Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets

Daniel S. McFarlin
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA USA 15213
dmcfarli@ece.cmu.edu

Volodymyr Arbatov
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA USA 15213
arbatov@ece.cmu.edu

Franz Franchetti
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA USA 15213
franzf@ece.cmu.edu

Markus Püschel
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
pueschel@inf.ethz.ch

ABSTRACT

The well-known shift to parallelism in CPUs is often associated with multicores. However another trend is equally salient: the increasing parallelism in per-core single-instruction multiple-date (SIMD) vector units. Intel's SSE and IBM's VMX (compatible to AltiVec) both offer 4-way (single precision) floating point, but the recent Intel instruction sets AVX and Larrabee (LRB) offer 8-way and 16-way, respectively. Compilation and optimization for vector extensions is hard, and often the achievable speed-up by using vectorizing compilers is small compared to hand-optimization using intrinsic function interfaces. Unfortunately, the complexity of these intrinsics interfaces increases considerably with the vector length, making hand-optimization a nightmare. In this paper, we present a peephole-based vectorization system that takes as input the vector instruction semantics and outputs a library of basic data reorganization blocks such as small transpositions and perfect shuffles that are needed in a variety of high performance computing applications. We evaluate the system by generating the blocks needed by the program generator Spiral for vectorized fast Fourier transforms (FFTs). With the generated FFTs we achieve a vectorization speed-up of 5.5–6.5 for 8-way AVX and 10–12.5 for 16-way LRB. For the latter instruction counts are used since no timing information is available. The combination of the proposed system and Spiral thus automates the production of high performance FFTs for current and future vector architectures.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*Code generation, Optimization*

ICS'11, May 31–June 4, 2011, Tuscon, Arizona, USA.

General Terms

Performance

Keywords

Autovectorization, super-optimization, SIMD, program generation, Fourier transform

1. Introduction

Power and area constraints are increasingly dictating microarchitectural developments in the commodity and high-performance (HPC) CPU space. Consequently, the once dominant approach of dynamically extracting instruction-level parallelism (ILP) through monolithic out-of-order microarchitectures is being supplanted by designs with simpler, replicable architectural features. This trend is most evident in the proliferation of architectures containing many symmetrical processing cores. Such designs provide for flexible power management and reduced area by trading dynamic ILP for static, software-defined thread-level parallelism. A similar trade-off is occurring with the steadily increasing vector-width and complexity of single-instruction-multiple-data (SIMD) vector instruction sets.

AVX and Larrabee. Intel's recent AVX and Larrabee (LRB) architectures feature 256-bit and 512-bit vector-lengths respectively; architectures with 1024-bit long vectors are already planned [2, 24]. Vector functional units and vector registers are regular structures which are fairly easy to replicate and expand. Like multiple cores, vector units provide for flexible power management in that individual vector functional units can be selectively idled. SIMD instructions also represent a form of scalar instruction compression thereby reducing the power and area consumed by instruction decoding. Collectively, this architectural trend towards multiple cores and wide vectors has fundamentally shifted the burden of achieving performance from hardware to software.

Programming SIMD extensions. In contrast to multiple cores, SIMD architectures require software to explicitly encode fine-grain data-level parallelism using the SIMD instruction set. These SIMD instruction sets are quickly evolving as vendors add new instructions with every CPU generation, and SIMD extensions are incompatible across CPU vendors. Consequently, explicitly vectorized code is hard to write and inherently non-portable. The complex-

ity of SIMD instruction sets complicates hand-vectorization while auto-vectorization just like auto-parallelization poses a continuing challenge for compilers.

The latest version of production compilers (Intel C++, IBM XL C, and Gnu C) all contain autovectorization technology [17, 36, 15] that provides speed-up across a large class of computation kernels. However, for many kernels like the fast Fourier transform (FFT) and matrix multiplication, the results are usually suboptimal [8] since optimal vectorization requires algorithm knowledge or there are simply too many choices that the compiler cannot evaluate.

Much of the difficulty in vectorization lies in the instructions required to transform and keep data in vector form. These shuffle or permutation instructions are generally the most complex and expensive operations in the SIMD instruction set. They tend to scale poorly, may not support arbitrary permutations and their parameters become increasingly non-obvious to use, especially with wider vector units. From a performance point of view, shuffle instructions are the overhead imposed by vectorization, which prevents the perfect speedup linear in the vector length. Consequently, minimizing the number and cost of shuffles is crucial.

Contribution. This paper makes two key contributions. First, we present a super-optimization infrastructure that takes as input the instruction set specification and automates the discovery of efficient SIMD instruction sequences for basic data reorganization operations such as small matrix transpositions and stride permutations. These are required, for example, by many high performance computing kernels including the FFT and linear algebra kernels.

Second, we incorporate this infrastructure into the library generator Spiral to generate the reordering blocks needed for FFT vectorization [8]. This approach effectively automates the porting of FFTs to new vector architectures. We then evaluate efficacy by automatically generating vectorized FFTs for AVX and LRB. We demonstrate speed-ups (measured using runtime or instruction counts) of 5.5–6.5 for 8-way AVX and 10–12.5 for 16-way LRB. We also compare the AVX code against Intel's IPP. For LRB, no benchmarks are available at the time of writing.

Besides this main contribution, with AVX and Larrabee it now becomes possible to study efficiency and overhead of vectorization methods across a range of vector lengths: 2, 4, 8, and 16 for single-precision floating-point. We include such a study for Spiral's FFT vectorization.

2. Related Work

The work in this paper extends the SIMD support in the Spiral system. It is related to vectorization techniques developed for traditional vector computers, SIMDization techniques developed for short length SIMD vector instruction sets, superoptimization, and SIMD support by program generators like FFTW.

SIMD instructions in Spiral. The inspiration for this work comes from earlier work extending Spiral [29, 7, 9] to SIMD vector architectures. Spiral is a domain-specific library generator that automates the production of high performance code for linear transforms, notably the discrete Fourier Transform [35]. Previous efforts to extend Spiral to SIMD vector architectures are described in [6, 8, 10]. Spiral's approach breaks the vectorization problem into two stages. First, rewriting produces SIMD FFTs [6, 8, 10] that reduce the problem to a small set of basic reordering operations (matrix transpositions of small matrices held in SIMD registers). Second, a small code generator is used to produce short instruction sequences for these operations [11] given only the instruction set specification as input. Unfortunately, experiments showed that the method in [11] does not scale (i.e., is too expensive) to AVX and LRB. Hence the motivation for this paper, which offers a replacement for [11] that is designed for both longer vector lengths and more complex instruction sets.

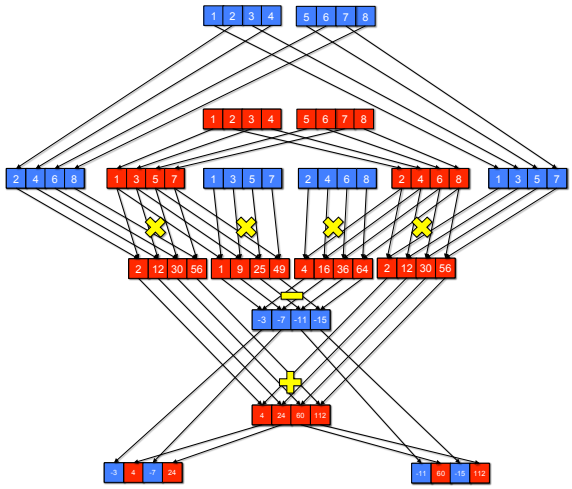

Figure 1: The dataflow of an SSE-vectorized kernel. The computation is an element-wise multiplication of two complex input arrays (blue and red) of length four in interleaved format.

Vectorization. Automatic vectorization has been the subject of extensive study in the literature. Two excellent references are [21, 37]. Vectorization becomes (again) increasingly important for SIMD extensions like Larrabee and the latest versions of SSE (SSE 4.1) that allow for efficient implementation of gather/scatter operations and large data caches, since the conditions on such architectures are similar to traditional vector computers.

SIMDization. Originating from SIMD within a register (SWAR) [5, 34], SIMDization was recognized as a hybrid between vectorization and instruction level parallelism extraction [1]. Recent advances in compilation techniques for SIMD vector instruction sets in the presence of alignment and stride constraints are described in [4, 28]. SIMD instruction extraction for two-way architectures aimed at basic blocks is presented in [22]. This technique is included in FFTW 2.1.5 [13, 12] and has shown good performance improvements across multiple two-way SIMD extensions. FFTW3 [14] contains SIMD codelets for SSE and AltiVec, supporting vector lengths of 2 and 4.

Superoptimization. The classic paper on super-optimization is [26] while [3] presents a modern approach that is close in spirit to our own. A dataflow-graph and integer-linear programming based approach to finding SIMD permutations was described by [23] and is similar to our approach though it is unclear what sort of vectorization efficiencies are attained. The approach explored in [30] also focuses on SIMD permutations with an emphasis on linear transforms including the FFT. However, only small kernels (max size: 64-point FFT) are investigated and the overall scalability of their solution to larger vector widths and larger kernels is not addressed. The difficulties of optimizing for a wide range of SIMD vector architectures are well explored in [27, 16].

3. Vectorization Efficiency and Motivation

Vectorization overhead impacts even simple kernels. Consider the case of the element-wise product of two arrays each containing four complex element in interleaved form (alternating real and imaginary parts). On a traditional scalar processor, this kernel requires 24 floating point operations: 4 multiplications, 1 addition and 1 subtraction per complex product. Figure 1 shows the dataflow

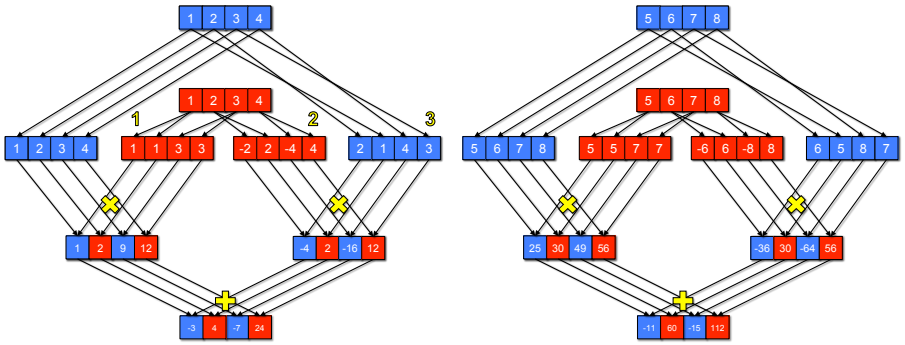

Figure 2: The same computation as in Fig. 1 performed on LRB using again a vector width of size four.

of a vectorized version of this kernel on Intel's SSE SIMD architecture with a vector width of four (4-way). The six vectorized arithmetic instructions (yellow symbols) in this figure are straightforward but the de- and re-interleaving of real and imaginary elements is less obvious and requires six shuffle instructions as overhead.

We quantify the vectorization efficiency by calculating the ratio of total floating point operations in the scalar kernel to the total number of vector instructions in the vectorized kernel. In Fig. 1 the efficiency is 24/(6 + 6) = 2. An ideal vectorization (not possible in this case) would yield 24/6 = 4 = vector length as efficiency.

Vectorization efficiency is a good first-order indicator of performance and enables the study of different vector architectures even if the architecture is not yet available.

Figure 2 gives a first idea of the difficulties in vectorization. It shows the same kernel as in Fig. 1 this time 4-way vectorized for LRB (only 4 out of the 16 slots in the vector register are shown for simplicity; the labels 1–3 are explained later). The data flow is non-intuitive but now has an overhead of only 4 shuffles and thus an improved efficiency of 24/(6 + 4) = 2.4.

4. AVX and Larrabee

We give a brief overview of Intel's AVX instruction set and a more in-depth view of LRB, with focus on the Larrabee new instructions (LRBni).

4.1 Advanced Vector Extension

Intel's latest extension to the SSE family is the Advanced Vector Extension (AVX) [2]. It extends the 128-bit SSE register into 256-bit AVX registers, that consist of two 128-bit lanes. An AVX lane is an extension of SSE4.2 functionality, including fused multiply-add instructions and three-operand instructions. AVX operates most efficiently when the same operations are performed on both lanes. Cross-lane operations are limited and expensive. AVX defines 4-way 64-bit double precision, 8-way 32-bit single precision, and integer operations.

AVX shuffle instructions. AVX essentially implements SSE's 128-bit shuffle operation for both lanes, with some extensions to support parameter vectors. In addition it defines one cross-lane shuffle operation. This leads to higher shuffle-overhead since many operations now require both cross-lane and intra-lane shuffling. In Listing 4.1 we show the intrinsic function prototypes of 4-way double and 8-way single AVX shuffle instructions. The parameter space of AVX shuffle instructions is much larger compared to 2-way and 4-way SSE instructions.

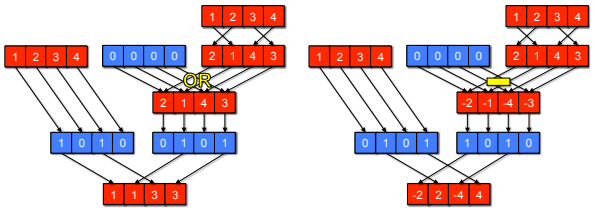

Figure 3: An expanded view of the LRB swizzle and writemask features used to sign-change and reorder vectors for complex multiplication. The left and right image corresponds to labels 1 and 2 in Figure 2, respectively. Each is a single LRB instruction.

Listing 1: AVX shuffle instructions.

```
__m256d _mm256_unpacklo_pd(__m256d a, __m256d b);
__m256d _mm256_unpackhi_pd(__m256d a, __m256d b);
__m256d _mm256_shuffle_pd(__m256d a, __m256d b, const int select);
__m256d _mm256_permute2_pd(__m256d a, __m256d b, __m256i control, int imm);
__m256d _mm256_permute2f128_pd(__m256d a, __m256d b, int control);
__m256d _mm256_permute_pd(__m256d a, int control);

__m256 _mm256_unpacklo_ps(__m256 a, __m256 b);
__m256 _mm256_unpackhi_ps(__m256 a, __m256 b);
__m256 _mm256_permute2f128_ps(__m256 a, __m256 b, int control);
__m256 _mm256_permute2_ps(__m256 a, __m256 b, __m256i control, int imm);
__m256 _mm256_shuffle_ps(__m256 a, __m256 b, const int select);
__m256 _mm256_permute_ps(__m256 a, int control);
__m256 _mm256_permutevar_ps(__m256 a, __m256i control);
```

4.2 Larrabee

Intel's LRB architecture can be described as a chip-level multiprocessor containing a large number of cache-coherent, in-order x86 cores. LRB leverages legacy code through compatibility with the standard Intel x86 32/64 scalar instruction set but features a novel and powerful SIMD vector instruction set known as LRBni (Larrabee New Instructions). We restrict our discussion of LRB to the architectural features most relevant to vectorization and refer the reader to [31, 24] for a more comprehensive discussion.

The LRB core is a dual-pipeline architecture that shares many similarities with the well known P5 Pentium architecture. LRB's vector unit and LRBni, however represent a significant departure from previous commodity vector architectures. To elaborate, we return to Figure 2. Label 1 shows data reordering on the second vector input. Label 2 shows data reordering and a sign-change of the same input vector. Label 3 shows data reordering being performed on the remaining input vector. This reordering operation is folded into the subsequent computation while Labels 1 and 2 require one instruction each. All told, there are 4 reordering instructions in this kernel compared to 6 reordering instructions in the SSE kernel shown in Figure 1.

LRBni ISA. We now briefly discuss the LRBni vector exten-

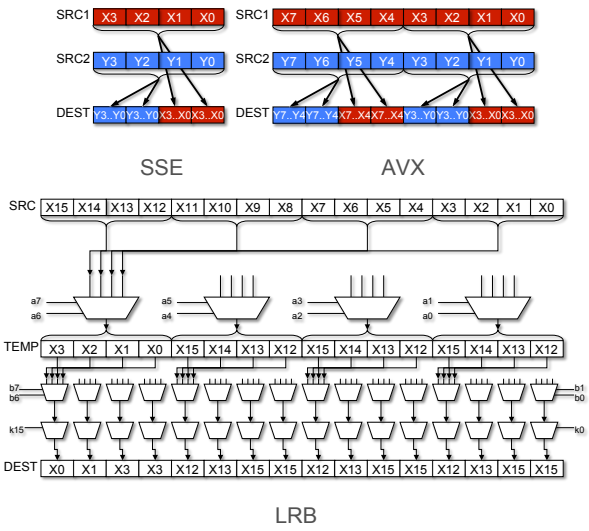

Figure 4: The LRBni vector extension.

sion. The 512-bit registers are grouped into 4 128-bit lanes. The 512-bit registers can hold either 8 double-precision numbers or 16 single-precision numbers. The 16-way vector can be interpreted as 4-by-4 matrix. Instructions (see Figure 3) contain multiple parts: 1) Source operands can be reordered (within lanes) before being used. 2) All standard arithmetic operations are supported (including addition, subtraction, multiplication, fused multiply-add and add-sub) and performed in parallel on all vector slots. 3) A selector describes which of the result vector slots are actually written into the destination, and which results are discarded. In addition, step 1 is exposed as instructions as well. LRBni instructions are complicated with may parameters, and the intrinsic interface decouples LRBni instructions into multiple intrinsics to make programming manageable. We show examples of LRBni instructions in Listing 4.2 Below we discuss some of the LRBni instructions important for this paper in more detail.

Swizzles. Returning to Figure 3 we note that the reduction in reordering instructions is achievable due to the dedicated reorder HW in LRB's vector unit. This HW provides for a limited set of non-destructive shuffles, known as swizzles, which can be performed on one in-register vector operand per vector instruction. The swizzles used to implement Labels 1 and 2 in Figure 2 are shown in Figure 3. Label 1's implementation is shown on the left and uses a binary-OR taking three vector inputs; vector instructions in LRB are ternary. The first and third operands are sourced from the same register. We binary-OR the swizzled third operand with the zero vector and merge the result with the first vector operand in accordance with a writemask. This writemask is stored in one of the mask registers and is an optional argument to most vector instructions. It dictates which elements in the third vector operand are overwritten.

Listing 2: Implementation of complex multiplication using LRB intrinsics

```
// v0, v1: input vectors of interleaved complex floats
__m512 zero  =  _mm512_setzero();
__m512 s0    =  _mm512_swizzle_r32(v0, _MM_SWIZ_REG_CDAB);
__m512 reals =  _mm512_mask_or_pi(v0, 0xAAAA, zero, s0);
__m512 imags =  _mm512_mask_sub_ps(v0, 0x5555, zero, s0);
__m512 t0    =  _mm512_mul_ps(reals, v1);
__m512 s1    =  _mm512_swizzle_r32(v1, _MM_SWIZ_REG_CDAB);
__m512 res   =  _mm512_madd231_ps(t0, imags, s1);
```

The computation required for Label 2 is similar with the use

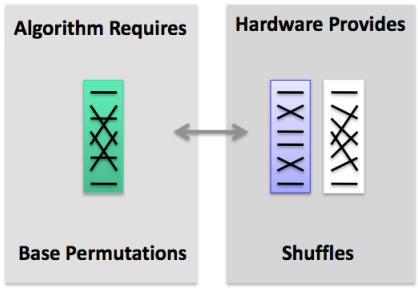

Figure 5: The problem of mapping basic permutations to vector shuffle instructions

of a subtraction instruction to affect a sign-change and a different writemask. For completeness, we show the C code with LRB intrinsics that implements the entire kernel in Listing 4.2. This code listing also shows the use of one of LRB's many fused multiply-add (FMA) instructions. The combination of FMAs and swizzles enables LRB's complex multiplication kernel to attain a vectorization efficiency of 3 for the simplified 4-way case; the 16-way case has the same relative efficiency at 12 floating-point operations/vector instruction.

Broadcasts, gathers, and memory operations. LRB's vector unit also features extensive support for L1-cache-to-register operations. Of particular interest is the replicate hardware which enables efficient scalar broadcasts from memory and can be used with virtually all vector instructions. Scatter/gather functionality exists in the form of two instructions which take a base address and a vector of offsets. Another useful pair of instructions are those which can pack/unpack data and handle unaligned memory accesses. For LRB's remaining non-reordering vector instructions we refer the reader to [25].

LRBni shuffle operations. Finally, there is the unary LRB shuffle, depicted at the bottom of Figure 4. Because the reorder hardware only supports a limited set of shuffles we must rely on the dedicated shuffle instruction for more general, arbitrary reorderings. As stated before, shuffle instructions are generally the most expensive vector instructions and do not particularly scale well; encoding a fully general unary shuffle for a 16-way architecture requires 64 bits. If this 64 bit value is stored directly in the shuffle instruction it complicates the instruction decoders. Conversely, storing this value in a separate, scalar register complicates the datapath.

5. Superoptimizer for Data Permutations

In this section we explain how we automatically derive short (efficient) instruction sequences to implement important basic data reorganizations (permutations). The data to be permuted fits into a few vector registers and the data permutations we consider have a regular structure. Two important examples are 1) the interleaving/deinterleaving of two vectors of complex numbers into/from one vector of real parts and one vector of imaginary parts, and 2) the in-register transposition of a square matrix whose number of rows is the vector length. Both can be viewed as transpositions of a small matrix. The motivation for considering these permutations is from [6], which shows that these are the only in-register shuffles needed to implement FFTs. The same permutations are also important in numerical linear algebra kernels and many other functions.

Fundamentally, we are faced with the challenge of mapping the basic permutations needed to a class of hardware reordering instructions that we refer to collectively as shuffles (see Figure 5).

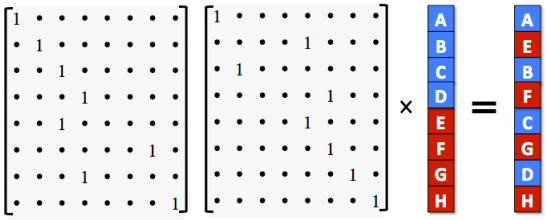

Figure 6: The basic permutation (perfect shuffle) that interleaves two 4-way vectors represented as a product of two binary matrices and two input vectors

Our goal is to generate efficient sequences of these reordering instructions in order to minimize the vectorization overhead. Efficient sequences are difficult to generate due to the complexity of shuffles and other reordering instructions in wide-vector architectures. To overcome these challenges, we developed an infrastructure to automate the generation of efficient reordering sequences.

Problem statement. Given a vector ISA and its shuffle operations and a transposition of a small matrix that is held in a few vector registers. We aim to generate a short instruction sequence that implements this matrix transposition with the minimal number of in-register shuffles.

Approach. We find the shortest instruction sequence that implements the required matrix transposition by 1) modeling instructions as binary matrices, 2) instruction sequences as products of binary matrices, 3) and transpositions as *stride permutation* matrices [20, 11]. Checking that an instruction sequence implements a certain transposition then is equivalent of checking that a product of matrices evaluates to the required stride permutation matrix. Based on this observation we build a superoptimizer based on matrix factorization to find the shortest instruction sequence that implements the required permutation.

5.1 Implementing the Superoptimizer

Formalization. The key insight to our approach is that we can represent permutations and the shuffle instructions that implement them as binary matrices [11]. This can be seen in Figure 6, which shows the permutation that reinterleaves a real vector and an imaginary vector into a vector of complex numbers as a product of two binary matrices operating on two concatenated input vectors of size 4.

This particular factorization of the permutation maps to two different sets of instructions on Intel SSE each with different performance characteristics. With the binary matrix representation in hand, we can formalize the generation of shuffle sequences as equivalent to finding a binary matrix factorization (BMF) of a given permutation matrix, P_m where each factor, F_i is a valid shuffle instruction in the vector instruction set architecture (ISA). For efficient shuffle sequences we generally want the least expensive sequence for some per-instruction cost function cost:

$$\text{minimize } \textstyle\sum_{i=0}^{n} \text{cost}(F_i)$$

$$\text{subject to } P_m = F_0 F_1 \cdots F_{n-1} \ \wedge \ F_0, \ldots, F_{n-1} \in \text{ISA}$$

Binary matrix factorization (BMF). While BMF is a convenient formalization it is known to be NP-hard [33]. The problem is further complicated by our need for exact factorizations and factors with specific matrix dimensions (2ν x 2ν for a vector width ν); existing solvers generally find approximate factorizations with factors of arbitrary dimension [32]. We therefore elected to go in the other direction by generating sequences of binary matrices where each binary matrix corresponds to a particular configuration of a particular shuffle instruction. The code implementing this description is shown in Listing 3. We then evaluate the sequence by matrix multiplying the sequence elements and comparing the Hamming distance to the desired base permutation matrix.

Super-optimization. In a sense, we are performing a kind of super-optimization on a limited number of complex shuffle instructions [26]. While conceptually straightforward, this approach, like general super-optimization, has limitations. One basic problem is that we have no indication of the minimal sequence size required to implement a particular base permutation. Furthermore, even though the matrices in the candidate sequences are all derived from a small set of shuffle instructions, we are still left with a very large search space; there are four billion variants of the unary LRB shuffle alone. More concretely, the code shown in Listing 3 produces k^n different sequences of shuffle instructions for a sequence length of n and vector ISA with k shuffle instructions. Considering the number of variants per shuffle instruction (or the number of different matrices each shuffle instruction represents) gives us a total number of different instruction sequences of:

$$\sum_{i=0}^{k^n-1} \left(\prod_{j=0}^{n} |S_{i,j}| \right)$$

where $S_{i,j}$ is the j^{th} shuffle instruction in the i^{th} instruction sequence and $|S_{i,j}|$ is the number of shuffle variants.

Guided search. Our solution for searching this space efficiently is a vector-instruction aware, heuristic-guided search system that can be integrated with the program generator Spiral, which is itself already a form of expert system.

Sequence length estimation. An example heuristic uses the vector width of the architecture, combined with a representation of the intrinsic interface of the most general shuffle instruction in the vector ISA to help determine a likely lower bound on the minimum number of shuffle instructions required to implement a particular base permutation. For example, a fully general unary shuffle can be used to implement the reinterleaving of an vector of real and imaginary parts to a complex vector in about four instructions.

Sequence culling. Other heuristics allow us to cull individual shuffle configurations from consideration (e.g. the identity shuffle) as well as instruction sequences (e.g. interleaving followed by immediate de-interleaving). The system also requires a generator program for each prospective shuffle instruction. The generator produces a binary matrix for a given configuration of the shuffle.

Listing 3: Building Sequences (Schedules) of Shuffle Instructions

```
// idx: index in the current schedule, numInstrs: # of shuf instrs in ISA
// schedLen: size of an instruction sequence, sched: the existing schedule
// instrs: array of shuffle instructions
void build_schedules(int idx,int numInstrs,int schedLen,schedule_t* sched){
   for(int i=0;i<numInstrs;++i){
        schedule_t nSched = new schedule_t(schedLen);
     // append the existing schedule
     nSched.add(sched);
     // add the ith instruction to the schedule
     nSched.add(idx,instrs[i]);
     if(idx+1 == schedLen){
       // finished creating the schedule
       // enqueue the schedule for processing
       threadQueue.enqueue(nSched);
     }else{
       // recursively build the remaining schedules
       build_schedules(idx+1,numInstrs,schedLen,nSched);
     }
   }
}
```

μ-op decomposition. We also decompose complex instructions into multiple stages; encoding each stage as a separate shuffle instruction to provide much finer grain resolution for the pattern

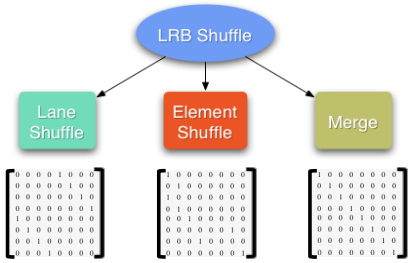

Figure 7: A decomposition of the LRB shuffle instruction into μ-ops

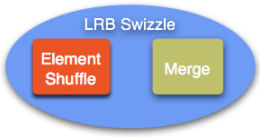

Figure 8: μ-op fusion: this particular element shuffle and merge can be implemented by one swizzle instruction

matching and rewriting that we employ to cull candidates and perform other optimizations. We show an example of this decomposition for the LRB shuffle in Figure 7 where we refer to individual stages as μ-ops. The μ-ops depicted in the figure are generally sufficient to describe most reordering operations. Ideally, we hope to subsume a sequence of these μ-ops with a less expensive instruction, performing in effect a type of strength reduction by "μ-op fusion." Figure 8 shows two μ-ops originating from a LRB shuffle which can be performed by a less expensive swizzle.

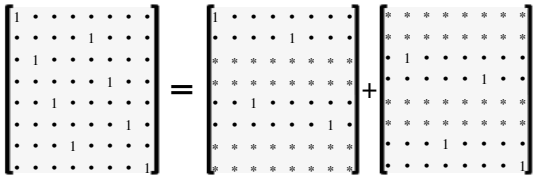

Figure 9: Partitioning of a stride permutation matrix (reinterleaving a real and an imaginary vector into a complex vector) for a 4-way vector architecture

Base permutation partitioning. Another technique used to accelerate search involves partitioning a base permutation matrix into a sequence of "hollow" matrices. These matrices have the same dimensions and initial contents as the source base permutation matrix. However, certain rows are converted into "don't care" rows; an example is shown in Figure 9.

Searches are then performed on a set of these "hollow" matrices in parallel using reduced length instruction sequences. The hope is that the shorter instruction sequences found for each "hollow" matrix can be inexpensively combined in a later pass to produce the full base permutation matrix. Because these shorter instruction sequences potentially contain many redundancies we employ a prefix tree to filter out common sub-sequences. The search mechanism is fully parallelized and can run on shared-memory and cluster machines and relies on a hand-tuned binary multiplication kernel shown in Listing 4. We describe its performance and efficacy in generating vectorized code presently.

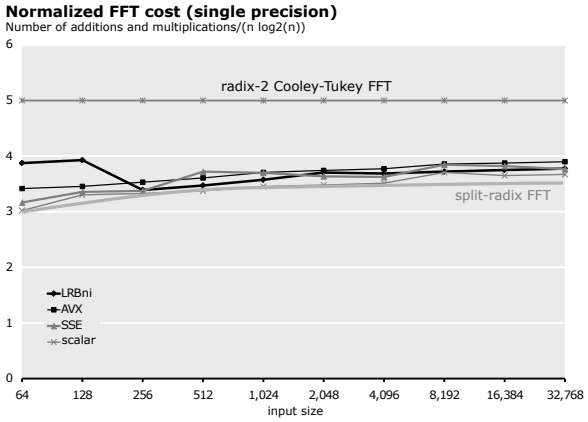

Figure 10: Number of additions and multiplications for Spiral-generated FFTs

Listing 4: Binary matrix multiplication kernel optimized for permutation matrices

```
// c: output binary matrices, a,b: input binary matrices, all matrices
// are represented as an  array of bit-vectors with n as the vector width
void binmat_mult(unsigned int* a, unsigned int* b, unsigned int* c){
  unsigned int mask = exp2(n-1);
  for(int j=0;j<n;++j){
    unsigned int bb = b[j];
    unsigned int w = 0;
    unsigned int m = mask;
    #pragma unroll(n)
    for(int i=0;i<n;++i) {
      unsigned int v = a[i] & bb; bool f = !(v & (v - 1)) && v;
      w = (w & ~m) | (-f & m); m >>= 1;
    }
    c[j] = w;
  }
}
```

6. Experimental Results

In this section we evaluate both the performance of the superoptimizer and the quality of the generated code. The latter is assessed by using the generated permutations inside Spiral-generated FFT code, whose efficiency and performance is then evaluated. For LRB we use instruction counts since no hardware is available. For compilation, we used Intel icc version 12.0.2 on Linux with the -O3 optimization flag as well as unrolling and alignment pragmas.

Generated FFTs. We experimentally evaluate our generator with 1D complex FFTs, both with 2-power sizes $n = 64, \ldots, 32768$ as well as for kernel sizes $n = 2, 3, \ldots, 32$. To do this we connected our generator with the program generation system Spiral effectively inserting the generated permutations into the generated FFT code. On LRB, Spiral's feedback-driven search uses the number of vector instructions instead of runtime as cost function. All FFTs used are $O(n \log(n))$ algorithms. For small sizes we also perform an experiment with direct $O(n^2)$ implementation.

First, we evaluate the impact of vectorization on the mathematical operations count (counting additions and multiplication) of the generated FFTs. Vectorization introduces overhead in the form of superfluous multiplications by 1 and superfluous additions with 0 due to data packing in registers. A degenerate strategy for minimizing this overhead could generate kernels with higher operations count and simpler structure. Figure 10 shows that this is not the approach with Spiral-generated FFTs. The y-axis shows the number of mathematical additions and multiplications of Spiral-generated vectorized code divided by $n \log_2 n$ where n is the input size shown on the x-axis. As upper bound we show the radix-2

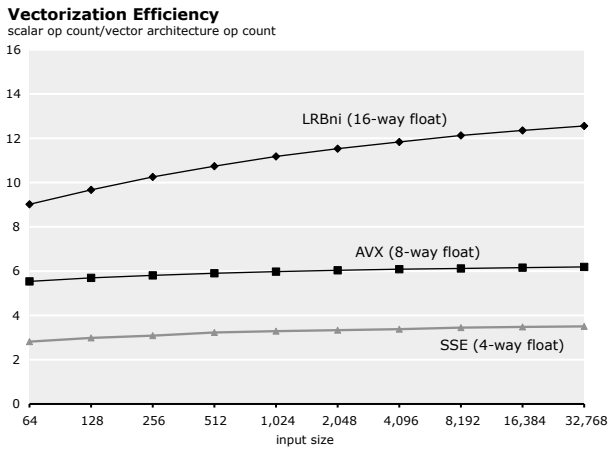

Figure 11: Spiral's vectorization efficiency across three Intel SIMD architectures

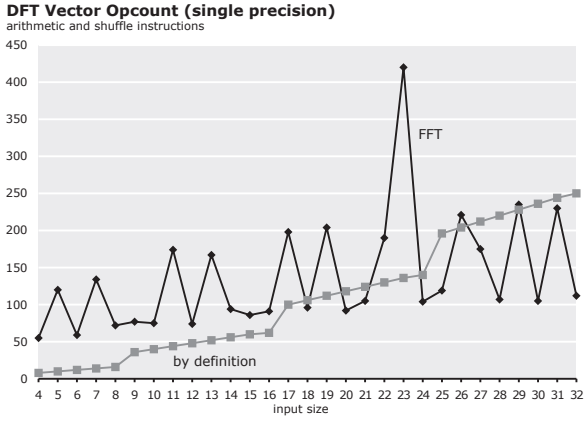

Figure 12: Comparison of vector operation counts for FFTs and "DFTs by definition" on LRB

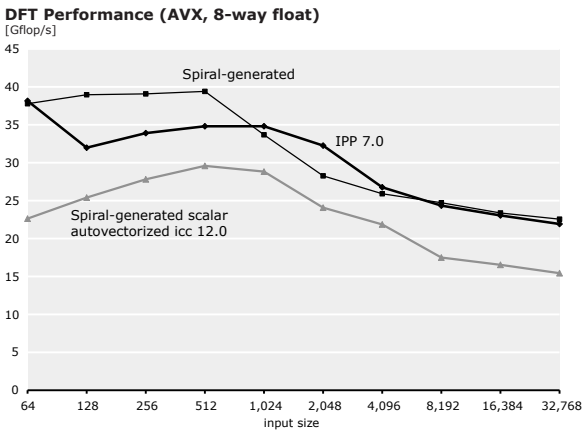

Figure 13: Comparison of 8-way AVX vectorized DFT implementations

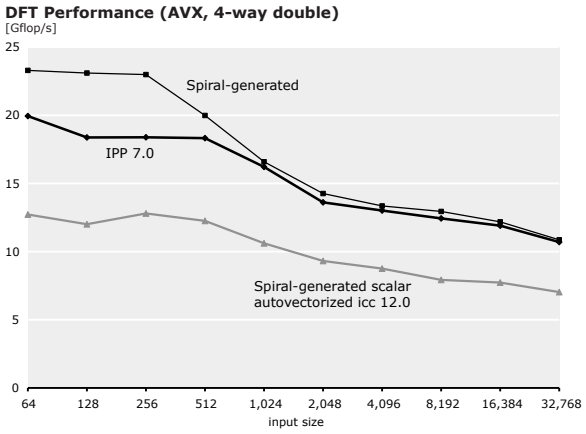

Figure 14: Comparison of 4-way AVX vectorized DFT implementations

Cooley-Tukey FFT, which requires $5n\log_2(n)$ operations. This number is usually (and also in this paper) used for FFT performance comparisons in Gflop/s, thus slightly overestimating the performance. As lower bound we show the split-radix FFT which requires $4n\log_2(n) - 6n + 8$ many operations. The plot shows that Spiral-generated vector code on all architectures is close to the latter.

Vectorization efficiency. We now examine the vectorization efficiency, defined in Section 3, of the Spiral-generated vectorized FFT. Ideally, the vectorization efficiency should approach the architecture's vector width. However, due to the required shuffles, this is not achievable. Figure 11 shows that across vector architectures and lengths, we achieve an efficiency of up to about 80% of the vector length. For AVX and LRB, this is mainly due to the superoptimizer presented in this paper.

We also note that AVX ramps up faster due to its more general, binary shuffle but LRB eventually achieves the same relative efficiency.

Next we investigate the trade-off between fast $O(n\log_2 n)$ algorithms and direct $O(n^2)$ computations for small kernel sizes. For these sizes the shuffles required by the fast algorithms can become prohibitive while the regular, FMA-friendly structure of the matrix-vector product allows for high efficiency. Figure 12 shows that indeed up to a size of about $n = 20$, the direct computation is preferable, even though the mathematical operations count (counting only additions and multiplications) is inferior. The reason is in LRB's dedicated replicate HW, which enables efficient scalar broadcasts and FMA instructions which are well-suited for a direct computation.

Evaluation against FFT libraries. We cannot evaluate our FFT implementations against state-of-the-art third-party FFT implementations using the vectorization efficiency metric. The Intel Integrated Performance Primitives (IPP) [18] and Math Kernel Library (MKL) [19] would be the ideal base line for comparisons, but are only distributed as binaries; thus, instruction counts are not available. The recent (Jan 2011) release of hardware implementing the AVX ISA allows for a runtime comparison. Figures 13 and 14 show a runtime performance comparison of 4-way (double precision) and 8-way (single precision) AVX vectorized FFTs from Intel IPP 7.0 with those generated by Spiral on a 3.3 GHz Intel Core i5-2500. Spiral's AVX performance compares well with IPP 7.0 on the full range of DFT sizes. Note, that this early platform implementing the AVX ISA does not feature support for FMAs. FFTW [14] is available in source code and thus amenable to instruction statistics, but at this point only supports 2-way double precision 4-way

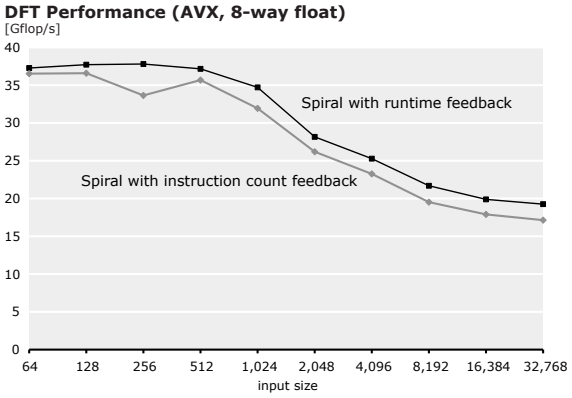

Figure 15: Comparison of 8-way AVX vectorized DFT implementations generated using different search metrics

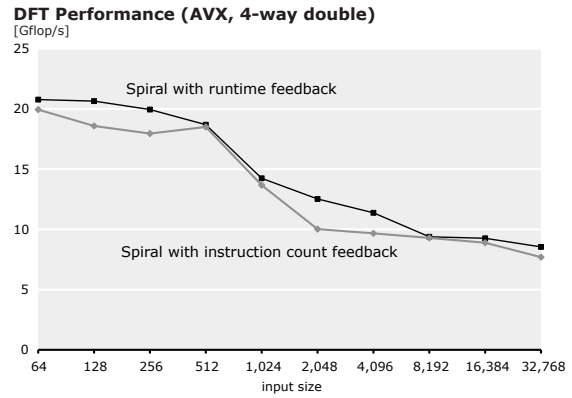

Figure 16: Comparison of 4-way AVX vectorized DFT implementations generated using different search metrics

single-precision SSE on Intel architectures. At the time of this writing there is no AVX or LRBni support.

Comparison to Intel's compiler vectorization. Intel's C/C++ compiler (icc) 12.0.1 supports AVX as a vectorization target. However, the auto-vectorizer does not utilize the AVX fused multiply-add instructions. We instruct the Intel C compiler to generate AVX assembly code from scalar (ANSI C with the necessary pragmas) and count the AVX arithmetic and reorder instructions, taking loop trips into account. No comparison for LRBni is possible. The Intel compiler performs well and achieves a vectorization efficiency of about 5.6 on 8-way single-precision AVX, thus achieving about 72% of our vectorization efficiency. Note that this high efficiency is in large parts due to Spiral's search, which in effect finds the code with the best structure for the compiler to succeed. Also the use of vectorization pragmas and buffer alignment declarations contributes.

Figures 13 and 14 include the performance of Spiral-generated scalar code, autovectorized by the compiler as described. For the 8-way and 4-way cases, Spiral vectorized DFTs were 30% and 40% faster, respectively, than icc compiled DFTs.

Vectorization efficiency as a performance guide. Spiral's conventional search for the best performing DFTs relies heavily on runtime performance feedback as a guiding metric. We have argued that in the absence of runtime performance feedback, vectorization efficiency can serve as substitute metric for guiding search. Figures 15 and 16 compare the performance of Spiral generated, AVX vectorized DFTs produced using two different metrics to guide search: runtime performance feedback and vectorization efficiency. In the 8-way case, the vectorization efficiency guided code approaches to within 7.4% on average of the performance of the code generated using runtime feedback. Similarly, in the 4-way case, the performance difference between the two generation methods is 8% on average. The relatively small performance disparity for smaller sizes is attributable to the delicate balance of arithmetic instructions and permutations required to handle functional unit latencies and port conflicts. For larger sizes, vectorization efficiency has difficulty achieving the right balance between loading precomputed constants and calculating the constants on the fly.

System	Million instruction sequences/sec
2.6 GHz Core i7	2.1
3.0 GHz Core 2 Quad	1.3
2.8 GHz Opteron 2200	0.8

Table 1: Search throughput on three x86-based CPUs

Permutation search results. Table 1 summarizes the throughput of our search mechanism on three different architectures. On average, finding a base permutation matrix for LRB required about two hours, roughly the equivalent of evaluating 14 billion instruction sequences of length six on the Core i7, which took about 2 hours. To put this figure in perspective, when expanded to μ-ops an instruction sequence of length six is roughly 16 μ-ops. An exhaustive search would need to evaluate more than 512^{16} different instruction sequences requiring about 10^{29} years on a Core i7. The shortest LRB instruction sequences discovered were for interleaving and deinterleaving two vectors of complex numbers, both of which require six instructions each: four shuffles and two swizzles. In contrast, both of these operations can be done in two instructions each on SSE. These sequence lengths compare favorably with the heuristic described above which estimated four shuffle instructions based on a fully general unary shuffle.

7. Conclusion

Near-term designs in the commodity architecture space show a clear trend towards more sophisticated SIMD vector instruction sets featuring ever wider vectors. Effectively vectorizing code for such architectures is a major challenge due to highly complex, non-intuitive and expensive vector reordering instructions. In this paper we presented a superoptimizer for data reorganization (permutations) that are important building blocks in many computations in linear algebra and signal processing. We show that—using enough resources—highly efficient automatic vectorization is possible for the the rather complex recently announced SIMD vector extensions: Our superoptimizer evaluated 14 billion instruction sequences in about 2 hours to find an efficient 6-instruction implementation of the core data reorganization. Using our optimizer we generated a library of building blocks required for implementing FFTs on AVX and Larrabee. We connected our optimizer to the program generation system Spiral and used it to generate efficient FFT implementations for AVX and Larrabee's LRBni vector instructions achieving a vectorization efficiency of up to 80% of the vector length across vector architectures.

8. Acknowledgments

This work was supported by a gift from Intel Corporation and by NSF through award 0702386. Daniel S. McFarlin was supported

by an NPSC and NDSEG graduate fellowship. We are indebted to Scott Buck, Randy Roost, Joshua Fryman and Mitchell Lum of Intel Corporation for granting early access to Larrabee and AVX and their technical advice and guidance.

9. References

[1] Saman Amarasinghe, Samuel Larsen, and Samuel Larsen. Exploiting superword level parallelism with multimedia instruction sets, 2000.

[2] Intel Advanced Vector Extensions programming reference, 2008. http://software.intel.com/en-us/avx/.

[3] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *SIGPLAN Not.*, 41(11):394–403, 2006.

[4] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. *SIGPLAN Not.*, 39(6):82–93, 2004.

[5] Randall J. Fisher, All J. Fisher, and Henry G. Dietz. Compiling for simd within a register. In *11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC98*, pages 290–304. Springer Verlag, Chapel Hill, 1998.

[6] F. Franchetti and M Püschel. Short vector code generation for the discrete Fourier transform. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, pages 58–67, 2003.

[7] F. Franchetti, Y. Voronenko, and M. Püschel. Loop merging for signal transforms. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 315–326, 2005.

[8] F. Franchetti, Y. Voronenko, and M. Püschel. A rewriting system for the vectorization of signal transforms. In *Proc. High Performance Computing for Computational Science (VECPAR)*, 2006.

[9] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, 2009.

[10] Franz Franchetti and Markus Püschel. SIMD vectorization of non-two-power sized FFTs. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 2, pages II–17, 2007.

[11] Franz Franchetti and Markus Püschel. Generating SIMD vectorized permutations. In *International Conference on Compiler Construction (CC)*, volume 4959 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2008.

[12] M. Frigo. A fast Fourier transform compiler. In *Proc. ACM PLDI*, pages 169–180, 1999.

[13] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, pages 1381–1384, 1998.

[14] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[15] The Gnu C compiler web site. gcc.gnu.org.

[16] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A simd optimization framework for retargetable compilers. *ACM Trans. Archit. Code Optim.*, 6(1):1–27, 2009.

[17] The Intel C compiler web site. software.intel.com/en-us/intel-compilers.

[18] Intel. Integrated performance primitives 5.3, User Guide.

[19] Intel. Math kernel library 10.0, Reference Manual.

[20] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing FFT algorithms on various architectures. *Circuits Systems Signal Processing*, 9:449–500, 1990.

[21] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[22] Stefan Kral, Franz Franchetti, Juergen Lorenz, Christoph W. Ueberhuber, and Peter Wurzinger. Fft compiler techniques. In *In Compiler Construction: 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004*, pages 217–231, 2004.

[23] Alexei Kudriavtsev and Peter Kogge. Generation of permutations for simd processors. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 147–156, New York, NY, USA, 2005. ACM.

[24] C++ Larrabee Prototype Library, 2009. http://software.intel.com/en-us/articles/prototype-primitives-guide.

[25] A first look at the Larrabee New Instructions (LRBni), 2009. http://www.ddj.com/hpc-high-performance-computing/216402188.

[26] Henry Massalin. Superoptimizer: a look at the smallest program. *SIGPLAN Not.*, 22(10):122–126, 1987.

[27] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. *SIGPLAN Not.*, 41(6):132–143, 2006.

[29] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232–275, 2005. Special issue on *Program Generation, Optimization, and Adaptation*.

[30] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for simd devices. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 118–131, New York, NY, USA, 2006. ACM.

[31] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, August 2008.

[32] Bao-Hong Shen, Shuiwang Ji, and Jieping Ye. Mining discrete patterns via binary matrix factorization. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 757–766, New York, NY, USA, 2009. ACM.

[33] V. Snasel, J. Platos, and P. Kromer. On genetic algorithms for

boolean matrix factorization. *Intelligent Systems Design and Applications, International Conference on*, 2:170–175, 2008.

[34] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28:363–400, 2000.

[35] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. SIAM, 1992.

[36] The IBM XL C compiler web site. www-01.ibm.com/software/awdtools/xlcpp.

[37] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers*. ACM, New York, NY, USA, 1991.