictor Workflow

ation. For the 1-shot model, we only consider the top prediction sorted by the optimization with the best observed speedup. For the 3-shot model, we consider the top 3 predictions sorted by the optimizations with the best observed speedup. A 3-shot evaluation will produce at least as good and most likely better results compared to 1-shot evaluation.

Tables 8, 9, and 10 list the predictor results for the TSVC loop nests for Nehalem and Haswell architectures. Many of the loop nests resulted in a predicted slowdown for both the 1-shot and 3-shot evaluation. To summarize our results, both arithmetic and geometric means were calculated. The predictor was more accurate when targeting the Haswell architecture. We were able to correctly predict speedup using the 3-shot model within 77% of optimal on Haswell (using arithmetic mean) and 74.3% of optimal on Nehalem. The 1-shot model was within 50.5% of optimal on Haswell and 54.2% of optimal on Nehalem.

Table 11 lists the predictor results for the PolyBench loop nests for Nehalem and Haswell architectures. Once again, when targeting the Haswell architecture, the predictor had better prediction results 88.7% of optimal on Haswell and 84.4% of optimal on Nehalem. Even when looking at the 1-shot evaluation, we observe far better results of the PolyBench loop nests when compared to the TSVC loop nests. There were some strange results with the PolyBench loop nests. Specifically, `2mm` and `3mm` all recorded slowdowns for the predictor when the CFG for each of the pairing loop nests in the training data were the same as the unseen program.

7 Threats to Validity

Threats of this work can be broken down into the following sections: Section 7.1 addresses threats related to program and optimization selection, Section 7.2 addresses threats related to program execution, and Section 7.3 addresses threats related to the machine learning model used in this work.

7.1 Program and Optimization Selection

The selection of the programs we used were based on prior research done in related research areas. PolyBench has previously been used for optimization selection for polyhedral compilers including autovectorization and autoparallelization code emission. The TSVC loop nests have been used to evaluate the vectorization performance of compilers. A natural extension to evaluating the compiler's ability to automatically vectorize loop nests is by providing additional hints and suggestions to the compiler. That's why we used the directive-based optimizations. The Intel Compiler provided many different optimizations that can affect vectorization. We chose the five optimizations (`vector always`, `ivdep`, `simd vectorlength(2)`, `simd vectorlength(4)`, and `simd vectorlength(8)`) that we did for the following reasons:

- `vector always` - overrides the built-in checking to determine benefit of vectorization. Useful to see if/when the compiler doesn't choose the best option

loop	Nehalem 1-shot	Nehalem 3-shot	Nehalem OPT	Haswell 1-shot	Haswell 3-shot	Haswell OPT
s000	0.65 (65.46%)	0.99 (99.02%)	1.00	0.77 (75.80%)	0.97 (95.46%)	1.02
s1111	0.66 (66.78%)	0.87 (87.55%)	1	0.93 (93.76%)	0.96 (96.44%)	1
s1112	0.67 (58.32%)	0.98 (85.12%)	1.15	0.84 (79.24%)	1.00 (94.21%)	1.06
s1113	0.99 (31.02%)	1.90 (59.55%)	3.20	1.00 (40.06%)	1.78 (71.66%)	2.49
s1115	0.94 (71.12%)	0.95 (71.92%)	1.32	0.95 (58.09%)	1.00 (61.02%)	1.63
s1119	1.00 (48.86%)	1.00 (48.93%)	2.04	0.98 (51.78%)	1.00 (52.64%)	1.90
s111	0.63 (63.43%)	0.66 (66.63%)	1	0.39 (39.43%)	0.77 (77.87%)	1
s112	0.78 (42.01%)	1.00 (53.67%)	1.86	0.96 (44.71%)	1.00 (46.51%)	2.15
s113	0.56 (50.23%)	0.98 (88.12%)	1.12	0.67 (66.23%)	0.98 (97.06%)	1.01
s114	0.55 (47.74%)	0.55 (47.80%)	1.15	0.64 (57.39%)	0.64 (57.74%)	1.11
s115	0.10 (8.84%)	0.10 (8.84%)	1.14	0.03 (3.68%)	0.03 (3.68%)	1.02
s1161	0.95 (73.37%)	1.00 (76.65%)	1.30	0.51 (51.19%)	0.73 (72.52%)	1.00
s116	0.95 (53.93%)	1.00 (56.22%)	1.77	0.41 (28.47%)	1.00 (69.47%)	1.44
s118	0.42 (20.14%)	0.42 (20.19%)	2.09	0.26 (17.65%)	0.26 (17.65%)	1.52
s119	0.18 (16.28%)	0.18 (16.29%)	1.13	0.18 (16.90%)	0.18 (16.90%)	1.11
s1213	0.11 (11.72%)	0.35 (35.80%)	1	0.08 (8.87%)	0.31 (31.05%)	1
s121	0.78 (78.33%)	0.98 (98.20%)	1.00	0.59 (59.55%)	0.94 (94.60%)	1
s1221	0.47 (38.29%)	0.99 (80.51%)	1.23	0.40 (40.29%)	0.75 (75.09%)	1
s122	0.68 (68.20%)	0.99 (97.98%)	1.01	0.53 (53.78%)	0.97 (97.87%)	1
s1232	0.99 (52.07%)	0.99 (52.27%)	1.90	0.93 (38.54%)	0.93 (38.84%)	2.41
s123	0.87 (87.78%)	1.00 (100.00%)	1	1.00 (99.96%)	1.00 (100.00%)	1
s1244	0.21 (21.07%)	0.50 (50.29%)	1	0.18 (18.57%)	0.57 (56.35%)	1.01
s124	0.79 (79.23%)	0.99 (99.63%)	1	0.90 (90.23%)	1.00 (99.80%)	1.00
s1251	0.81 (81.01%)	1.00 (99.41%)	1.00	0.89 (89.02%)	1.00 (99.49%)	1.00
s125	0.36 (36.22%)	0.36 (36.30%)	1.00	0.22 (22.40%)	0.22 (22.42%)	1.00
s126	1.91 (30.53%)	1.92 (30.70%)	6.28	1.74 (29.33%)	1.84 (30.89%)	5.95
s1279	0.80 (65.78%)	1.00 (81.34%)	1.22	0.74 (74.19%)	0.93 (93.08%)	1
s127	0.71 (71.21%)	0.98 (98.06%)	1	0.71 (70.33%)	0.99 (97.80%)	1.02
s1281	0.76 (76.49%)	1.00 (99.60%)	1.00	0.84 (84.68%)	0.99 (99.09%)	1.00
s128	0.96 (95.90%)	0.98 (97.51%)	1.01	0.95 (68.14%)	0.99 (71.32%)	1.39
s13110	0.49 (42.44%)	0.49 (42.53%)	1.17	0.26 (26.08%)	0.26 (26.27%)	1.02
s131	0.78 (68.64%)	1.00 (87.18%)	1.14	0.63 (59.78%)	1.00 (94.89%)	1.05
s132	0.35 (30.84%)	1.00 (86.94%)	1.15	0.31 (31.66%)	1.00 (99.93%)	1.00
s1351	0.81 (74.27%)	0.99 (90.62%)	1.09	0.94 (88.45%)	1.00 (93.97%)	1.06
s141	0.68 (60.87%)	0.69 (61.08%)	1.12	0.70 (70.87%)	0.72 (72.22%)	1
s1421	0.80 (70.23%)	0.97 (85.32%)	1.14	0.75 (72.39%)	0.97 (93.37%)	1.04
s151	0.76 (69.00%)	1.00 (90.46%)	1.10	0.62 (60.58%)	1.00 (96.37%)	1.03
s152	0.99 (51.64%)	1.50 (77.78%)	1.93	0.99 (60.66%)	1.00 (61.07%)	1.63
s161	0.63 (54.99%)	1.00 (87.35%)	1.14	0.40 (40.51%)	0.77 (77.27%)	1
s162	0.82 (81.10%)	0.99 (98.07%)	1.01	0.74 (71.81%)	1.00 (96.57%)	1.03
s171	0.76 (75.36%)	0.98 (97.23%)	1.01	0.70 (65.75%)	1.00 (92.90%)	1.07
s172	0.77 (73.44%)	1.00 (95.43%)	1.04	0.68 (66.69%)	1.00 (96.77%)	1.03
s173	0.81 (78.48%)	1.00 (96.87%)	1.03	0.72 (65.17%)	1.03 (93.58%)	1.10
s174	0.81 (79.50%)	0.99 (97.21%)	1.02	0.68 (63.44%)	1.00 (93.07%)	1.07
s175	0.78 (75.68%)	0.98 (95.31%)	1.03	0.62 (58.15%)	1.00 (93.54%)	1.07
s176	0.38 (20.79%)	0.38 (20.88%)	1.85	0.34 (31.11%)	0.34 (31.25%)	1.11
s2101	0.84 (84.87%)	0.90 (90.68%)	1	0.64 (64.72%)	0.89 (89.14%)	1.00
s2111	0.29 (25.96%)	0.29 (26.07%)	1.12	0.30 (28.64%)	0.30 (28.70%)	1.05
s211	0.35 (25.61%)	0.91 (65.86%)	1.39	0.23 (23.49%)	0.87 (86.70%)	1.01
s212	0.18 (18.41%)	0.47 (46.69%)	1.01	0.13 (13.76%)	0.51 (50.75%)	1.00
s2192	0.04 (4.70%)	0.04 (4.74%)	1	0.04 (4.13%)	0.04 (4.15%)	1.10
s221	0.39 (27.84%)	1.42 (100.00%)	1.42	0.36 (27.09%)	1.35 (100.00%)	1.35
s222	0.51 (30.37%)	0.98 (57.67%)	1.70	0.58 (26.33%)	1.00 (45.38%)	2.20
s2233	0.13 (12.87%)	0.13 (13.21%)	1.01	0.11 (11.20%)	0.11 (11.22%)	1.02
s2244	0.87 (85.68%)	0.99 (97.37%)	1.02	0.49 (49.64%)	0.83 (83.49%)	1.00
s2251	2.03 (83.23%)	2.44 (100.00%)	2.44	1.49 (90.99%)	1.64 (100.00%)	1.64
s2275	0.09 (9.41%)	0.09 (9.44%)	1.00	0.05 (5.27%)	0.05 (5.45%)	1.00
s231	0.14 (7.19%)	0.14 (7.26%)	2.01	0.09 (5.75%)	0.09 (5.77%)	1.69
s232	0.55 (17.08%)	0.55 (17.08%)	3.27	0.41 (28.06%)	0.41 (28.11%)	1.46

Table 8: Leave One Out Cross Validation Results for TSVC. Nehalem and Haswell are the target architectures, 1-shot refers to a 1-shot predictor, 3-shot refers to a 3-shot predictor, and OPT refers to the optimal speedup for a given loop nest.

loop	Nehalem 1-shot	Nehalem 3-shot	Nehalem OPT	Haswell 1-shot	Haswell 3-shot	Haswell OPT
s233	0.12 (11.07%)	0.12 (11.15%)	1.14	0.11 (11.33%)	0.11 (11.38%)	1.00
s235	0.08 (8.24%)	0.08 (8.37%)	1.01	0.05 (5.20%)	0.05 (5.22%)	1
s241	0.19 (18.94%)	0.54 (53.93%)	1.01	0.13 (13.30%)	0.47 (47.35%)	1
s242	0.40 (27.11%)	1.00 (67.51%)	1.48	0.37 (29.15%)	0.91 (70.46%)	1.29
s243	0.23 (23.43%)	0.59 (59.40%)	1.00	0.18 (17.96%)	0.60 (60.73%)	1
s244	0.46 (33.77%)	1.36 (100.00%)	1.36	0.32 (24.67%)	1.18 (90.31%)	1.30
s251	0.81 (72.56%)	0.99 (88.66%)	1.12	0.88 (88.31%)	0.99 (98.80%)	1.00
s252	0.70 (68.73%)	1.00 (97.94%)	1.02	0.37 (35.91%)	0.98 (95.53%)	1.03
s253	0.68 (56.93%)	0.95 (78.63%)	1.21	0.64 (62.47%)	1.00 (97.51%)	1.02
s254	0.73 (72.99%)	0.95 (95.83%)	1	0.55 (55.81%)	0.98 (98.53%)	1.00
s255	0.99 (30.30%)	1.14 (34.78%)	3.29	0.98 (30.73%)	1.00 (31.18%)	3.20
s256	0.98 (12.45%)	0.99 (12.50%)	7.92	0.99 (12.64%)	1.00 (12.67%)	7.88
s257	0.99 (22.70%)	1.07 (24.51%)	4.36	0.93 (21.01%)	0.95 (21.36%)	4.45
s258	1.00 (22.81%)	1.00 (22.87%)	4.38	1.00 (14.54%)	1.00 (14.64%)	6.87
s261	0.21 (21.86%)	0.48 (48.69%)	1	0.15 (15.40%)	0.58 (57.94%)	1.00
s2710	0.54 (54.22%)	1.00 (99.28%)	1.00	0.55 (55.26%)	1.00 (99.94%)	1.00
s2711	0.75 (44.84%)	1.00 (59.33%)	1.68	0.60 (59.80%)	0.96 (95.54%)	1.01
s2712	0.75 (74.37%)	1.00 (97.99%)	1.02	0.73 (69.12%)	1.00 (94.18%)	1.06
s271	0.74 (50.33%)	1.00 (68.04%)	1.47	0.62 (61.22%)	0.97 (95.86%)	1.01
s272	1.00 (38.80%)	1.11 (43.42%)	2.57	0.67 (64.27%)	1.00 (95.86%)	1.04
s273	0.65 (59.13%)	1.00 (90.64%)	1.10	0.61 (61.61%)	0.88 (88.32%)	1
s274	0.60 (46.75%)	1.00 (77.88%)	1.28	0.50 (50.36%)	0.99 (99.21%)	1
s275	0.13 (9.98%)	0.13 (10.07%)	1.35	0.06 (5.65%)	0.06 (5.74%)	1.06
s276	0.76 (75.25%)	0.99 (98.24%)	1.01	0.62 (61.15%)	0.99 (97.87%)	1.01
s277	1.00 (37.50%)	1.00 (37.54%)	2.66	0.62 (59.24%)	1.00 (94.42%)	1.05
s278	0.65 (56.40%)	1.00 (86.23%)	1.16	0.71 (71.79%)	0.99 (99.59%)	1
s279	0.47 (45.26%)	1.00 (96.32%)	1.03	0.46 (46.43%)	0.99 (98.00%)	1.01
s281	1.00 (50.79%)	1.38 (70.21%)	1.96	1.00 (50.22%)	1.36 (68.77%)	1.99
s291	0.66 (65.20%)	1.00 (97.70%)	1.02	0.55 (55.85%)	0.96 (96.91%)	1
s292	0.69 (65.95%)	1.00 (95.13%)	1.05	0.30 (29.80%)	0.97 (94.72%)	1.03
s293	1.00 (27.79%)	1.50 (41.73%)	3.59	1.00 (37.55%)	1.86 (70.10%)	2.66
s3110	0.99 (86.80%)	0.99 (86.95%)	1.14	0.98 (98.51%)	0.99 (99.85%)	1
s31111	0.99 (77.21%)	1.00 (77.73%)	1.28	0.74 (74.19%)	1.00 (99.99%)	1
s3111	0.45 (45.43%)	0.90 (90.27%)	1.00	0.26 (26.82%)	0.91 (91.60%)	1
s3112	1.99 (49.34%)	4.03 (100.00%)	4.03	1.99 (44.21%)	4.52 (100.00%)	4.52
s3113	0.50 (50.15%)	0.84 (83.69%)	1.00	0.26 (26.65%)	0.98 (98.16%)	1
s311	0.64 (64.81%)	0.81 (81.47%)	1	0.33 (32.65%)	0.83 (83.07%)	1.01
s312	0.49 (22.69%)	2.19 (100.00%)	2.19	0.26 (26.31%)	0.99 (99.74%)	1
s313	0.75 (74.19%)	1.02 (100.00%)	1.02	0.45 (45.17%)	0.96 (96.91%)	1
s314	0.50 (36.79%)	1.00 (73.22%)	1.36	0.26 (26.28%)	0.99 (99.36%)	1.00
s315	0.46 (46.17%)	0.94 (93.96%)	1	0.21 (21.36%)	0.76 (76.86%)	1
s316	0.49 (33.99%)	1.00 (69.05%)	1.44	0.25 (25.83%)	0.99 (99.59%)	1.00
s317	0.49 (21.79%)	0.99 (43.65%)	2.28	0.26 (26.01%)	0.99 (99.27%)	1
s318	0.39 (36.69%)	1.00 (92.68%)	1.08	0.23 (23.17%)	0.83 (83.11%)	1
s319	0.87 (87.46%)	0.99 (99.57%)	1	0.86 (85.95%)	0.99 (99.28%)	1
s321	0.50 (23.49%)	0.97 (45.61%)	2.13	0.44 (26.66%)	1.00 (60.48%)	1.65
s322	0.67 (25.89%)	1.00 (38.52%)	2.59	0.51 (26.61%)	1.00 (51.96%)	1.92
s323	0.39 (25.80%)	1.00 (65.40%)	1.52	0.34 (27.16%)	1.00 (78.32%)	1.27
s3251	0.35 (24.73%)	0.99 (70.16%)	1.42	0.26 (26.31%)	0.75 (74.38%)	1.00
s331	0.48 (45.48%)	0.99 (92.68%)	1.07	0.42 (42.72%)	0.83 (83.17%)	1
s332	0.99 (99.93%)	1.00 (99.97%)	1	1.00 (99.86%)	1.00 (99.97%)	1.00
s341	1.00 (99.83%)	1.00 (100.00%)	1.00	0.98 (97.38%)	1.00 (100.00%)	1.00
s342	1.88 (100.00%)	1.88 (100.00%)	1.88	0.84 (83.96%)	1.00 (99.42%)	1.00
s343	0.99 (99.38%)	1.00 (100.00%)	1	1.00 (98.08%)	1.02 (100.00%)	1.02
s351	0.52 (52.26%)	0.56 (55.98%)	1.00	0.32 (32.57%)	0.61 (60.84%)	1.00
s352	1.00 (70.75%)	1.00 (70.75%)	1.42	0.49 (40.12%)	1.24 (100.00%)	1.24
s353	0.81 (81.09%)	0.82 (82.67%)	1	0.97 (46.59%)	1.00 (47.68%)	2.09
s4112	0.88 (86.50%)	1.00 (97.37%)	1.02	0.98 (73.45%)	1.00 (74.96%)	1.33
s4113	0.99 (87.85%)	1.07 (94.39%)	1.13	0.70 (70.24%)	0.98 (98.12%)	1.00

Table 9: Continued Leave One Out Cross Validation Results for TSVC.

loop	Nehalem 1-shot	Nehalem 3-shot	Nehalem OPT	Haswell 1-shot	Haswell 3-shot	Haswell OPT
s4114	0.85 (85.11%)	0.97 (97.32%)	1	0.86 (86.73%)	0.97 (97.65%)	1
s4115	0.90 (90.28%)	0.99 (99.17%)	1.00	0.97 (69.43%)	1.39 (100.00%)	1.39
s4116	0.90 (89.13%)	0.99 (98.26%)	1.01	0.97 (70.43%)	1.00 (72.42%)	1.38
s4117	0.57 (57.42%)	0.84 (84.86%)	1	0.59 (59.72%)	1.00 (99.73%)	1.00
s4121	0.79 (79.58%)	0.96 (96.28%)	1	0.81 (81.33%)	0.95 (95.82%)	1.00
s421	1.00 (40.69%)	1.89 (77.22%)	2.45	0.99 (38.26%)	1.76 (67.61%)	2.60
s422	0.99 (38.24%)	1.89 (72.38%)	2.61	1.00 (36.88%)	1.76 (64.98%)	2.71
s423	0.99 (38.67%)	1.90 (74.19%)	2.57	0.99 (36.85%)	1.77 (65.53%)	2.71
s424	0.99 (34.63%)	1.94 (67.24%)	2.88	1.00 (38.83%)	1.77 (69.05%)	2.57
s431	0.66 (66.10%)	0.87 (87.60%)	1	0.62 (62.75%)	0.90 (90.92%)	1
s441	0.47 (45.57%)	1.00 (95.53%)	1.04	0.38 (38.70%)	0.99 (99.75%)	1
s442	0.47 (41.64%)	1.00 (87.39%)	1.14	0.32 (31.55%)	1.00 (98.33%)	1.01
s443	0.58 (58.00%)	0.87 (87.54%)	1	0.66 (65.92%)	0.98 (98.58%)	1.00
s451	0.51 (45.18%)	1.00 (87.48%)	1.14	0.32 (31.28%)	1.00 (97.11%)	1.03
s452	0.78 (77.86%)	0.99 (98.82%)	1.00	0.83 (83.93%)	0.96 (96.54%)	1
s453	0.51 (47.97%)	0.99 (91.63%)	1.08	0.57 (55.55%)	0.99 (95.65%)	1.04
s471	0.85 (84.31%)	1.00 (98.46%)	1.01	0.81 (81.43%)	0.99 (99.62%)	1
s481	0.99 (87.27%)	1.00 (87.73%)	1.14	0.99 (99.89%)	0.99 (99.91%)	1
s482	1.00 (99.76%)	1.00 (100.00%)	1.00	1.00 (99.97%)	1.00 (100.00%)	1
s491	1.00 (67.69%)	1.39 (94.16%)	1.47	0.96 (96.69%)	0.99 (99.82%)	1
va	0.61 (61.13%)	0.91 (91.63%)	1.00	0.73 (73.78%)	0.89 (89.64%)	1.00
vag	0.96 (84.49%)	0.99 (87.82%)	1.13	0.97 (69.92%)	0.99 (71.08%)	1.39
vas	1.00 (92.74%)	1.06 (98.38%)	1.07	0.98 (98.40%)	0.99 (99.48%)	1
vbor	0.51 (49.52%)	1.00 (96.85%)	1.03	0.25 (24.69%)	0.96 (94.28%)	1.02
vdotr	0.76 (65.31%)	1.16 (100.00%)	1.16	0.48 (42.82%)	1.09 (96.67%)	1.13
vif	0.75 (44.34%)	0.96 (56.39%)	1.71	0.45 (45.04%)	0.97 (97.16%)	1.00
vpv	0.76 (67.12%)	1.01 (88.76%)	1.14	0.68 (67.06%)	0.97 (95.44%)	1.02
vpvpv	0.82 (70.96%)	1.01 (86.80%)	1.16	0.83 (81.08%)	0.99 (96.64%)	1.02
vpvts	0.77 (75.99%)	0.99 (97.59%)	1.01	0.68 (64.61%)	0.99 (93.98%)	1.05
vpvtv	0.81 (79.85%)	1.00 (98.48%)	1.02	0.82 (76.75%)	0.98 (91.38%)	1.08
vsumr	0.57 (56.55%)	0.82 (81.69%)	1.01	0.33 (32.31%)	0.86 (82.16%)	1.04
vtv	0.76 (66.52%)	1.00 (86.73%)	1.15	0.69 (65.15%)	1.02 (95.98%)	1.06
vtvtv	0.82 (80.82%)	0.99 (97.95%)	1.01	0.81 (79.17%)	0.99 (96.80%)	1.02
A-MEAN	0.70 (54.21%)	0.98 (74.32%)	1.47	0.62 (50.49%)	0.94 (77.25%)	1.36
G-MEAN	0.61 (46.47%)	0.85 (64.94%)	1.32	0.51 (41.82%)	0.80 (66.15%)	1.21

Table 10: Continued Leave One Out Cross Validation Results for TSVC.

loop	Nehalem 1-shot	Nehalem 3-shot	Nehalem OPT	Haswell 1-shot	Haswell 3-shot	Haswell OPT
2mm-1	0.25 (24.92%)	0.25 (25.10%)	1.00	0.67 (67.08%)	0.68 (68.53%)	1
2mm-2	0.19 (19.82%)	0.20 (20.45%)	1	0.65 (65.63%)	0.70 (70.20%)	1
3mm-1	0.24 (24.64%)	0.27 (26.99%)	1	0.29 (29.19%)	0.71 (70.94%)	1.01
3mm-2	0.25 (25.08%)	0.28 (27.97%)	1.00	0.54 (54.78%)	0.68 (68.08%)	1
3mm-3	0.26 (25.65%)	0.26 (25.65%)	1.04	0.19 (19.07%)	0.24 (24.24%)	1
adi-1	1.02 (91.36%)	1.02 (91.36%)	1.11	0.97 (93.34%)	0.97 (93.34%)	1.04
adi-2	0.99 (98.69%)	1.00 (98.74%)	1.01	1.00 (100.00%)	1.00 (100.00%)	1.00
adi-3	1.00 (91.78%)	1.00 (91.85%)	1.08	0.94 (93.92%)	0.97 (96.92%)	1.00
adi-4	0.99 (71.09%)	1.32 (94.33%)	1.4	1.22 (99.95%)	1.22 (99.95%)	1.22
adi-5	1.00 (89.71%)	1.11 (99.92%)	1.11	1.00 (100.00%)	1.00 (100.00%)	1.00
adi-6	0.95 (85.81%)	1.00 (89.79%)	1.11	0.94 (93.66%)	1.00 (99.75%)	1.01
atax-1	0.99 (99.72%)	1.00 (100.00%)	1	1.00 (100.00%)	1.00 (100.00%)	1.00
atax-2	1.00 (90.85%)	1.06 (96.57%)	1.10	1.00 (99.44%)	1.00 (99.44%)	1.00
bicg-1	0.99 (99.80%)	1.00 (99.95%)	1	1.00 (98.65%)	1.01 (100.00%)	1.01
bicg-2	0.90 (61.45%)	0.90 (61.52%)	1.47	1.01 (51.08%)	1.98 (100.00%)	1.98
cholesky	1.08 (84.37%)	1.09 (84.60%)	1.29	0.66 (64.57%)	1.00 (97.68%)	1.02
correlation-1	1.00 (82.00%)	1.22 (99.85%)	1.22	0.98 (96.68%)	1.00 (99.07%)	1.01
correlation-2	0.73 (73.30%)	1.00 (99.56%)	1.00	0.95 (92.88%)	0.97 (94.97%)	1.02
correlation-3	0.99 (81.28%)	1.00 (81.41%)	1.22	1.01 (96.26%)	1.04 (98.61%)	1.05
correlation-4	0.82 (24.53%)	1.18 (35.30%)	3.34	1.71 (50.78%)	3.35 (99.24%)	3.38
covariance-1	1.37 (81.42%)	1.68 (100.00%)	1.68	1.02 (100.00%)	1.02 (100.00%)	1.02
covariance-2	1.00 (61.82%)	1.45 (90.01%)	1.61	0.99 (97.68%)	0.99 (97.68%)	1.01
covariance-3	1.00 (81.40%)	1.22 (99.72%)	1.23	0.96 (94.68%)	0.99 (97.95%)	1.01
doitgen-1	1.02 (78.44%)	1.28 (98.97%)	1.30	0.29 (29.16%)	0.67 (67.41%)	1
doitgen-2	1.00 (97.20%)	1.00 (97.61%)	1.03	0.99 (98.28%)	1.00 (99.83%)	1.00
durbin-1	0.99 (92.33%)	1.08 (100.00%)	1.08	0.99 (99.05%)	1.00 (99.58%)	1.00
durbin-2	1.00 (99.85%)	1.00 (99.85%)	1.00	1.07 (96.89%)	1.11 (100.00%)	1.11
dynprog-1	0.98 (89.42%)	1.00 (90.48%)	1.10	0.99 (98.60%)	0.99 (98.78%)	1.01
dynprog-2	0.29 (27.16%)	0.99 (91.88%)	1.08	0.38 (37.90%)	0.59 (58.84%)	1.00
fdtd-2d-1	1.00 (99.64%)	1.00 (99.93%)	1.00	1.05 (99.99%)	1.05 (99.99%)	1.05
fdtd-2d-2	0.99 (99.93%)	0.99 (99.93%)	1	1.06 (99.01%)	1.06 (99.01%)	1.07
fdtd-2d-3	0.91 (90.25%)	0.92 (91.21%)	1.01	0.87 (87.14%)	0.99 (99.20%)	1.00
fdtd-2d-4	1.00 (99.99%)	1.00 (99.99%)	1.00	0.86 (85.47%)	1.00 (99.76%)	1.00
fdtd-apml	0.66 (66.24%)	0.66 (66.24%)	1.00	0.67 (67.44%)	0.68 (67.81%)	1.00
floyd-warshall	5.22 (46.20%)	8.52 (75.43%)	11.30	25.88 (89.68%)	25.88 (89.68%)	28.86
gemm	0.14 (14.04%)	0.32 (32.39%)	1	0.12 (12.01%)	0.38 (38.18%)	1
gemver-1	1.03 (98.39%)	1.03 (98.41%)	1.05	1.06 (98.25%)	1.06 (98.25%)	1.08
gemver-2	1.13 (63.80%)	1.13 (63.80%)	1.78	2.12 (100.00%)	2.12 (100.00%)	2.12
gemver-3	0.99 (99.73%)	1.00 (100.00%)	1.00	0.79 (78.79%)	1.00 (100.00%)	1.00
gemver-4	1.00 (99.94%)	1.00 (99.94%)	1.00	0.99 (99.91%)	0.99 (99.91%)	1
gesummv	1.84 (75.87%)	2.37 (97.65%)	2.43	0.98 (30.25%)	3.24 (100.00%)	3.24
gramschmidt-1	0.86 (86.03%)	0.99 (99.63%)	1	1.02 (95.70%)	1.02 (95.70%)	1.07
gramschmidt-2	1.00 (100.00%)	1.00 (100.00%)	1.00	1.03 (99.97%)	1.03 (99.97%)	1.03
gramschmidt-3	1.00 (12.28%)	7.38 (90.31%)	8.17	3.37 (64.58%)	3.40 (65.11%)	5.22
jacobi-1d-imper	0.99 (93.23%)	1.01 (94.61%)	1.06	1.03 (99.21%)	1.03 (99.21%)	1.03
jacobi-2d-imper	0.96 (89.02%)	1.00 (92.66%)	1.08	0.88 (86.48%)	0.88 (86.48%)	1.02
lu-bench	0.47 (38.01%)	0.49 (39.69%)	1.23	1.25 (97.06%)	1.25 (97.06%)	1.29
ludcmp-1	1.08 (79.82%)	1.12 (83.32%)	1.35	0.88 (72.13%)	1.01 (82.25%)	1.23
ludcmp-2	1.02 (77.96%)	1.02 (77.96%)	1.31	1.11 (89.50%)	1.11 (89.50%)	1.24
ludcmp-3	0.96 (83.89%)	0.98 (85.43%)	1.15	0.93 (91.14%)	0.96 (94.57%)	1.02
ludcmp-4	0.92 (88.77%)	0.95 (91.00%)	1.04	0.88 (87.95%)	0.98 (97.97%)	1
mvt-1	0.99 (98.48%)	1.01 (100.00%)	1.01	1.05 (99.96%)	1.05 (100.00%)	1.05
mvt-2	1.17 (59.23%)	1.37 (69.25%)	1.99	1.65 (55.85%)	1.65 (55.85%)	2.96
reg-detect-1	0.40 (29.36%)	0.98 (70.69%)	1.38	0.37 (37.40%)	0.57 (57.71%)	1.00
reg-detect-2	0.88 (88.63%)	1.00 (99.74%)	1.00	0.20 (20.39%)	0.22 (22.21%)	1.00
reg-detect-3	0.99 (93.79%)	1.00 (94.51%)	1.06	1.00 (98.59%)	1.00 (98.59%)	1.02
reg-detect-4	0.99 (98.06%)	0.99 (98.50%)	1.01	1.00 (99.49%)	1.00 (99.75%)	1.00
seidel-2d	1.00 (88.68%)	1.00 (88.68%)	1.12	0.95 (93.96%)	1.01 (100.00%)	1.01
symm	1.14 (93.38%)	1.14 (93.38%)	1.22	1.09 (95.95%)	1.09 (95.95%)	1.13
syr2k-1	1.00 (92.60%)	1.02 (93.86%)	1.08	1.03 (97.18%)	1.03 (97.18%)	1.06
syr2k-2	0.99 (98.18%)	1.00 (99.05%)	1.01	0.95 (95.56%)	0.97 (97.30%)	1
syrk-1	1.00 (99.92%)	1.00 (99.93%)	1	1.02 (97.52%)	1.02 (97.52%)	1.05
syrk-2	1.00 (90.14%)	1.00 (90.14%)	1.11	0.44 (44.79%)	0.45 (45.27%)	1.00
trisolv	1.00 (72.09%)	1.27 (91.90%)	1.39	1.00 (84.69%)	1.18 (100.00%)	1.18
trmm	1.00 (99.41%)	1.00 (99.80%)	1.00	1.00 (94.44%)	1.05 (100.00%)	1.05
A-MEAN	0.97 (76.31%)	1.21 (84.44%)	1.46	1.34 (81.02%)	1.47 (88.74%)	1.66
G-MEAN	0.85 (68.76%)	0.98 (79.36%)	1.24	0.90 (74.74%)	1.03 (85.50%)	1.20

Table 11: Leave One Out Cross Validation Results for PolyBench

- `ivdep` - ignores unknown vector dependences. It doesn't affect known vector dependences
- `simd vectorlength(N)` - vectorizes the entire loop nest, even those with nested loops. Useful for multi-loop vectorization and overriding vector dependences. We vary the safe vector length size to encourage different vectorization size generation ($n = 2, 4, 8$).

Improper usage of these directives can produce invalid code. That's why we analyzed the output of the execution to verify its correctness. For PolyBench we looked at the live-out data to verify correctness. For TSVC loop nests, we looked at a computed checksum which was previously used to verify correctness. All versions of each benchmark were compared to the default version (no optimization) to verify correctness.

7.2 Program Execution

An important aspect of a speedup predictor is ensuring that you have an accurate speedup. Timing was performed at millisecond granularity for TSVC and clock cycle granularity for PolyBench. We also ran each loop nest 10 times and took the average. All execution times were within 0.8% of one another, so the the timing was accurate enough considering the variance in speedup from applying optimizations. To help mitigate any additional runtime constraints, all executions were performed on a system in single-user mode with no network access activated.

The speedup measurements recorded were done at a kernel level. We timed the entire kernel execution although we were only optimizing a single loop nest. For the TSVC benchmark suite, this wasn't a problem as each kernel consisted of only one loop nest. For PolyBench benchmarks the speedup values are more representative of a trend instead of actual speedup. To know the best optimizations to be applied to an entire PolyBench benchmark, results from each loop nest would need aggregated and applied. Once each local optimization is applied to the entire kernel, the kernel could be compiled and executed to obtain an actual speedup. We did not measure the best kernel speedup for PolyBench in this work.

7.3 Machine Learning Model

We used the same machine learning model presented by Park et. al. [24], including the *shortest-path graph kernel*. There are two differenced between the model they used and the model we used – both of which are with the training data. First, the optimization bit vector in our work is only defining levels of vectorization. Second, we use a smaller control flow graph to just focus on the target loop nest. By limiting our optimization search space to a single type of optimization and restricting the control flow graph for classification to only focus on the loop nest being optimized, the generated model is more targeted toward finding small variations between different data points. with similar kernel matrices being generated.

8 Related Work

The closest work related to this research is the work by Stock et al. [28]. The major limiting contribution of this work is the scope of the class of loops which the model is able to optimize (tensor and stencils). Their training model is derived from information within the inner-most loop and vectorization information. A takeaway from their work was the importance of (a) the fact that a single feature alone does not correlate to good performance and (b) that a weighted rank model always outperformed support vector machines. Additional related work has been divided into the following three sections: Section 8.1 which discusses Vectorization, Section 8.2 discusses Compiler Optimization, and Section 8.3 discusses Machine Learning.

8.1 Vectorization

Callahan et. al [5] created an initial set of micro-benchmarks to evaluate the vectorization capability of compilers. The benchmarks were written in Fortran and the case study was performed when SIMD was first being introduced in consumer, general purpose hardware. Maleki et al. [20] extended this to support C as well as the addition of several more micro-benchmarks. They then manually modified some codes that did not vectorize which could ultimately be vectorized. Our work builds on Maleki's contribution by automatically relaxing vectorization heuristics to properly vectorize certain benchmarks.

Henretty et. al [13] was able to improve vectorization of stencil applications with a compiler focused on stencil codes, but does not extend to other types of kernels. Nuzman et. al [23] improved performance of kernels with outer-loop vectorization on CBE and PowerPC architectures. Our work extends this by targeting Intel microarchitectures automatically with the use of the `#pragma simd` directive.

Holewinski et. al, Evans et. al and Barik et. al [15, 11, 4] used trace information to determine vectorization potential and to automate the selection of vector instructions.

Hohenauer et. al proposed a framework to enable retargetable compilers to emit more appropriate SIMD instructions [14]. McFarlin et al. automatically vectorized FFT kernels [21]. Both of these works focused on modifying or constructing a compiler to perform the optimizations. McFarlin's work is limited in scope to FFT kernels while Hohenauer's work cannot be easily extended to new architectures or compilers.

8.2 Compiler Optimization

Kong et al. used polyhedral transformations to help drive improved vectorization [18]. This method can help improve performance, but it does not help determine where or why existing compilers cannot better optimize certain code. Pouchet et al. [26] used iterative and model-driven optimizations to drive auto-parallelization. The same can be done for autovectorization. Eagan et al. used directives to drive compiler optimization selection [10].

8.3 Machine Learning

Park et al., Dubach et al., and Cavazos et al. [25, 24, 9, 8, 7] focused on using machine learning methods to construct prediction models based on performance counters. Park extended the existing work by using graph-based program characterization [24]. Agakov [3] used machine learning to reduce and eliminate branches of optimizations being applied. Work has also been done in the area of finding which features to use for machine learning models for optimized compilation [19].

9 Future Work

We plan to continue development of `VALT` to support multiple compiler backends and extend `autovec` to other types of directive-based optimizations. PGI Compilers have their own directive language, and we could construct a backend for `VALT` to emit the PGI directives to evaluate another compiler on the same architecture. `autovec` could also switch from an iterative code generator to an auto-tuner capable of some intelligent selection of optimizations to explore, thus we could consider its inclusion in general-purpose auto-tuning frameworks [22, 12].

We could also modify how we represent optimizations with our speedup predictor. We could annotate our graph-based representation of our programs with optimization information. This would reduce some of the encoding options that we currently have and would remove any encoding restrictions.

Due to increased support of vectorization directives such as `#pragma simd` and `#pragma ivdep` we will be able to extend this existing work directly to compilers such as future versions of GCC that will supporting these pragmas. In addition to considering other compilers, we could explore architectures with wider vectorization sizes such as the Intel Xeon Phi (Knight's Corner) and the upcoming Knight's Landing and Skylake microarchitectures where AVX-512 is supported in GCC and Intel Compiler.

10 Conclusion

In this paper we provided techniques, both manual and automatic, for determining the best way to optimize programs using pragmas that control vectorization performed by the Intel Compiler. The utilities we developed for iterative compilation and code generation can be further used by non-experts in the generation and analysis of programs. Finally, we leveraged the obtained knowledge to design a graph-based speedup predictor to predict the speedup of a program given a sequence of optimizations.

The results of this work can help programmers tailor their applications to make the most out of their vectorizable codes. The iterative compilation utilities can be especially useful when exploring tuning options during the code optimization phase. The speedup predictor can be used as part of a general-purpose auto-tuning strategy that does not require human interaction. Overall, the contributions introduced in this paper can help programmers guide the compiler into generating

optimized code without requiring expert knowledge on the compiler inner workings or the underlying architecture.

References

[1] Gcc 4.9 release series changes, new features, and fixes. `https://gcc.gnu.org/gcc-4.9/changes.html`, 2014.

[2] Minimal intermediate representation (minir). `http://www.assembla.com/wiki/show/minir-dev`, 2014.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 11 pp.–, March 2006.

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

[5] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 98–105, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[6] CAPS Entreprise. Caps compilers. `http://www.caps-entreprise.com/`, May 2014.

[7] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. P. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 24–34, New York, NY, USA, 2006. ACM.

[8] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, pages 185–197, March 2007.

[9] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, pages 131–142, New York, NY, USA, 2007. ACM.

[10] B. Eagan, G. Civario, and R. Miceli. Investigating performance benefits from openacc kernel directives. In M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 616–625. IOS Press, 2014.

[11] G. C. Evans, S. Abraham, B. Kuhn, and D. A. Padua. Vector seeker: A tool for finding vector potential. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 41–48, New York, NY, USA, 2014. ACM.

[12] G. Fursin, R. Miceli, A. Lokhmotov, M. Gerndt, M. Baboulin, A. D. Malony, Z. Chamski, D. Novillo, and D. Del Vento. Collective mind: Towards practical and collaborative auto-tuning. *Scientific Programming*, 22(3), Sept. 2014.

[13] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[14] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr. A simd optimization framework for retargetable compilers. *ACM Trans. Archit. Code Optim.*, 6(1):2:1–2:27, Apr. 2009.

[15] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 371–382, New York, NY, USA, 2012. ACM.

[16] Intel Corporation. A guide to auto-vectorization with intel c++ compilers. `https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf`, April 2012.

[17] Intel Corporation. User and reference guide for the intel c++ compiler 14.0. `https://software.intel.com/en-us/compiler_14.0_ug_c`, September 2013.

[18] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, New York, NY, USA, 2013. ACM.

[19] H. Leather, E. Bonilla, and M. O'boyle. Automatic feature generation for machine learning–based optimising compilation. *ACM Trans. Archit. Code Optim.*, 11(1):14:1–14:32, Feb. 2014.

[20] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, Oct 2011.

[21] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel. Automatic simd vectorization of fast fourier transforms for the larrabee and avx instruction sets. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 265–274, New York, NY, USA, 2011. ACM.

[22] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In P. Manninen and P. Öster, editors, *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science*, pages 328–342. Springer Berlin Heidelberg, 2013.

[23] D. Nuzman and A. Zaks. Outer-loop vectorization: Revisited for short simd architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[24] E. Park, J. Cavazos, and M. A. Alvarez. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 196–206, New York, NY, USA, 2012. ACM.

[25] E. Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 119–129, April 2011.

[26] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010.

[27] L. N. Pouchet. Polybench/c: the polyhedral benchmark suite. `http://www.cs.ucla.edu/~pouchet/software/polybench/`, March 2012.

[28] K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Trans. Archit. Code Optim.*, 8(4):50:1–50:23, Jan. 2012.