Using Machine Learning to Improve Automatic Vectorization

Kevin Stock, The Ohio State University
Louis-Noël Pouchet, The Ohio State University
P. Sadayappan, The Ohio State University

Automatic vectorization is critical to enhancing performance of compute-intensive programs on modern processors. However, there is much room for improvement over the auto-vectorization capabilities of current production compilers, through careful vector-code synthesis that utilizes a variety of loop transformations (e.g. unroll-and-jam, interchange, etc.).

As the set of transformations considered is increased, the selection of the most effective combination of transformations becomes a significant challenge: currently used cost-models in vectorizing compilers are often unable to identify the best choices. In this paper, we address this problem using machine learning models to predict the performance of SIMD codes. In contrast to existing approaches that have used high-level features of the program, we develop machine learning models based on features extracted from the generated assembly code, The models are trained off-line on a number of benchmarks, and used at compile-time to discriminate between numerous possible vectorized variants generated from the input code.

We demonstrate the effectiveness of the machine learning model by using it to guide automatic vectorization on a variety of tensor contraction kernels, with improvements ranging from 2× to 8× over Intel ICC's auto-vectorized code. We also evaluate the effectiveness of the model on a number of stencil computations and show good improvement over auto-vectorized code.

1. INTRODUCTION

With the increasing degree of SIMD parallelism in processors, the effectiveness of automatic vectorization in compilers is crucial. Most production compilers, such as Intel's ICC, GNU GCC, PGI's pgcc, IBM's XL/C etc., perform automatic vectorization. While the automatic vectorization of these compilers often provides very good performance improvement compared to scalar code, the achieved performance nevertheless is often far below the peak performance of the processor, even when the data fits within L1 cache and no cache misses are incurred.

The main reason for the sub-optimal performance of vectorized code from current production compilers is the extreme difficulty of choosing the best among a huge number of possible transformations of high level loop nests into assembly code. The complex execution pipelines with multiple functional units in modern processors makes it extremely challenging to develop analytical performance models to predict the execution time for a sequence of machine instructions. Production compilers therefore typically use simple heuristics to guide loop transformations and back-end code generation.

Machine learning (ML) models have garnered considerable interest in the compiler community. ML approaches have been used in numerous situations, for instance to optimize compiler flag settings [Agakov et al. 2006; Cavazos et al. 2007], choose effective loop unrolling factors [Monsifrot et al. 2002], and optimize tile sizes [Yuki et al. 2010]. However, we are unaware of any prior use of ML models to assist in optimizing vector code generation. In this paper, we develop such a model.

The present work focuses particularly on the vectorization of a class of compute-intensive loops that arise commonly in quantum chemistry codes — tensor contractions. Tensor contractions are essentially generalized higher dimensional matrix-matrix products, where the tensors can have more than two dimensions and the summations can be performed over several dimensions. Various types of tensor contractions are required in the implementation of high accuracy quantum chemistry models such as the coupled cluster method [Crawford and Schaefer III 2000]. These codes are vectorized by current compilers, but the achieved performance is often far below machine peak. We use a code generator that explicitly generates vector intrinsics, after considering various possible loop permutations, unrolling and choices of the loop to be vectorized along. We build an ML model to predict the performance of the generated assembly code for the various possible transformations. After training the ML model using of a number of generated variants from a training set of synthetic nested loops representing tensor contractions, the model is used to optimize vectorized code generation for a number of tensor contractions from the CCSD (Couple Cluster Singles and Doubles) method. We show that the code generated using the ML model is significantly better than that generated through auto-vectorization by GCC and ICC. We complete our experimental study by assessing the effectiveness of the generated ML model to successfully rank-order the performance of a number of vectorized variants for several stencil computations.

The rest of the paper is structured as follows. In Section 2 we describe the approach we use for vectorizing tensor contractions. We then characterize the optimization search space in Section 3, showing that the percentage of high-performance points in this space is small. In Section 4, we discuss the different ML models that we evaluate, along with a description of the various feature sets that we use with the ML models. Section 5 presents experimental results and an analysis of the relative effectiveness of the evaluated models. We then develop a composite ML model in Section 6 from the individual models discussed in Section 4 and show its improvement upon the individual ML models. Section 7 discusses the application of our techniques to stencil computations and analyzes experimental results. Section 8 provides a summary of the state of the art and related work.

2. SYNTHESIS OF VECTORIZED CODE

2.1. Automatic Vectorization of Tensor Contractions

In this work, we focus on the class of loops that arise with tensor contractions (TC). Tensor contractions represent the core computational kernels in accurate *ab initio* methods for electronic structure calculations such as the coupled cluster (CC) and configuration interaction (CI) models in quantum chemistry suites [Harrison et al. 2005; Crawford and Schaefer III 2000; Baumgartner et al. 2002]. A TC involves d-nested loops that update a k-dimensional output tensor ($k < d$). As an illustration, we show below an example of a TC, where N, M, O, and P represent the size of each dimension. A and B represent the input tensors, and C is the output tensor.

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
  for (k=0; k<O; k++)
   for (l=0; l<P; l++)
     C[l][j][k] += A[i][j][k]*B[i][l];
```

In a TC, each index of the nested loop appears exactly twice as an array index, each appearance being in a different tensor. In the example, i appears as an index for arrays A and B (which makes it a 'contracted' index), j and k index into arrays A and C, while l indexes B and C. The loop nest is fully permutable, i.e., all 24 possible loop orderings for this example are valid. We label a TC according to its array indices as represented in C, A, and B. The example TC would be represented as `ljk-ijk-il`.

Stock et al. developed a customized vectorization algorithm that generates SIMD intrinsics-based code [Stock et al. 2011] for six specific TC kernels from MADNESS [Harrison et al. 2005]. That code generation algorithm is also suitable for generating vectorized variants of arbitrary TCs. We first provide a brief description of the algorithm, before discussing the set of optimizations that are applied to generate the vectorized variants.

Vectorization may be achieved along any of the nested loop dimensions in the iteration space for a TC. For those arrays in the statement for which the fastest varying dimension (referred to as the unit-stride dimension, e.g., the rightmost index in C/C++ and the leftmost index in Fortran/MATLAB) matches the loop dimension chosen for vectorization, groups of adjacent elements in memory can be directly loaded into vector registers after an unroll-and-jam transformation of the vectorized loop with the unroll factor being the vector length. If the loop dimension chosen for vectorization does not appear at all among an array's indices, then a replication of that element on all components of a vector register will be required, i.e., a *splat* operation. Finally, if the vectorized loop dimension appears in some other position than the fastest varying dimension, a transposition is used after unroll-and-jam of the loop corresponding to the index of the fastest varying dimension of that array. This enables use of vector loads followed by an inter-register transpose to gather non-contiguous data into vector registers. We note that the needed data movement is not dependent on the loop permutation used, although the total data movement cost is a function of both the loop which is vectorized as well as the loop permutation and unroll factors.

2.2. Tensor Contractions Considered

We consider a set of TCs from CCSD (Coupled Cluster Single Double), a computational method for *ab initio* electronic structure calculations [Crawford and Schaefer III 2000; Hirata 2003]. CCSD requires the execution of many TCs, from which we extracted 19 unique kernels. These are representative of the kinds of TCs that occur in computational chemistry and were used to evaluate the effectiveness of an ML model in identifying effective vectorized code variants. In order to train an ML model, a different set of 30 randomly generated TCs was generated, as described in Section 3.2.

2.3. Optimization Space

For any TC, the set of possible vectorized code variants is determined by the three degrees of freedom of the vector synthesis algorithm [Stock et al. 2011]: i) the loop order of the TC (that is, the result of a sequence of loop interchanges), ii) the specific loop from the loop nest along which vectorization is performed, and iii) the loops among the loop nest on which unroll-and-jam is applied, along with the associated unroll factors. The total number of variants for a TC can be very large due to these three degrees of freedom: for the 30 TCs used to train the ML model, the number of code variants ranged between 42 and 2497, due to differences in the number of loops and array indices in the TCs.

Loop permutation: The first of the transformations we discuss is loop permutation. We focus on full tiles which fit into the first level of cache (TCs are fully tileable computations [Wolfe 1989]), so the effect of loop permutation with regards to improving spatial locality of memory accesses is minimal. However, loop permutation can enable some load and store operations to be hoisted out of the innermost loop. This can create a significant improvement between variants with different loop orders, when the hoisted instructions are relatively expensive (e.g., splats and register transpose operations). For a d-dimensional loop nest, $d!$ distinct loop orders exist.

Vectorized loop: The choice of the loop to be vectorized affects how memory is loaded to and stored from the SIMD registers. All loops that are either parallel or represent reductions are considered for vectorization. Thus, for TCs all the loops are considered, because each loop is either fully parallel or a reduction. If an array is accessed in unit-stride along the vectorized dimension, a standard vector load instruction can be used, which is efficient on modern architectures. If an array is not accessed by the vectorized dimension, a vector splat is required to load the data, which involves a scalar load followed by replication of the value to all elements of a SIMD register. If an accumulation is performed on an array not accessed by the vectorized dimension, a reduction is involved and the SIMD register must be reduced before a scalar store is executed. Finally, an array can be accessed in non-unit stride by the vectorized loop. In general, in order to vectorize the statement, consecutive elements along the vectorized dimension must be gathered into a SIMD register by using multiple scalar loads. However, with TCs, data can be loaded with vector loads along the array's unit-stride dimension, followed by a register level transpose to place consecutive elements along the vectorized dimension into vector registers. This transformation contributes a multiplicative factor equal to the loop nest depth to the size of the space of vectorized code variants considered.

Unroll-and-Jam: The final transformation considered is unroll-and-jam, which can increase register reuse and therefore increase the arithmetic intensity. All loops in TCs are eligible for unroll-and-jam, so that the potential search space for this transformation alone can be very large. We restrict the possible unroll factors of each loop to the set of divisors of the loop size to eliminate the need to generate inefficient edge-case handling code within the optimized loop nest, with the result that loops in our benchmarks are unrolled $1\times$, $2\times$, $4\times$, or $8\times$. In the context of vectorization, unroll-and-jam is also used to enable the register transpose operation described above.

2.4. Detailed Example

To show how these optimizations interact, we present in Figure 1 an optimized version of the `ijkl-imkn-jnlm` TC from the CCSD test set, for double precision floating point numbers, and the AVX vector instruction set. The choices with respect to the above three optimizations for this example are as follows:

```
for (m=0; m<8; m+=8) {
  for (j=0; j<4; j+=1) {
    for (l=0; l<8; l+=8) {
      for (k=0; k<4; k+=4) {
        for (n=0; n<8; n+=1) {
          B_temp0=_mm256_loadu_pd(&B[(j)*512+(n)*64+(l)*8+(m)]);
          /* Omitting 14 similar loads into B_temp1 ... B_temp14 */
          B_temp15=_mm256_loadu_pd(&B[(j)*512+(n)*64+(l+7)*8+(m+4)]);

          __t0 = _mm256_unpacklo_pd(B_temp0,B_temp1);
          __t1 = _mm256_unpackhi_pd(B_temp0,B_temp1);
          __t2 = _mm256_unpacklo_pd(B_temp2,B_temp3);
          __t3 = _mm256_unpackhi_pd(B_temp2,B_temp3);
          B_temp0 = _mm256_permute2f128_pd(__t0, __t2, 0x20);
          B_temp2 = _mm256_permute2f128_pd(__t1, __t3, 0x20);
          B_temp1 = _mm256_permute2f128_pd(__t0, __t2, 0x31);
          B_temp3 = _mm256_permute2f128_pd(__t1, __t3, 0x31);
          /* Omitting 3 similar 4-by-4 transposes, for B_temp4 ... B_temp15 */

          for (i=0; i<2; i+=1) {
            A_temp0=_mm256_broadcast_sd(&A[(i)*256+(m)*32+(k)*8+(n)]);
            /* Omitting 30 similar splats into A_temp1 ... A_temp30 */
            A_temp31=_mm256_broadcast_sd(&A[(i)*256+(m+7)*32+(k+3)*8+(n)]);

            C_temp0=_mm256_loadu_pd(&C[(i)*128+(j)*32+(k)*8+(l)]);
            /* Omitting 6 similar loads into C_temp1 ... C_temp6 */
            C_temp7=_mm256_loadu_pd(&C[(i)*128+(j)*32+(k+3)*8+(l+4)]);

            C_temp0=_mm256_add_pd(_mm256_mul_pd(A_temp0,B_temp0),C_temp0);
            /* Omitting 62 similar muliply adds updating C_temp1 ... C_temp6 */
            C_temp7=_mm256_add_pd(_mm256_mul_pd(A_temp31,B_temp15),C_temp7);

            _mm256_storeu_pd(&C[(i)*128+(j)*32+(k)*8+(l)],C_temp0);
            /* Omitting 6 similar stores from C_temp1 ... C_temp6 to C */
            _mm256_storeu_pd(&C[(i)*128+(j)*32+(k+3)*8+(l+4)],C_temp7);
          }
} } } } }
```

Fig. 1. Example of generated code for the `ijkl-imkn-jnlm` Tensor Contraction

Loop permutation: The loop order in this example is `mjlkni`, which allows access to `B` to be hoisted from the innermost loop. Doing so is important because the loads of `B` require an expensive transpose.

Vectorized loop: For this code variant, the tensor contraction is vectorized along the `l` dimension. Therefore the loads and stores to the result array `C` can be done with standard vector operations. However, loads from `A` require a splat operation and loads from `B` involve register-level transposition with respect to the `m` and `l` dimensions.

Unroll-and-jam: The amount each loop is unrolled is indicated by how much its iterator is incremented. The loops that are unrolled promote reuse of certain registers. In this example unroll-and-jam of the `m` loop allows for more reuse of the registers with values from `C`. Because of the unroll-and-jam, the resulting code is almost 300 lines long. This is shown in Fig. 1, where repetitive sections of similar code have been replaced by comments describing them.

3. CHARACTERIZATION OF THE OPTIMIZATION SPACE

In order to characterize the complexity of the optimization problem we address, we evaluate numerous possible variants (that is, each one corresponds to a specific set of parameters given to the vectorization algorithm) for a variety of TCs.

3.1. Tested Configurations

We performed our evaluation across several compilers, processors and instruction sets. Specifically, we considered two production compilers: GNU GCC 4.6 and Intel ICC 12.0; two processors: Intel Core i7 with the Nehalem micro-architecture and Intel Core i7 with the Sandy Bridge micro-architecture; and two data types for the tensor contractions: `float` and `double`. For the Nehalem, we evaluated using the SSE4.2 SIMD instruction set, while for the Sandy Bridge we evaluated using SSE and AVX instruction sets. The cartesian product of these possibilities results in a total of twelve different configurations.

The Nehalem processor runs at 2.66 GHz, with a theoretical peak per single-threaded application of 10.64 GFlop/s (`double`) and 21.28 GFlop/s (`float`). The Sandy Bridge processor runs at 3.4 GHz, with theoretical single-threaded peak SSE performance of 13.6 GFlop/s (`double`) and 27.2 GFlop/s (`float`) and peak AVX performance of 27.2 GFlop/s (`double`) and 54.4 GFlop/s (`float`).

3.2. Generation of Tensor Contractions for Training Set

In order to train the ML models, a set of 30 TCs were randomly synthesized. These were generated using 3 to 6 indices and between 1 and 4 contracted indices. The array indices, corresponding to the contraction being performed, were chosen as to span a variety of contractions and transposition of matrices which can occur in quantum chemistry codes. The TCs were named in a canonical fashion to eliminate duplicates, and any that occur in the 19 CCSD TCs were removed from the set of synthetic TCs for ML training. The process of generating random TCs for each number of indices was repeated until 30 acceptable TCs were generated: there are 4 with 3 indices, 10 with 4 indices, 5 with 5 indices, and 11 with 6 indices. Of these, only 11 have an obvious dimension for vectorization (where two of the arrays have the same unit stride loop index).

Since TCs are fully tileable, contractions on large tensors can always be tiled such that all data accessed by a tile fits entirely in L1 cache. We therefore focus on L1-resident datasets for construction of the ML model and its evaluation. For each TC, the sizes of the tensors were chosen to ensure that all tensors could together fit in L1 cache, with sizes along the fastest varying dimension of all tensors being a perfect multiple of the vector length.

For our experiments, we used in-program timing code to monitor the execution time. Each TC was run approximately 10^5 times, although the actual number of repetitions was set so that a total of roughly 50 million floating point operations were executed. The execution time was computed as an average over all the repetitions. The variance among independent runs for any TC was less than 4%.

3.3. Variability Analysis

Figure 2 shows the sorted performance distribution, of all the considered variants for four representative TCs from the CCSD application, for one of the twelve configurations of processor, instruction set, data type, and compiler.

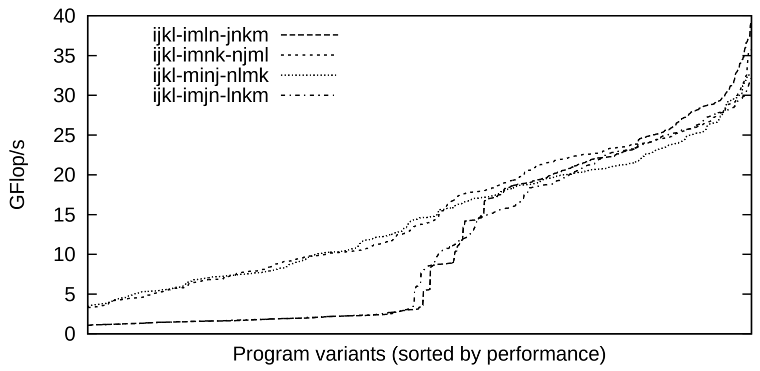

Fig. 2. Performance distribution of Tensor Contractions using the Sandy Bridge/AVX/float/ICC configuration.

We observe that only a very small fraction, fewer than 4% of the space of vectorized code variants, attains 80% or more of the search-space optimal performance. The code generated by ICC's auto-vectorization on the input code (that is, without using the code synthesis algorithm to generate the vector intrinsics-based code) only achieves 1.90 GFlop/s for the `ijkl-imln-jnkm` TC, which is worse than 97% of the points in this search space, where the best variant performs at 39.08 GFlop/s. In this paper, our objective is to develop an effective model that can assist in automatically finding the best variant in the search space.

For a single TC from the CCSD code, Figure 3 shows the sorted performance distribution for three configurations (Nehalem with SSE, Sandy Bridge with SSE and Sandy Bridge with AVX) All three configurations use floats and ICC. We observe that the distribution of performance for different configurations can be quite different. For the considered TC, ICC auto-vectorization of the input code on Sandy Bridge using AVX results in performance of 2.96 GFlop/s. Here 74% of the search space is faster, with the space optimal variant achieving 38.57 GFlop/s.

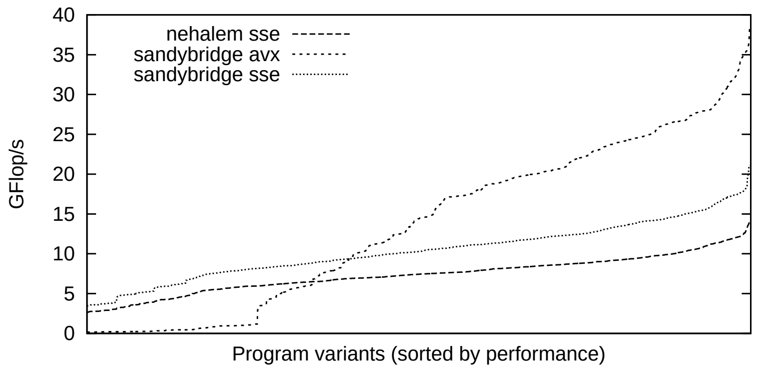

Fig. 3. Performance distribution of a Tensor Contraction across different configurations.

We also observe that the compiler has a critical impact on the relative performance of different vectorized variants. Figure 4 plots, for the same representative TC, the performance of all its vectorized variants when using ICC and GCC. The variants are sorted according to their performance when compiled with ICC. We observe wild performance variations between ICC and GCC for a given variant, as illustrated by the numerous spikes. The best performing vectorized variant with GCC (28.1 GFlop/s) performs poorly when compiled with ICC, achieving only 11.25 GFlop/s. The converse is also true, as shown by the relatively low performance with GCC in the far right of the Figure.

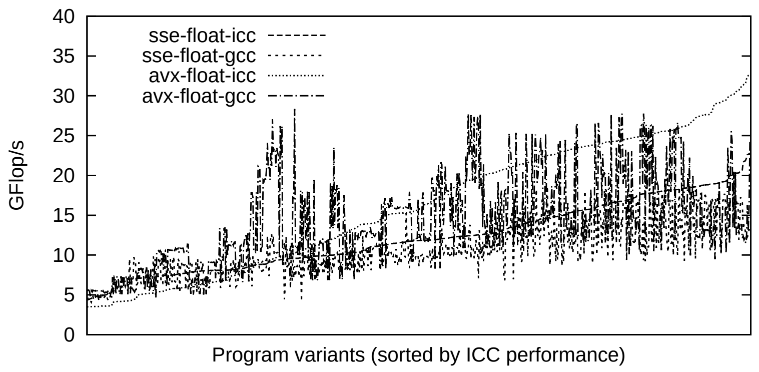

Fig. 4. GCC results sorted in ascending order of ICC performance.

The choice of compiler also affects the maximal performance achievable for a particular TC. For example, for the `ijkl-imjn-nlmk` TC on Sandy Bridge using AVX and floats, ICC attains 33.16 GFlops/s, while GCC is 13% slower at 28.94 GFlop/s. However, ICC is not always more effective.

There are instances where the best among the vectorized variants compiled with GCC is significantly faster than the best of all variants compiled with ICC.

4. MACHINE LEARNING MODELS

We have shown in Section 3 that the best vectorized variant of a TC is application-dependent, machine-dependent and even compiler-dependent. We seek to build a decision model to determine which variant is expected to have the best performance.

4.1. Problem Formulation

Traditional machine learning approaches to automatically select a loop transformation sequence require explicit modeling of the sequence in the learning problem [Agakov et al. 2006; Dubach et al. 2009; Cooper et al. 2002]. We depart from this view and instead build *performance predictors* that are independent of the algorithms used to generate the vectorized variants. We solve this problem by building models that can predict the performance of vectorized programs *without running them*, so that we can rank programs (i.e., all the vectorized variants of the benchmark) according to their predicted performance. The program that ranks first is selected as the output of our model. In our approach, there is no need for the model to correctly predict the numerical value associated with the performance of a program variant: only the relative order between predicted variants is relevant.

Our approach is motivated by two important factors. First, by developing a technique that is not tied to the specifics of the optimization algorithm, we significantly increase the applicability of the model. The second motivation comes from the optimization space to consider. Building a model that operates directly on a high-level transformation sequence (e.g., the parameters to be given to a vectorization algorithm) would have to take into account that the parameter space varies across benchmarks. Input programs to be optimized may have very different optimization spaces: for example, the decision of which loop to vectorize is made from the set of parallel and reduction loops of the program, and this set is not constant across programs. The fact that the decision space is not constant across programs makes it poorly suited to machine learning models, and severely challenges the capability of any model to be generic across arbitrary benchmarks. In contrast, our approach is robust to arbitrary search spaces and algorithms to construct them.

Compiler optimization heuristics play a significant role in the final performance of a program. These optimizations are usually organized in passes that are successively applied. The optimization heuristics implemented in compilers are often fragile and sensitive to the order in which those passes are performed, and of course to the precise structure of the input program. The main reason for the limited success of analytical models for performance prediction is that they attempt to predict performance before the compiler optimizations are applied. Analytically modeling each and every optimization of a compiler is just not feasible, and even if feasible would require a redesign of the model for each compiler revision.

To overcome this problem we propose to build models that *operate on the assembly code* that is produced by the compiler. By working on the end result of the entire compiler optimization process, we avoid the need to model the impact of those optimizations. In particular, after instruction scheduling and register allocation are performed, we can analyze important performance factors such as the arithmetic intensity of the vectorized loops (that is, the ratio of arithmetic vs. memory movement operations) and the distance between producer and consumer operations.

4.2. Assembly Features

We focus our feature extraction on the inner-most loop of the kernels as it is the dominant contributor to execution time. Within the inner-most loop we consider the number of occurrences of each type of vector instruction and the distance (in number of instructions) to the first consumer of the value produced by each vector instruction, if any.

Vector Operation Counts: Five parameters counting each of the following types of vector operations: addition, multiplication, load, store, and miscellaneous (e.g. shuffles); and additionally the total number of vector operations, equal to the sum of the five counts. Other useful metrics can be derived from these values as described below. As an example, Figure 5 shows a piece of assembly code with

3 loads, 0 miscellaneous, 1 multiply, 1 add, and 1 store. Scalar operations in the assembly code are largely ignored in our model, since separate hardware outside of the vector units is available on CPUs to execute scalar operations and they are not expected to play a significant role in determining the performance of loops dominated by vector instructions. The experimental results shown later in the paper show that vector operation counts form a sound basis for the input set.

```
movups    544(%rsp,%rdi), %xmm1
movups    560(%rsp,%rdi), %xmm2
movups    572(%rsp,%rdi), %xmm3
addq      $32, %rdi
mulps     %xmm1, %xmm2
addps     %xmm2, %xmm3
movups    %xmm3, 572(%rsp,%rdi)
incq      %r9
cmpq      $8, %r9
jb        ..B1.6
```

Fig. 5. Example of x86 assembly code.

```
shufps   $0, %xmm6, %xmm6
movaps   %xmm15, %xmm14
shufps   $0, %xmm4, %xmm4
movaps   %xmm6, %xmm12   (used in 3)
mulps    %xmm5, %xmm14   (used in 3)
mulps    %xmm2, %xmm15   (used in 4)
mulps    %xmm5, %xmm12   (used in 5)
addps    %xmm14, %xmm7   (not used)
mulps    %xmm2, %xmm6    (used in 5)
addps    %xmm15, %xmm10  (not used)
mulps    %xmm4, %xmm5    (used in 4)
addps    %xmm12, %xmm0   (not used)
mulps    %xmm2, %xmm4    (used in 3)
addps    %xmm6, %xmm1    (not used)
addps    %xmm5, %xmm3    (not used)
addps    %xmm4, %xmm11   (not used)
```

Fig. 6. Example of x86 assembly code with many arithmetic instructions with *Sufficient Distance* annotations.

Arithmetic Intensity: The ratio of vector arithmetic operations to vector loads. Although this is a derived metric computed from the vector operation counts, it is a sufficiently significant metric to warrant explicit inclusion as an input feature for the machine learning models — maintaining a high arithmetic intensity is essential to high performance on modern CPUs. In the example of Figure 5, the arithmetic intensity is $\frac{2}{3}$ since there are 2 arithmetic operations (1 add and 1 multiply) and 3 loads.

Sufficient Distance: The number of arithmetic vector operations that produce a result that is not consumed in the next four instructions. The specific distance of four was chosen as it is representative of the latency of vector arithmetic operations used on both tested x86 CPUs, however this value could be easily tuned for other architectures. The rationale for this metric is that operations with a sufficiently large distance between producer and consumer instruction are unlikely to cause pipeline stalls, while operations with limited distance between producer and consumer can be expected to be more performance limiting. Figure 6 shows an example of x86 of arithmetic instructions annotated with the distance until their output is used (the right operand is the output). In this example there are 10 instructions with sufficient distance. When extracting this feature we also consider values produced that are not used until the next iteration of the loop.

Sufficient Distance Ratio: The percentage of arithmetic vector operations which have sufficient distance from their first consumer. Higher values of this ratio suggest that available instruction level parallelism may be better exploited by the multiple vector functional units of a processor, without pipeline stalls due to dependences. In Figure 6 the ratio is $\frac{10}{13}$ since there are 13 arithmetic instructions, of which 10 have sufficient distance.

Total Operations: The count of the total number of instructions in the innermost loop, including non-vector operations. In Figure 5 this value is 10, and in Figure 5 this value is 16. This is important because modern processors are more effective in processing loops with a limited body size which they can cache and reuse the decoded micro operations between iterations of the loop.

Critical Path: An approximation of the minimum number of instructions that must be executed in serial order in the inner most loop. This metric is based on the vector operation counts and number of ports available to process each type of instruction. In example Figure 5 the critical path is computed as 5 instructions. This is because only vector operations are considered, of which there are 6, but the add and multiply are considered only one cycle for this metric as the CPU can issue the two of them in parallel.

These features are combined to form feature sets which are used as inputs to each of the machine learning models. Our strategy for building an effective feature set is to start with the simplest possible set which should be able to predict performance reasonably well, and incrementally add features in order of importance based on the additional information conveyed by the features. However, there may come a point where additional features fail to provide benefit and may even reduce the effectiveness of the models. We provide a detailed analysis in Section 5.

4.3. Overview of the Process

Our overall approach is to train ML models to predict the performance P of a vector of input features $ASM_{features}$ that characterizes a program. This training is done off-line, typically during the installation of the compiler. When a new program is to be optimized, the model is used to predict the performance of a number of transformed variants of the input program. This evaluation *never requires actual execution of the program* nor any of the transformed variants. The model determines which of the transformed variants is predicted to perform best, and this variant is the output of the optimization process.

4.4. Training and Evaluation

We train a specific model for each of the configurations detailed in Section 3.1. That is, for each processor type, compiler, data type (float or double) and SIMD instruction set (AVX or SSE), a dedicated model is trained. This is relevant as we have shown the sensitivity of the best variant to each of these factors. For the training of the models, we use *exclusively the 30 synthetic TC kernels* as described in Section 3.2. Thus, none of the CCSD TC kernels used to evaluate the models in Sec. 6 are included among the set of kernels used for training.

A model is trained as follows. A collection of vectorized variants is generated for each of the 30 kernels, and for each of them their feature vector $ASM_{feature}$ is computed. Each assembly code variant is run on the target machine and its performance P_{actual} is recorded. We used GFlop/s as the metric for P_{actual}. The model is trained with the tuple $(ASM_{feature}, P_{actual})$. In the experiments of Sec. 4, we use the standard *Leave One Benchmark Out Cross-Validation* procedure for evaluating our models on the 30 TC set. That is, the models are trained on $N-1$ benchmarks and all their associated variants, that is approximately 20000 programs. Models are then *evaluated on all the variants of the benchmark that has been left out*. For each variant, the feature vector $ASM_{feature}$ is fed to the model, which outputs $P_{predicted}$, the expected performance. This procedure is repeated individually for each benchmark to be evaluated: each evaluation is done on a benchmark and all its associated vector variants that were never seen by the model during the training. We remark that for the experiments of Sec. 6 and Sec. 7, models are trained on all variants of the 30 TC set, and evaluated on a fully distinct test set (CCSD and Stencil benchmark suites).

The running time of the training procedure depends on the compiler used, ICC being slower than GCC for our test suite. The total training time ranges from 30 minutes to 2 hours, depending on the configuration. For the evaluation of an unseen benchmark the total time is dominated by the time to compile all variants, and therefore depends on the number of variants. In our experiments it ranged from about 30 seconds to 10 minutes.

4.5. Learning Algorithms Evaluated

We implemented performance prediction models using six different machine learning algorithms available in Weka [Bouckaert et al. 2010]: Perceptron is an acyclic artificial neural network trained using back propagation; K* and IBk are both instance based learning algorithms which predict based on similar instances from the training set; M5P generates M5 model trees which are binary trees where each internal node compares an input value to a constant determined during training; SVM is a support vector machine algorithm using sequential minimal optimization; finally LR is linear regression. All these algorithms were used with the default parameter values provided with Weka.

5. ANALYSIS OF THE GENERATED MODELS

We now analyze the feature space and evaluate choices for selection of features to be used as inputs to the models and the resulting performance.

5.1. Correlation Between Features

To characterize the feature space, we performed a study of the coefficients of determination between the features described in Section 4.2, averaged for all vectorized variants of the CCSD application. This is summarized as a Hinton diagram shown in Figure 7.

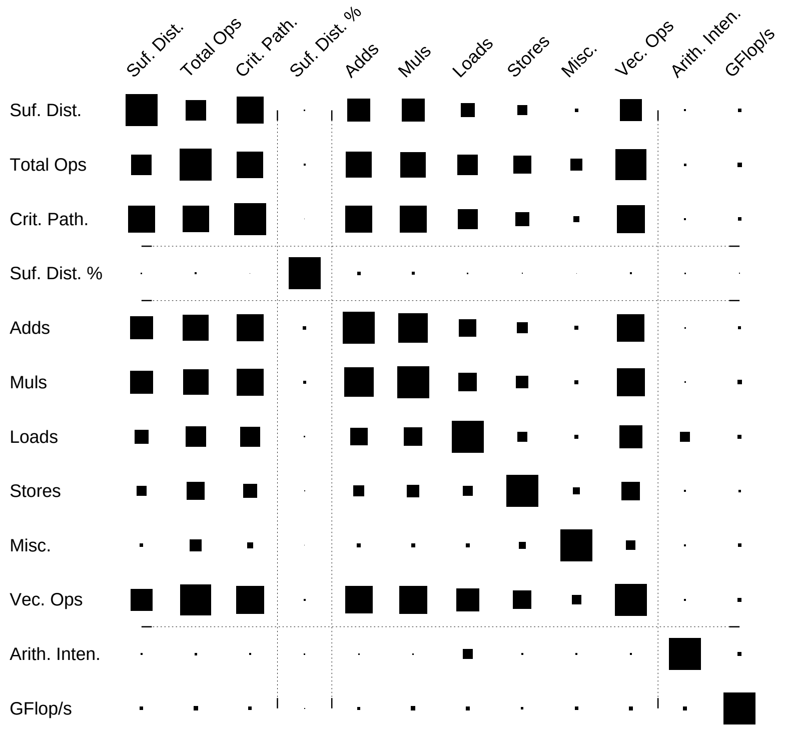

Fig. 7. Hinton diagram of the coefficient of determination between features averaged for all vectorized variants of CCSD

The most prominent aspect of the diagram is that none of the features are exceptionally well correlated with performance, motivating the need for combining multiple features to create an input set to the models. Looking at arithmetic intensity, it is among the best correlated to GFlop/s, our performance metric. Arithmetic intensity alone was used by Stock et al. in their cost model [Stock et al. 2011], and is a logical choice since the higher the arithmetic intensity, the less memory bound the computation. The performance achieved by using this feature alone is analyzed below, and is referenced as fs1 (feature set 1). We show that using fs1 as the only input to the model exhibits poor predictions.

Another category of features shows good correlation with performance, this is the set of vector operation count (Adds to Vec. Ops). We observe that those features are quite well correlated with each other, especially Muls, the number of vector multiply operations, is the most correlated to performance in this set. We analyze below the performance of a feature set fs2 that contains those features in addition to fs1, and show that using fs2 greatly improves the quality of the predictions.

The sufficient distance ratio feature is by far the least correlated to other features. As such it is able to provide information to the predictors which is unavailable from the other features. Although it is also the least correlated to performance, we show below that adding this feature improves the quality of the model. We analyze the set fs3 below, which contains this feature in addition to fs2.

Finally, the sufficient distance, critical path and total operation features are all highly correlated with each other, and show good correlation with performance. We analyze below the performance of the feature set fs4 that contains these features in addition to fs3. However we find that this does not improve the performance of the predictors over using only fs3.

5.2. Prediction Quality for the Feature Sets

To determine the quality of predictions, we compute an *efficiency metric* defined as follows:

$$\text{Efficiency} = \frac{\text{Performance of predicted best}}{\text{Performance of actual best}}$$

The efficiency is 100% if the predicted best variant is the actual best variant for the benchmark. The actual best variant was found by evaluating on the target machine the entire space of possible vectorized variants generated by our algorithm, and the one with the maximum measured performance is taken as the best variant.

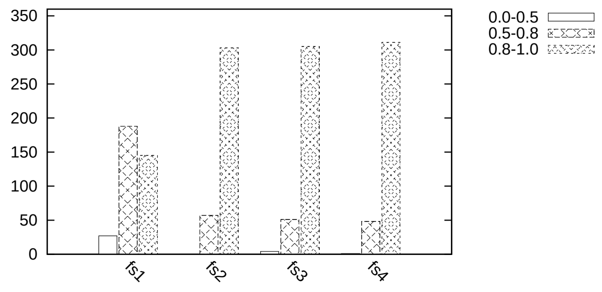

Fig. 8. Quality of predictions for each of the four feature sets over all configurations.

In Figure 8 we quantify, for each feature set considered, using the K* model, the number of benchmarks for which the predicted best achieves less than 50% efficiency (0-0.5), those achieving 50% to 80% (0.5-0.8) and those achieving more than 80% efficiency (0.8-1.0). The poor results using fs1 illustrates how arithmetic intensity alone is not a sufficient criterion to determine performance. This is consistently observed across all ML models using fs1. On the other hand, the relatively similar performance of fs2, fs3 and fs4 indicates that arithmetic intensity coupled with vector operation counts gathers the most important information to predict performance, as would be predicted by their relatively good correlation with performance as seen in Figure 7. Additional features in fs3 and fs4 provide minimal benefit. A complete analysis shows that for some specific combinations of ML model and configuration, fs4 performs marginally better than fs3. However, we conclude that fs3 is the most suitable feature set in our experiments, based on its ability to obtain the best average efficiency across many of the models and configurations, while limiting the required number of inputs.

5.3. Evaluation of the Models

We plot in Figure 9 the result of the predictive modeling using fs3, for three different configurations. We also plot the performance of ICC's auto-vectorization on the original source code, using the same efficiency metric, and the result of a random choice from the search space of variants.

We are able to significantly outperform ICC auto-vectorization by considering a large search space of transformations and making a better decision about which transformation to apply. Our models find performance dependencies between the transformations instead of predicting the effect of each transformation individually. Also, by considering vectorization along all dimensions instead of only those accessed in unit-stride, we are able to utilize the SIMD units of the processors where current compilers fail to find the best transformations to optimize the vectorized code.

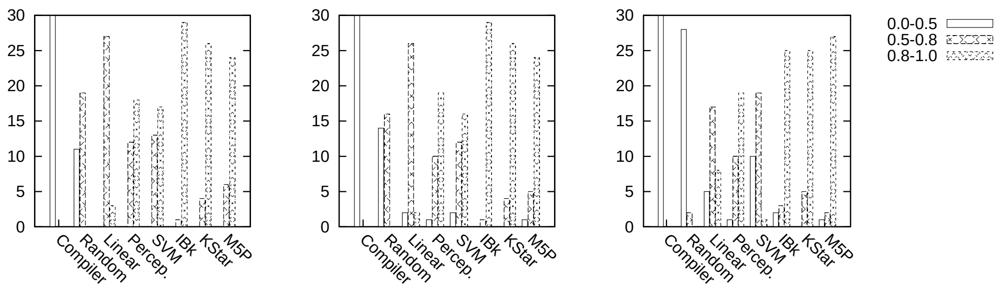

Fig. 9. Quality of the prediction, using the fs3 feature set, on `float` data using ICC. (a) is Nehalem with SSE, (b) is Sandy Bridge with SSE, (c) is Sandy Bridge with AVX.

The efficiency achieved by Random search reflects the variation in the quantity of good vectorized variants across configurations : for Sandy Bridge w/AVX, all but 1 benchmark in Random fail to exhibit more than 50% efficiency. For this configuration, the space contains fewer 'good' variants than with Nehalem. In contrast, our ML models, in particular instance-based learning algorithms such as IBk and KStar, achieve 80% efficiency or more for a vast majority of the benchmarks. We note that simpler classification models such as Multi-Layer Perceptron MLP and SVM, while still significantly outperforming Random selection, exhibit a lower prediction quality. However, a detailed analysis shows that while on average those models perform worse, for a few benchmarks they outperform the instance and tree based algorithms.

We conducted a similar study across all configurations, concluding that IBk, KStar and M5P are consistently the best ML models for accurately ranking the performance of the various vectorized variants. We report in Table I the efficiency of these three models across all twelve configurations. We abbreviate the configuration using four letters: N for Nehalem and S for Sandy Bridge, S for SSE and A for AVX, F for `float` and D for `double`, and I for ICC and G for GCC. When ties occur (a model may predict the same performance for two or more variants), we report the average efficiency of the variants which ranked first. Table I shows that no single model is consistently best across all configurations, although all three models perform similarly within any given configuration.

Table I. Average efficiency of the most successful individual models from leave one out analysis across configurations. **N**ehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Model	NSDG	NSDI	NSFG	NSFI	SADG	SADI	SAFG	SAFI	SSDG	SSDI	SSFG	SSFI
IBk	0.88	0.94	0.88	0.96	0.89	0.81	0.89	0.87	0.87	0.93	0.89	0.94
KStar	0.87	0.96	0.87	0.92	0.91	0.88	0.89	0.88	0.88	0.95	0.89	0.93
M5P	0.86	0.93	0.86	0.90	0.91	0.85	0.89	0.89	0.87	0.94	0.83	0.87

6. COMPOSING MODELS

We now provide in-depth experimental results for TCs from the CCSD application. For the remainder of the section, we use ML models that were trained on the set of 30 random TCs that we generated. We evaluate those models on the 19 unseen TCs from CCSD, that is, none of the TCs from CCSD were used for training the models.

6.1. Performance Analysis

A careful study of the individual results shows that while some models can correctly output an 80%+ variant for a given benchmark, other models may fail to do so. We illustrate this in Table II, which details the performance of the various models for Sandy Bridge with AVX, using `float` data and ICC, with the fs3 feature set. We use a description of the indices from the three arrays in the TCs to identify the benchmarks. For instance, `ij-ik-kj` represents a contraction of two 2D Tensors to produce a 2D Tensor, and the loop depth is 3 (that is, the number of different indices). The St-m column indicates the efficiency obtained by using Stock's static cost model [Stock et al. 2011] to select the vectorized variant.

Table II. Efficiency of the ML models on the CCSD test set, using Sandy Bridge with AVX, `float` data and ICC.

Tensor Contraction	ICC	Random	St-m	IBk	KStar	LR	M5P	MLP	SVM	Weighted Rank
ij-ik-kj	0.14	0.38	0.85	1.00	1.00	0.70	1.00	1.00	0.63	1.00
ij-ikl-ljk	0.10	0.32	0.05	0.97	0.76	0.81	0.76	0.73	0.81	0.97
ij-kil-lkj	0.10	0.31	0.55	1.00	0.89	0.67	1.00	0.67	0.67	1.00
ijk-ikl-lj	0.14	0.47	0.70	0.85	0.73	0.63	0.92	0.85	0.63	0.73
ijk-il-jlk	0.23	0.29	0.48	0.99	0.99	0.55	0.82	0.71	0.80	0.99
ijk-ilk-jl	0.25	0.34	0.23	0.99	0.99	0.69	0.98	0.81	0.80	0.99
ijk-ilk-lj	0.24	0.43	0.23	0.84	0.84	0.65	0.81	0.81	0.80	0.84
ijk-ilmk-mjl	0.08	0.25	0.75	0.98	0.70	0.66	0.38	0.70	0.78	0.98
ijkl-imjn-lnkm	0.06	0.35	0.05	0.86	0.84	0.73	0.76	0.86	0.58	0.86
ijkl-imjn-nlmk	0.10	0.35	0.86	0.89	0.96	0.77	0.77	0.79	0.57	0.96
ijkl-imkn-jnlm	0.05	0.27	0.05	0.96	0.90	0.62	0.78	0.67	0.51	0.96
ijkl-imkn-njml	0.09	0.31	0.28	0.93	0.93	0.94	0.88	0.92	0.78	0.93
ijkl-imln-jnkm	0.05	0.28	0.04	0.92	0.92	0.70	0.87	0.74	0.67	0.92
ijkl-imln-njmk	0.05	0.28	0.77	0.88	0.92	0.44	0.92	0.77	0.44	0.92
ijkl-imnj-nlkm	0.05	0.33	0.03	0.83	0.99	0.48	0.88	0.75	0.48	0.89
ijkl-imnk-njml	0.08	0.39	0.54	0.78	0.74	0.87	0.80	0.83	0.37	0.74
ijkl-minj-nlmk	0.12	0.46	0.44	0.91	0.86	0.65	0.90	0.92	0.65	0.86
ijkl-mink-jnlm	0.09	0.37	0.02	0.73	0.91	0.46	0.94	0.82	0.46	0.91
ijkl-minl-njmk	0.09	0.46	0.24	1.00	1.00	0.63	0.97	0.61	0.30	1.00
Average	0.11	0.35	0.38	0.91	0.89	0.67	0.85	0.79	0.62	0.92

While IBk and KStar are consistently the two best models, we observe numerous cases where one is able to achieve a significantly better efficiency than the other. For instance, for `ijk-ilmk-mjl` IBk achieves 98% efficiency while KStar is limited to 70%. The situation can also be reversed, as with `ijkl-mink-jnlm`. In order to benefit from the ability of different models to predict best transformations for different benchmarks, we evaluated the use of a second-order model combining the predictions of the two best individual models.

6.2. The Weighted-Rank Model

Each individual model produces a predicted performance for all variants of a program, leading to a rank of the variants according to this prediction. That is, given a model M and a variant v, we obtain R_v^M, the rank of the variant according to its predicted performance. The variant v with the best predicted performance has $R_v^M = 1$, the second predicted best has rank 2, etc.

We also observe from Table II that the simpler models do perform significantly worse on average across all benchmarks. Hence we do not consider all models, but only the best performing ones to develop the composite model. Our composite model combines the ranks obtained by a variant on IBk and KStar, to obtain a composite rank WR for a variant:

$$WR_v = \gamma_1.R_v^{IBK} + \gamma_2.R_v^{K*}$$

The factor γ_i represents the contribution of each individual models to the final vote. For instance, choosing $\gamma_i = \frac{1}{2}$ creates a 'fair' voting model, where both ML models have the same decision power. We have evaluated a fair voting model and observed that it is not consistently better than the best individual model, IBk.

To build the WeightedRank model, we used Linear Regression to find the γ_i coefficient values. The problem learned is:

$$(R_v^{IBK}, R_v^{K*}) \to WR_v$$

and the model is trained using for WR_V the actual rank of each variant. We used Weka's Linear Regression model, with the default parameter values. We used the 30 TCs and all their associated variants for the training, and report the evaluation on the 19 CCSD Tensor Contractions in Table II.

The Weighted Rank model performs on average marginally better than the two individual models, with an average efficiency of 92%. However, its most important contribution is to improve over the ‘worst’ performing benchmarks when considering configurations individually. In all but 3 cases, the Weighted Rank model outputs the maximal efficiency of either KStar or IBk, thus effectively selecting the model that predicts the better variant.

We performed a complete analysis across all twelve configurations, and found that the Weighted Rank model is consistently better on average than each individual model. The results are shown in Table III, which reports the average efficiency of all models, across all twelve configurations, for the CCSD test set. We have also experimented with other composite models, including a Weighted Rank model using all six individual models that uses a neural network to learn the weight function; it did not perform as well as the 2-way Weighted Rank model described above.

Table III. Average efficiency of the ML models on the CCSD set, across all configurations. **N**ehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Configuration	ICC/GCC	Random	St-m	IBk	KStar	LR	M5P	MLP	SVM	Weighted Rank
NSDG	0.42	0.64	0.82	0.86	0.85	0.83	0.81	0.84	0.83	0.86
NSDI	0.37	0.66	0.78	0.95	0.96	0.80	0.92	0.93	0.93	0.95
NSFG	0.31	0.53	0.79	0.91	0.86	0.64	0.86	0.80	0.63	0.90
NSFI	0.19	0.54	0.84	0.92	0.89	0.72	0.89	0.88	0.84	0.92
SADG	0.27	0.51	0.75	0.84	0.89	0.70	0.87	0.83	0.72	0.85
SADI	0.22	0.38	0.44	0.82	0.86	0.67	0.88	0.69	0.75	0.88
SAFG	0.21	0.49	0.65	0.81	0.82	0.68	0.81	0.81	0.67	0.81
SAFI	0.11	0.35	0.38	0.91	0.89	0.67	0.85	0.79	0.62	0.92
SSDG	0.43	0.67	0.86	0.88	0.85	0.83	0.78	0.85	0.75	0.87
SSDI	0.33	0.67	0.79	0.95	0.95	0.75	0.93	0.94	0.91	0.94
SSFG	0.33	0.53	0.82	0.88	0.87	0.63	0.88	0.78	0.63	0.88
SSFI	0.20	0.52	0.84	0.92	0.89	0.67	0.81	0.80	0.78	0.92
Average	0.28	0.54	0.73	0.88	0.88	0.71	0.85	0.83	0.75	0.89

6.3. Performance Evaluation on CCSD

We complete our experimental evaluation by reporting in Table IV the performance, in GFlop/s, obtained by using the WeightedRank model to select at compile-time an effective vectorized variant for the 19 CCSD tensor contractions. As previously discussed, none of the CCSD contractions were seen during training. We compare these results against the compiler’s auto-vectorization (`icc -fast` or `gcc -O3`) on the original input code. We report in the avg column the mean performance across all 19 contractions. To illustrate the variance across the benchmarks, we also report min, the performance of the TC with the lowest GFlop/s across the benchmark suite, as well as max, the one with the highest GFlop/s. Table IV highlights the very strong benefit in using our model on top of Stock’s vector synthesis algorithm, when compared to ICC’s automatic vectorization. Despite the small fraction of high-performance variants in this complex search space, as illustrated in Section 3, the WeightedRank model is able to achieve single-core improvements ranging from 2× up to 8× on average. We observe that for some TCs, complex loop permutation sequences are required, along with vectorizing (one of) the reduction loop(s). To the best of our knowledge, ICC’s auto-vectorization is limited to vectorizing dimensions accessed in unit-stride, and thus do not vectorize some of the benchmarks.

Interestingly, Table IV also shows that GCC (`gcc -O3`) can outperform ICC for some configurations; this is especially the case for the max column for the original code. For the original code, manual investigation of a few benchmarks showed that GCC applies unroll-and-jam more effectively than ICC. The best absolute performance is found on Sandy Bridge with AVX, where ICC attains 43GFlop/s, which is about 80% of the machine peak. However, for Sandy Bridge with SSE, GCC achieves 21.4GFlop/s while ICC tops at 21 GFlop/s (77% of machine peak). GCC’s ability to slightly outperform ICC for some benchmarks was observed across various configurations, but ICC performs consistently better on average when compiling vector intrinsics code for the Nehalem and Sandy Bridge systems.

Table IV. GFlop/s obtained for the CCSD benchmarks, using compiler auto-vectorization and our WeightedRank model. For each configuration, min is the performance of the slowest CCSD TC, max the performance of the fastest CCSD TC, avg is the average performance across all 19 CCSD TCs. **N**ehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Configuration	Compiler			Weighted Rank			Improv.
	min	avg	max	min	avg	max	
NSDG	1.38GF/s	3.02GF/s	8.48GF/s	3.55GF/s	6.02GF/s	6.96GF/s	2.00×
NSDI	1.30GF/s	2.82GF/s	5.29GF/s	6.69GF/s	7.24GF/s	8.11GF/s	2.57×
NSFG	1.39GF/s	4.34GF/s	16.70GF/s	9.22GF/s	11.77GF/s	14.24GF/s	2.71×
NSFI	1.30GF/s	2.71GF/s	5.98GF/s	6.77GF/s	12.13GF/s	14.30GF/s	4.47×
SADG	2.31GF/s	4.55GF/s	11.63GF/s	10.35GF/s	14.26GF/s	17.88GF/s	3.13×
SADI	1.89GF/s	3.92GF/s	6.69GF/s	11.50GF/s	14.64GF/s	22.23GF/s	3.73×
SAFG	2.40GF/s	6.87GF/s	24.47GF/s	14.69GF/s	25.84GF/s	35.47GF/s	3.76×
SAFI	1.89GF/s	4.15GF/s	9.79GF/s	24.92GF/s	33.18GF/s	43.30GF/s	7.99×
SSDG	2.31GF/s	4.57GF/s	11.62GF/s	5.47GF/s	8.86GF/s	10.35GF/s	1.94×
SSDI	1.89GF/s	3.90GF/s	6.69GF/s	10.06GF/s	10.97GF/s	12.68GF/s	2.81×
SSFG	2.40GF/s	6.89GF/s	24.74GF/s	10.02GF/s	16.96GF/s	21.41GF/s	2.46×
SSFI	1.89GF/s	4.16GF/s	9.57GF/s	8.93GF/s	16.58GF/s	20.97GF/s	3.99×

6.4. Performance Evaluation Without Synthesis of Vectorized Code

To complete the evaluation of the ML-based approach, we perform a complementary evaluation of the predictors. We consider the case of the CCSD benchmarks with only unroll-and-jam factors and loop permutation transformations being applied, both of which are implemented in modern compilers such as ICC and GCC. That is, *we do not use the customized generation of vector intrinsics* but simply provide the compiler with variants where unroll-and-jam and loop permutation were chosen by our model and applied on the source code. The objective is to highlight the limitations of the current state-of-the-art heuristics implemented in the compiler to compute the unroll-and-jam factors and loop permutation, and show the improvement that can be obtained over the current state-of-the-art production compilers simply by using our ML model in place of the internally implemented heuristics.

Table V shows the performance obtained for the CCSD application, with variants chosen from this restricted search space, for the Intel ICC compiler, averaged for each of the six configurations.

Table V. Performance (Perf.) in GFlop/s and improvement (Imp.) of the WeightedRank predictor used only for unroll-and-jam factors and loop permutation, over ICC default auto-vectorization heuristic. **N**ehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble

Tensor	NSD		NSF		SAD		SAF		SSD		SSF	
Contraction	Perf.	Imp.	Perf.	Imp.	Perf.	Imp.	Perf.	Imp.	Perf.	Imp.	Perf.	Imp.
ij-ik-kj	8.11	2.13	11.11	2.94	9.01	1.38	10.94	1.80	14.30	2.40	19.79	3.09
ij-ikl-ljk	5.18	2.47	7.40	3.54	5.90	1.97	12.69	4.14	9.44	3.14	11.62	3.69
ij-kil-lkj	6.67	3.14	10.58	5.01	5.86	1.88	10.16	3.22	11.13	3.57	17.37	5.49
ijk-ikl-lj	6.76	2.09	11.51	3.84	12.30	2.80	21.83	3.98	9.59	2.18	19.13	3.44
ijk-il-jlk	7.31	1.37	16.87	2.87	18.38	2.71	23.05	2.53	9.65	1.46	27.24	2.92
ijk-ilk-jl	7.23	1.36	15.78	2.62	17.24	2.58	21.92	2.24	9.64	1.45	27.74	2.89
ijk-ilk-lj	7.61	1.43	14.69	2.48	15.53	2.35	27.04	2.88	9.45	1.42	22.97	2.40
ijk-ilmk-mjl	5.03	2.17	10.13	4.67	10.06	3.20	10.16	3.59	9.38	2.86	16.22	5.43
ijkl-imjn-lnkm	6.77	4.97	9.52	7.31	8.64	4.25	18.80	9.34	11.51	5.77	16.91	8.64
ijkl-imjn-nlmk	5.77	2.21	10.86	4.63	7.48	1.99	18.33	5.53	9.63	2.56	17.22	5.13
ijkl-imkn-jnlm	6.82	5.29	9.31	6.93	8.48	4.60	16.69	9.04	10.28	5.40	16.10	8.13
ijkl-imkn-njml	6.19	3.15	12.88	6.03	9.56	3.46	19.10	6.47	10.04	3.52	19.06	6.43
ijkl-imln-jnkm	6.36	5.03	7.19	5.34	8.44	4.44	19.27	9.86	10.55	5.63	12.97	6.76
ijkl-imln-njmk	5.51	2.12	9.46	7.26	8.93	2.38	17.32	8.31	8.94	2.40	14.90	7.50
ijkl-imnj-nlkm	6.33	2.43	9.58	7.22	8.43	2.27	18.16	9.88	9.39	2.56	16.63	8.31
ijkl-imnk-njml	4.66	2.24	12.62	5.93	9.88	3.32	29.09	10.18	7.95	2.63	22.47	7.66
ijkl-minj-nlmk	5.93	1.67	9.64	3.66	7.81	1.68	16.94	4.33	9.96	2.08	16.06	4.09
ijkl-mink-jnlm	6.02	2.42	10.38	4.14	8.26	2.33	17.52	5.05	9.92	2.79	16.43	4.57
ijkl-minl-njmk	5.57	2.44	12.09	5.41	10.50	3.13	33.51	9.90	9.34	2.84	20.28	6.27
Average:	6.31	2.64	11.14	4.83	10.04	2.77	19.08	5.91	10.01	2.98	18.48	5.41

The ML models were trained using the same feature set as above, by evaluating numerous variants where the production compiler (Intel ICC here) was used to generate the final vector code. With an average improvement of 2.6× to 5.9× over the heuristic implemented in the ICC auto-vectorization pass, our models are clearly capable of achieving good performance by using optimization parameters that are internally accessible in modern compilers. The heuristics guiding these optimization in production compilers could be immediately improved by using our models.

7. EVALUATION ON STENCIL COMPUTATIONS

We complete our evaluation by examining the capability of the machine learning models to rank-order the performance of vectorized variants of several stencil computations. We note that *contrary to Tensor Contractions, the algorithm we employed to generate vectorized stencil variants is restricted*, and thus the final performance is not fully representative of effectively vectorized stencil codes. We used a 'naive' version of Stock's vectorization algorithm for the sole purpose of generating a search space of vectorized variants, on which the ML models can be evaluated. Nevertheless, we obtain good performance improvements over the compiler's auto-vectorization in the vast majority of cases.

The stencils selected for the test suite were chosen from [Deitz et al. 2001]: Partial derivatives (disofive, disothree, drowthree); Biharmonic operator (dbigbiharm, dlilbiharm); NAS MG (dresid, drprjthree); and Noise cleaning (inoiseone, inoisetwo). All are 2D or 3D stencils. We used the criteria that the stencils each have only a single pass, fewer than 25 points, and at least one non-trivial coefficient. We did not apply any time-tiling on the considered stencils.

7.1. Vectorization Algorithm

Stencils differ from TCs in many regards. The vectorization algorithm of [Stock et al. 2011] was modified to fit the domain of stencil computations. An example of a simple 1-dimensional stencil is:

```
for (i=0; i<N; i++)
  B[i] = A[i-1] + 2*A[i]+ A[i+1];
```

Vectorization for this code is done by loading three vector registers from `A`, multiplying the middle vector by 2, summing the vectors, and storing them back into `B`. In all the stencils considered, there are only two arrays, `A` as the input and `B` as the output. In general, stencils always have the same ordering of accessors, and thus both arrays always share the same unit-stride accessor. As described below, these factors significantly reduce the space of possible vectorizations, and thus the stencil benchmarks we consider only have 12 to 233 vectorized variants each.

Vectorized loop: Because both arrays share the same unit-stride accessor, there is no ambiguity in choosing which dimension to vectorize, and thus for all stencils we only consider vectorization along the unit-stride accessing dimension. This simplifies the vector operations that need to be considered, and there are only standard vector load, stores, and arithmetic in the loops, but the scalar coefficients must be splatted to registers (which can be hoisted outside the loops).

Loop permutation: All loops in the stencils considered are parallel, and thus all loop permutations are valid. However, because there are only 2 or 3 loops, the search space of loop permutations is much smaller compared to TCs having 3 to 6 loops. Furthermore, the benefit of being able to hoist operations out of the innermost loop is nonexistent since all memory accesses in stencils depend on all loops.

Unroll-and-Jam: In the case of TCs, register reuse was gained from accesses which were not dependent on the unrolled loop, but, as stated before, stencils do not have such accesses. However, stencil are still able to benefit from the unroll-and-jam transformation as some vectors may be reused as different points in future iterations. In the presented example, if vectors of length 2 are used, then the value of `A[i+1]` will be reused as `A[i-1]` in the next iteration since `A[(i+2)-1]=A[i+1]`.

7.2. Overview of the Search Space Complexity

In contrast to the performance distribution of TC variants when using Stock's algorithm, the performance distribution of the considered stencils shows a much narrower gap between the best and worst vectorized variants. This is illustrated in Figure 10, which plots the distribution of three representative stencil benchmarks, across three distinct configurations (Nehalem/SSE, Sandy Bridge/SSE and Sandy Bridge/AVX).

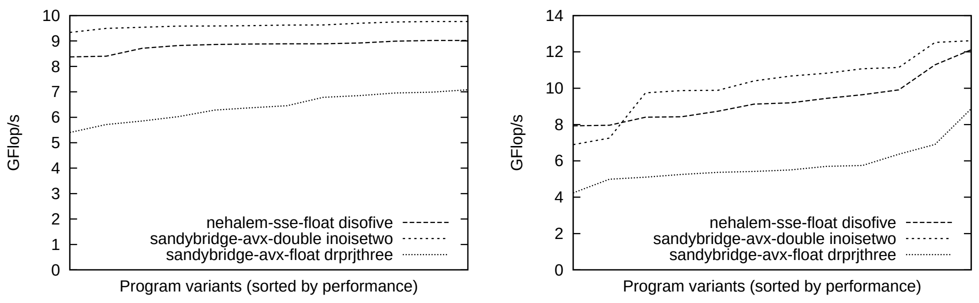

Fig. 10. Performance distribution of three representative stencils. (Left) using Intel ICC, (Right) using GNU GCC.

Figure 10 plots on the left the distribution when using the ICC compiler. For `disofive` and `inoisetwo`, the distribution is flat as all variants have about the same performance. Nevertheless there is a significant performance improvement over ICC's auto-vectorization on the original code: the best variant for `disofive` is 1.8$\times$ faster than the auto-vectorized code with ICC, which achieves 4.8GFlop/s. The fact that all variants have roughly the same performance indicates that ICC is performing aggressive low-level optimizations on the intrinsics code, that possibly undo some transformations (e.g., unroll-and-jam and loop permutation).

While our search space still exhibits interesting performance improvement, it is clear that for all benchmarks that have a flat distribution, even purely random selection among the variant will be very effective, *de facto* reducing the need of complex machine learning techniques. This observation is not true for all stencil benchmarks, and in particular when using the GCC compiler. Figure 10 plots on the right the performance of the same benchmarks and same hardware configuration, but using GCC. A much wider performance variation is seen, for all three benchmarks. We also observe a significant performance improvement over GCC's auto-vectorization; Sandy Bridge using single-precision AVX averages 2.86$\times$ improvement across the benchmarks.

7.3. Experimental Results

To evaluate the quality of the predicting models we have built, we generated vectorized variants of all nine stencils in our benchmark suite, and evaluated them with the models *that were trained on the Tensor Contraction training set*. That is, similarly to the CCSD evaluation, none of the stencil benchmarks were seen during the training of the models.

We report in Table VI the average efficiency across all nine stencils of the six considered ML models and the Weighted Rank model, across all twelve configurations, in addition to the compiler auto-vectorization efficiency as well as random selection of a vectorized variant.

As indicated by the performance distributions in Figure 10, Random performs very well when using ICC. Nevertheless the Weighted Rank model does consistently outperform Random, and is on average better than any individual model. Its capability to identify a good vectorized variant is illustrated in particular when using the GCC compiler, when the performance distribution offers greater challenges to the optimization selection process. For instance, for Nehalem/SSE using `float`, the Weighted Rank model outperforms all individual model by a significant margin, achieving 89% efficiency for an average 2.8$\times$ performance improvement over GCC's auto-vectorization.

We conclude our experimental evaluation of stencil benchmarks by showing in Table VII the performance, in GFlop/s, obtained when using the WeightedRank model to select at compile-time an effective vectorized variant for the 9 stencils. This table follows the format of Table IV.

We observe a significant performance improvement over compiler auto-vectorization, although not as high as compared to the CCSD case. This is mostly due to the vectorization algorithm we used to

Table VI. Average efficiency of the ML models on the Stencil set, across all configurations. **N**ehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Configuration	ICC/GCC	Random	IBk	KStar	LR	M5P	MLP	SVM	Weighted Rank
NSDG	0.60	0.81	0.95	0.87	0.64	0.80	0.84	0.64	0.93
NSDI	1.05	0.94	0.95	0.95	0.96	0.93	0.94	0.94	0.95
NSFG	0.32	0.74	0.84	0.72	0.60	0.62	0.85	0.60	0.89
NSFI	0.41	0.94	0.95	0.95	0.96	0.93	0.93	0.95	0.96
SADG	0.41	0.80	0.85	0.82	0.68	0.75	0.74	0.68	0.86
SADI	0.79	0.93	0.92	0.92	0.92	0.93	0.94	0.93	0.92
SAFG	0.33	0.91	0.90	0.93	0.91	0.90	0.91	0.91	0.92
SAFI	0.41	0.95	0.96	0.96	0.94	0.95	0.93	0.94	0.96
SSDG	0.56	0.83	0.97	0.95	0.62	0.74	0.73	0.62	0.99
SSDI	1.03	0.97	0.97	0.97	0.97	0.97	0.96	0.96	0.97
SSFG	0.32	0.80	0.80	0.81	0.72	0.72	0.86	0.71	0.84
SSFI	0.42	0.95	0.96	0.96	0.96	0.96	0.95	0.96	0.96
Average	0.55	0.88	0.92	0.90	0.82	0.85	0.88	0.82	0.93

Table VII. GFlop/s obtained for the Stencil benchmarks, using compiler auto-vectorization and our WeightedRank model. For each configuration, min is the performance of the slowest stencil, max the performance of the fastest stencil, avg is the average performance across all 9 stencils. **N**ehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Configuration	Compiler			Weighted Rank			Improv.
	min	avg	max	min	avg	max	
NSDG	2.17GF/s	3.35GF/s	4.12GF/s	3.48GF/s	5.34GF/s	6.91GF/s	1.59×
NSDI	4.26GF/s	5.59GF/s	6.65GF/s	4.33GF/s	5.24GF/s	6.97GF/s	0.94×
NSFG	3.20GF/s	3.78GF/s	4.45GF/s	7.22GF/s	10.50GF/s	12.52GF/s	2.77×
NSFI	2.76GF/s	4.20GF/s	5.10GF/s	8.85GF/s	9.97GF/s	12.26GF/s	2.37×
SADG	3.41GF/s	4.65GF/s	5.52GF/s	6.58GF/s	9.86GF/s	13.39GF/s	2.12×
SADI	6.44GF/s	7.89GF/s	9.02GF/s	7.90GF/s	9.23GF/s	11.49GF/s	1.17×
SAFG	4.40GF/s	5.05GF/s	6.13GF/s	11.36GF/s	14.44GF/s	19.08GF/s	2.86×
SAFI	4.17GF/s	5.85GF/s	7.02GF/s	10.41GF/s	13.74GF/s	16.07GF/s	2.35×
SSDG	3.41GF/s	4.66GF/s	5.52GF/s	6.19GF/s	8.44GF/s	10.26GF/s	1.81×
SSDI	6.48GF/s	7.87GF/s	8.88GF/s	6.21GF/s	7.61GF/s	9.97GF/s	0.97×
SSFG	4.36GF/s	5.02GF/s	6.14GF/s	9.51GF/s	13.41GF/s	16.05GF/s	2.67×
SSFI	4.17GF/s	5.86GF/s	7.02GF/s	12.38GF/s	13.48GF/s	16.01GF/s	2.30×

generate the variants, whose original design target was tensor contractions. The best absolute performance is attained with Sandy Bridge/AVX, using `float` with the GCC compiler, reaching 19GFlop/s (35% of machine peak). For both Nehalem/SSE and Sandy Bridge/AVX using `double`, ICC outperforms the best vectorized variant by a small margin. However, for all other cases, a performance improvement ranging on average between 1.17× and 2.86× can be observed.

We expect this performance improvement to greatly increase with a better suited stencil vectorization algorithm. We recall that our vectorization strategy is designed to operate on assembly code, independent of the transformation algorithm. It is thus expected that our models can be reused as-is on any customized stencil vectorization algorithm developed in the future.

8. RELATED WORK

Automatic vectorization has been the subject of extensive study in literature [Kennedy and Allen 2002; Wolfe 1996]. Numerous previous works have focused on generating effective code dealing with hardware alignment and stride issues [Eichenberger et al. 2004; Nuzman et al. 2006; Fireman et al. 2007; Larsen and Amarasinghe 2000; Larsen et al. 2002], outer-loop vectorization [Nuzman and Zaks 2008] and multi-platform auto-vectorization [Nuzman and Henderson 2006; Hohenauer et al. 2009]. The automatic vectorizer of GNU GCC implements many of these techniques [Nuzman and Henderson 2006; Nuzman et al. 2006; Nuzman and Zaks 2008] and thus represents the state-of-the-art. Since Intel's ICC is a closed source compiler, it is more difficult to assess what is currently implemented. Nevertheless its very good performance compared to GCC for many codes suggests advanced techniques for automatic vectorization have been implemented.

Stock et al. showed that transformations such as unroll-and-jam and loop permutation have a critical impact on the performance of TCs, in conjunction with dedicated vector intrinsics code generation [Stock et al. 2011]. It can outperform ICC by a large factor for the tensor contractions in the MADNESS kernel. To the best of our knowledge, this is the best performing method for SIMD execution of the MADNESS tensor contraction kernels. Here we have generalized the algorithm to arbitrary tensor contractions, and have shown the deficiencies of its cost model, in particular for the AVX instruction set. Our machine learning models can achieve up to $2.5\times$ better performance than Stock's model.

We consider a set of transformations that are a superset of the SIMD-related transformations considered by Hall et al. in their auto-tuning work [Chen et al. 2008; Chen et al. 2007], where only unroll-and-jam and loop permutation pertaining to SIMD optimization are considered. We have observed that additional dedicated optimizations such as register transpose and intrinsics code generation, which are considered by our approach, can provide up to $2\times$ additional performance improvement over the compiler's auto-vectorization. In addition, in contrast to previous work on auto-tuning [Chen et al. 2008; Yi and Qasem 2008; Tiwari et al. 2009], our work *operates at compile time and does not require the execution of any variants on the machine*. It is a purely model-driven approach.

Trifunovic et al. proposed an analytical cost model for evaluating the impact of loop permutation and loop strip-mining [Trifunovic et al. 2009] on vectorization. While it is applicable to TCs, in contrast to our work it does not take into account the interplay of subsequent compiler passes (e.g., vector code generation, instruction selection, scheduling and register allocation) and does not consider critical optimizations for performance, such as unroll-and-jam and register transpose.

Deciding the enabling or disabling of loop unrolling was done by Monsifrot et al. [Monsifrot et al. 2002] using decision tree learning, and was one of the early efforts on using machine learning to tune a high-level transformation. Numerous other works considered the use of machine learning to drive the optimization process [Kisuki et al. 2000; Cooper et al. 1999; Cooper et al. 2002; Franke et al. 2005; Haneda et al. 2005; Agakov et al. 2006]. None of these works considered the advanced vectorization techniques that are used in this paper. Cavazos et al. address the problem of predicting good compiler optimizations by using performance counters to automatically generate compiler heuristics [Cavazos et al. 2007]. That work was limited to the traditional optimization space of the PathScale compiler. Further, the program to be optimized by the compiler first had to be executed (without optimization) to determine the feature vector of performance counters that were then input to the trained ML model to predict the best optimization sequence. In contrast, our approach is a purely compile-time technique.

9. CONCLUSION

Most production compilers today have automatic vectorization capability for multi-core processors with short-vector SIMD instruction sets, but the achieved performance is often significantly lower than machine peak. A primary reason is that the space of possible loop transformations is very large and effective models do not exist to select the best transformations. In this paper, we have developed a machine-learning-based performance model to guide short-vector SIMD compiler optimization. The use of the performance model for vectorizing tensor contraction computations demonstrated significantly better performance than production vectorizing compilers, and the model was shown to be beneficial for the domain of stencil computations.

Acknowledgment

This work was supported in part by the by the U.S. National Science Foundation through awards 0811781, 0926688, and by the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127.

REFERENCES

AGAKOV, F., BONILLA, E., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M., THOMSON, J., TOUSSAINT, M., AND WILLIAMS, C. 2006. Using machine learning to focus iterative optimization. In *CGO*.

BAUMGARTNER, G., BERNHOLDT, D., COCIORVA, D., HARRISON, R., HIRATA, S., LAM, C.-C., NOOIJEN, M., PITZER, R., RAMANUJAM, J., AND SADAYAPPAN, P. 2002. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Supercomputing*.

BOUCKAERT, R. R., FRANK, E., HALL, M. A., HOLMES, G., PFAHRINGER, B., REUTEMANN, P., AND WITTEN, I. H. 2010. WEKA–experiences with a java open-source project. *Journal of Machine Learning Research 11*, 2533–2541.

CAVAZOS, J., FURSIN, G., AGAKOV, F. V., BONILLA, E. V., O'BOYLE, M. F. P., AND TEMAM, O. 2007. Rapidly selecting good compiler optimizations using performance counters. In *CGO*.

CHEN, C., CHAME, J., AND HALL, M. 2008. CHiLL: A framework for composing high-level loop transformations. Tech. Rep. 08-897, U. of Southern California.

CHEN, C., SHIN, J., KINTALI, S., CHAME, J., AND HALL, M. 2007. Model-guided empirical optimization for multimedia extension architectures: A case study. *IPDPS*.

COOPER, K. D., SCHIELKE, P. J., AND SUBRAMANIAN, D. 1999. Optimizing for reduced code space using genetic algorithms. In *LCTES*.

COOPER, K. D., SUBRAMANIAN, D., AND TORCZON, L. 2002. Adaptive optimizing compilers for the 21st century. *J. Supercomput. 23,* 1, 7–22.

CRAWFORD, T. AND SCHAEFER III, H. 2000. An Introduction to Coupled Cluster Theory for Computational Chemists. In *Reviews in Computational Chemistry*. Vol. 14. 33–136.

DEITZ, S. J., CHAMBERLAIN, B. L., AND SNYDER, L. 2001. Eliminating redundancies in sum-of-product array computations. In *ICS*.

DUBACH, C., JONES, T. M., BONILLA, E. V., FURSIN, G., AND O'BOYLE, M. F. 2009. Portable compiler optimization across embedded programs and microarchitectures using machine learning. In *MICRO*.

EICHENBERGER, A., WU, P., AND O'BRIEN, K. 2004. Vectorization for simd architectures with alignment constraints. In *PLDI*.

FIREMAN, L., PETRANK, E., AND ZAKS, A. 2007. New algorithms for simd alignment. In *CC*.

FRANKE, B., O'BOYLE, M., THOMSON, J., AND FURSIN, G. 2005. Probabilistic source-level optimisation of embedded programs. In *LCTES*.

HANEDA, M., KNIJNENBURG, P. M. W., AND WIJSHOFF, H. A. G. 2005. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT*.

HARRISON, R. J., FANN, G. I., GAN, Z., YANAI, T., SUGIKI, S., BESTE, A., AND BEYLKIN, G. 2005. Multiresolution computational chemistry. *Journal of Physics: Conference Series 16,* 1, 243.

HIRATA, S. 2003. Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A 107,* 46, 9887–9897.

HOHENAUER, M., ENGEL, F., LEUPERS, R., ASCHEID, G., AND MEYR, H. 2009. A simd optimization framework for retargetable compilers. *ACM TACO 6,* 1.

KENNEDY, K. AND ALLEN, J. 2002. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann.

KISUKI, T., KNIJNENBURG, P. M. W., AND O'BOYLE, M. F. P. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT*.

LARSEN, S. AND AMARASINGHE, S. P. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*.

LARSEN, S., WITCHEL, E., AND AMARASINGHE, S. P. 2002. Increasing and detecting memory address congruence. In *PACT*.

MONSIFROT, A., BODIN, F., AND QUINIOU, R. 2002. A machine learning approach to automatic production of compiler heuristics. In *AIMSA*. 41–50.

NUZMAN, D. AND HENDERSON, R. 2006. Multi-platform auto-vectorization. In *CGO*.

NUZMAN, D., ROSEN, I., AND ZAKS, A. 2006. Auto-vectorization of interleaved data for simd. In *PLDI*.

NUZMAN, D. AND ZAKS, A. 2008. Outer-loop vectorization: revisited for short simd architectures. In *PACT*.

STOCK, K., HENRETTY, T., MURUGANDI, I., HARRISON, R., AND SADAYAPPAN, P. 2011. Model-driven simd code generation for a multi-resolution tensor kernel. In *IPDPS*.

TIWARI, A., CHEN, C., CHAME, J., HALL, M., AND HOLLINGSWORTH, J. K. 2009. A scalable autotuning framework for computer optimization. In *IPDPS*.

TRIFUNOVIC, K., NUZMAN, D., COHEN, A., ZAKS, A., AND ROSEN, I. 2009. Polyhedral-model guided loop-nest auto-vectorization. In *PACT*.

WOLFE, M. 1989. More iteration space tiling. In *SC*.

WOLFE, M. J. 1996. *High Performance Compilers For Parallel Computing*. Addison-Wesley.

YI, Q. AND QASEM, A. 2008. Exploring the optimization space of dense linear algebra kernels. In *LCPC*.

YUKI, T., RENGANARAYANAN, L., RAJOPADHYE, S., ANDERSON, C., EICHENBERGER, A., AND O'BRIEN, K. 2010. Automatic creation of tile size selection models. In *CGO*.