Generating SIMD Vectorized Permutations

Franz Franchetti and Markus Püschel*

Electrical and Computer Engineering,
Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213
{franzf,pueschel}@ece.cmu.edu
http://www.spiral.net

Abstract. This paper introduces a method to generate efficient vectorized implementations of small stride permutations using only vector load and vector shuffle instructions. These permutations are crucial for high-performance numerical kernels including the fast Fourier transform. Our generator takes as input only the specification of the target platform's SIMD vector ISA and the desired permutation. The basic idea underlying our generator is to model vector instructions as matrices and sequences of vector instructions as matrix formulas using the Kronecker product formalism. We design a rewriting system and a search mechanism that applies matrix identities to generate those matrix formulas that have vector structure and minimize a cost measure that we define. The formula is then translated into the actual vector program for the specified permutation. For three important classes of permutations, we show that our method yields a solution with the minimal number of vector shuffles. Inserting into a fast Fourier transform yields a significant speedup.

1 Introduction

Most current instruction set architectures (ISAs) or ISA extensions contain single instruction multiple data (SIMD) vector instructions. These instructions operate in parallel on subwords of a large vector register (typically 64-bit or 128-bit wide). Typically SIMD vector ISA extensions support 2-way–16-way vectors of floating-point or integer type. The most prominent example is Intel's SSE family.

From a software development point of view the most challenging difference across vector extensions is their widely varying capability in reorganizing data with in-register shuffle instructions. To obtain highest performance it is crucial to limit memory accesses to transfers of entire vectors and to perform any data reordering within the register file, ideally using the minimal number of shuffle instructions. Unfortunately, the optimal solution is difficult to find and depends on the target architecture.

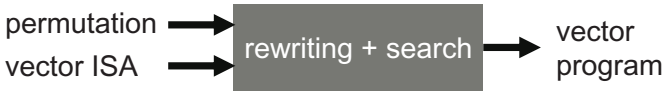
The above optimizations are most relevant for highly optimized numerical kernels where the slowdown suffered from "useless" instructions can be punishing. For instance, on a Core2 Duo loading an aligned unit-stride 16-way vector of

* This work was supported by NSF through awards 0234293, 0325687, and by DARPA through the Department of Interior grant NBCH1050009.

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 116–131, 2008.

8-bit elements costs one vector load, while gathering the same data at a stride costs at least 24 instructions, some of them particularly expensive. However, finding a short instruction sequence that reorganizes data in-register in a desired way is akin to solving puzzles.

Contribution. In this paper we automatically generate vector programs for an important class of permutations called stride permutations or matrix transpositions, given only the specification of the permutation and the specification of the target vector instruction set architecture (ISA).



The basic idea is that we model both instructions and permutations as matrices, and instruction sequences as matrix formulas using the Kronecker product formalism [1]. We design a rewriting system that applies matrix identities to generate, using a dynamic programming backtracking search, vectorized matrix formulas that minimize a cost measure that we define. The formula is then translated into the actual vector program implementing the specified permutation. For 3 important classes of permutations, we show that our method yields a solution with the minimal number of vector shuffles. We also demonstrate a significant speedup when inserting the generated permutation into small unrolled fast Fourier transform (FFT) kernels generated by Spiral [2].

Related Work. The motivation of this work arose from generating optimized programs for the discrete Fourier transform (DFT) in Spiral [2]. Spiral automates the entire code generation and optimization process, including vectorization [3,4], but, for FFTs, relies on three classes of in-register permutations [3]. These had to be implemented by hand for each vector extension. This paper closes the loop by generating these basic blocks for complete automatic porting across vector extensions. While our method is domain specific, it could in principle be applied to optimize vectorization of strided data access in a general purpose compiler, in an application similar to the approach in [5].

Reference [6] served as inspiration for our approach. Reference [7] derives the number of block transfers and [8] an optimal algorithm for transpositions on multi-level memories. Both index computation time and I/O time are considered in [9], and [10] optimizes matrix transpositions using a combination of analytical and empirical approaches. In contrast to prior work, we generate vectorized programs for small stride permutations that are optimized specifically for the peculiarities of current SIMD extensions.

General compiler vectorization techniques such as loop vectorization [11] or extraction of instruction-level parallelism and data reorganization optimization [12,5] operate on input programs while we generate programs for a very specific functionality using a declarative language and rewriting.

2 Background

We briefly overview SIMD vector instructions and introduce our mathematical framework to describe, manipulate, and vectorize stride permutations.

2.1 Vector SIMD Extensions

Most current general purpose and DSP architectures include short vector SIMD (single instruction, multiple data) extensions. For instance, Intel and AMD defined over the years MMX, Wireless MMX, SSE, SSE2, SSE3, SSSE, SSE4, SSE5, 3DNow!, Extended 3DNow!, and 3DNow! Professional as x86 SIMD vector extensions. On the PowerPC side AltiVec, VMX, the Cell SPU, and BlueGene/L's custom floating-point unit define vector extensions. Additional extensions are defined by PA-RISC, MIPS, Itanium, XScale, and many VLIW DSP processors.

Common to all vector extensions are stringent memory transfer restrictions. Only naturally aligned vectors can be loaded and stored with highest efficiency. Accessing unaligned or strided data can be extremely costly. For performance this requires that data be reordered (shuffled) inside the register file, using *vector shuffle instructions*. However, the available vector shuffle instructions vastly differ across SIMD extensions.

We mainly base our discussion on Intel's SSE2 extension, which defines six 128-bit *modes*: 4-way single-precision and 2-way double-precision vectors, and 2-way 64-bit, 4-way 32-bit, 8-way 16-bit, and 16-way 8-bit integer vectors. We denote the vector length of a SIMD extension mode with ν.

C intrinsic interface. Compilers extend the C language with vector data types and intrinsic functions for vector instructions. This way, the programmer can perform vectorization and instruction selection while register allocation and instruction scheduling are left to the C compiler. We use the C extension defined by the Intel C++ compiler (also supported by Microsoft Visual Studio and the GNU C compiler).

SSE2 vector shuffle instructions. Intel's SSE2 extension provides one of the richest sets of vector shuffle instructions among the currently available SIMD extensions. Table 1 summarizes the instructions native to SSE2's 6 modes.

For example, `_mm_shuffle_pd` is a parameterized binary shuffle instruction for the 2-way 64-bit floating-point mode. It shuffles the entries of its two operand vectors according to a compile time constant (a 2-bit integer). `_mm_unpacklo_ps` is a 4-way 32-bit floating-point shuffle instruction that interleaves the lower halves of its two operands.

We observe some intricacies of the SSE2 instruction set that considerably complicate its usability in general and the vectorization of permutations in particular: 1) The floating-point modes of SSE2 extend SSE while the integer modes extend MMX. This leads to inconsistencies among the available operations and naming conventions. 2) The parameterized shuffle instructions like `_mm_shuffle_ps` are not as general as in AltiVec. 3) Integer vector instructions for coarser granularity

Table 1. SSE2 shuffle instructions. {...} denotes a vector value. `a` and `b` are vectors, `a.i` and `b.i` vector elements. `<...>` denotes an integer compile time parameter derived from the constants inside `<>`. $0 \leq$ `j`, `k`, `m`, `n` $< 4 \leq$ `t`, `u`, `v`, `w` < 8, and $0 \leq$ `r`, `s` < 2.

2-way 64-bit floating-point
`_mm_unpacklo_pd(a, b)` → {a.0, b.0}
`_mm_unpackhi_pd(a, b})` → {a.1, b.1}
`_mm_shuffle_pd(a, b, <r, s>)` → {a.r, b.s}
4-way 32-bit floating-point
`_mm_unpacklo_ps(a, b)` → {a.0, b.0, a.1, b.1}
`_mm_unpackhi_ps(a, b)` → {a.2, b.2, a.3, b.3}
`_mm_shuffle_ps(a, b, <j, k, m, n>)`, → {a.j, a.k, b.m, b.n}
2-way 64-bit integer
`_mm_unpacklo_epi64(a, b)` → {a.0, b.0}
`_mm_unpackhi_epi64(a, b)` → {a.1, b.1}
4-way 32-bit integer
`_mm_unpacklo_epi32(a, b)` → {a.0, b.0, a.1, b.1}
`_mm_unpackhi_epi32(a, b)` → {a.2, b.2, a.3, b.3}
`_mm_shuffle_epi32(a, <j, k, m, n>)`→ {a.j, a.k, a.m, a.n}
8-way 16-bit integer
`_mm_unpacklo_epi16(a, b)` → {a.0, b.0, a.1, b.1, a.2, b.2, a.3, b.3}
`_mm_unpackhi_epi16(a, b)` → {a.4, b.4, a.5, b.5, a.6, b.6, a.7, b.7}
`_mm_shufflelo_epi16(a, <j,k,m,n>)` → {a.j,a.k,a.m,a.n,a.4,a.5,a.6,a.7}
`_mm_shufflehi_epi16(a, <t,u,v,w>)` → {a.0,a.1,a.2,a.3,a.t,a.u,a.v,a.w}
16-way 8-bit integer
`_mm_unpacklo_epi8(a, b)` → {a.0, b.0, a.1, b.1,...,a.7, b.7}
`_mm_unpackhi_epi8(a, b)` → {a.8, b.8, a.9, b.9,...,a.15, b.15}

(for instance, 4-way 32-bit) can be used with vectors of finer granularity (for instance, 8-way 16-bit and 16-way 8-bit).

Gather vs. vectorized permutation. Data permutations can be implemented in two ways:

- using vector shuffle instructions, or
- using gather/scatter operations that load/store ν scalars from/to non-contiguous memory locations.

The goal of this paper is to generate fast implementations of the former and evaluate against the latter. We focus on small permutations where no cache effects are visible. Vector shuffle operations increase the instruction count without doing "useful" computation and may be executed on the same execution units as vector arithmetic operations. However, once loaded, data stays in the register file provided no spilling is necessary. Conversely, implementing vector gathers and scatters can become costly: On SSE2, 2, 7, 16, and 24 instructions are needed per gather/scatter for 2-way, 4-way, 8-way, and 16-way vectors, respectively. On the Cell SPU it is even more costly, even though scalar loads are supported.

2.2 Mathematical Background

We introduce the mathematical formalism used in this paper. For more details, we refer the reader to [1,2,6]. All vectors in this paper are column vectors.

Direct sum of vectors. The direct sum $x \oplus y \in \mathbb{R}^{m+n}$ of two vectors $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$ is the concatenation of their elements.

Permutations and permutation matrices. The goal of this paper is to generate vectorized data permutations. We represent permutations as *permutation matrices* $P \in \mathbb{R}^{n \times n}$.

We define two basic permutation matrices: the $n \times n$ *identity matrix* I_n and the *stride permutation matrix* L^{mn}_m, which permutes an input vector x of length mn as $in + j \mapsto jm + i,\ 0 \le i < m,\ 0 \le j < n$. For example ("$\cdot$" represents "0"),

$$\mathrm{L}^6_2 \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_1 \\ x_3 \\ x_5 \end{pmatrix}, \quad \text{with } \mathrm{L}^6_2 = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{pmatrix}.$$

If x is viewed as an $n \times m$ matrix, stored in row-major order, then L^{mn}_m performs a transposition of this matrix.

Matrix operators. We need three matrix operators. The matrix *product* $C = AB$ is defined as usual. The *tensor* (or Kronecker) product of matrices is defined by

$$A \otimes B = (a_{i,j}B)_{i,j} \quad \text{with} \quad A = (a_{i,j})_{i,j}\,.$$

In particular,

$$\mathrm{I}_n \otimes A = \begin{pmatrix} A & & \\ & \ddots & \\ & & A \end{pmatrix}.$$

Finally, the *stacking* of two matrices A and B is defined in the obvious way:

$$C = \begin{pmatrix} A \\ B \end{pmatrix}.$$

Permutation matrix identities. Our approach uses factorization properties of stride permutation matrices. We summarize those that we use throughout this paper. Intuitively, these identities express performing a matrix transposition by two passes instead of using a single pass. They are including blocked matrix transposition.

Identity matrices can be split into tensor products of identity matrices if their sizes are composite numbers, $\mathrm{I}_{mn} = \mathrm{I}_m \otimes \mathrm{I}_n$. Further, we use four factorizations of stride permutations:

$$\mathrm{L}^{kmn}_n = \left(\mathrm{L}^{kn}_n \otimes \mathrm{I}_m\right)\left(\mathrm{I}_k \otimes \mathrm{L}^{mn}_n\right) \tag{1}$$

$$\mathrm{L}^{kmn}_n = \mathrm{L}^{kmn}_{kn}\,\mathrm{L}^{kmn}_{mn} \tag{2}$$

$$\mathrm{L}^{kmn}_{km} = \left(\mathrm{I}_k \otimes \mathrm{L}^{mn}_m\right)\left(\mathrm{L}^{kn}_k \otimes \mathrm{I}_m\right) \tag{3}$$

$$\mathrm{L}^{kmn}_{km} = \mathrm{L}^{kmn}_k\,\mathrm{L}^{kmn}_m\,. \tag{4}$$

Table 2. Translating matrix formulas into Matlab style code. x denotes the input and y the output vector. The subscript of A and B specifies the size of the matrix. `x[b:s:e]` denotes the subvector of x starting at `b`, ending at `e` and extracted at stride `s`.

Matrix formula	Code
$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1];)`
$y = (\mathrm{I}_m \otimes A_n)x$	`for (i=0;i<m;i++)` `  y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1]);`
$y = (A_m \otimes \mathrm{I}_n)x$	`for (i=0;i<m;i++)` `  y[i:n:i+m-1] = A(x[i:n:i+m-1]);`

$y = \mathrm{L}_m^{mn}\, x$	`for (i=0;i<m;i++)`
	`  for (j=0;j<n;j++)`
	`  y[i+m*j]=x[n*i+j];`

$y = \binom{A_{m\times k}}{B_{n\times k}} x$	`y[0:1:m-1]  = A(x[0:1:k-1]);` `y[m:1:m+n-1] = B(x[0:1:k-1]);`

Translating matrix expressions into programs. Matrix formulas, constructed using the above formalism, can be recursively translated into standard scalar programs by applying the translation rules in Table 2 [13].

3 Vector Programs and Matrix Expressions

In this section we explain how we model vector instructions as matrices, sequences of vector instructions as matrix expressions, and how these are translated into programs.

3.1 Modeling Vector Shuffle Instructions as Matrices

We consider only unary and binary vector shuffle instructions. (AltiVec's `vec_perm` is a three-operand instruction but the third operand is a parameter.) The basic idea is to view each such instruction, when applied to its input vector(s), as a matrix-vector product. The matrix becomes a declarative representation of the instruction. As an example, consider the instruction `_mm_unpacklo_ps`, which performs the operation (see Table 1).

```
_mm_unpacklo_ps(a, b) -> {a.0, b.0, a.1, b.1}
```

Setting $x_0 = (a_0, a_1, a_2, a_3)^T$ and $x_1 = (b_0, b_1, b_2, b_3)^T$, this shuffle becomes the the matrix-vector product

$$y = M^4_{\texttt{\_mm\_unpacklo\_ps}}(x_0 \oplus x_1), \quad \text{with } M^4_{\texttt{\_mm\_unpacklo\_ps}} = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \end{pmatrix}.$$

Hence, the instruction `_mm_unpacklo_ps` is represented by the matrix $M^4_{\texttt{\_mm\_unpacklo\_ps}}$. The subscript indicates the instruction and the superscript the vector length ν.

Unary and binary instructions. In general, for a ν-way mode, unary instructions are represented as $\nu \times \nu$ matrices and binary instructions as $\nu \times 2\nu$ matrices. Further, each binary instruction induces a unary instruction by setting both of its inputs to the same value. The exact form of these matrices follow directly from Table 1.

Polymorphic instructions. Some instructions can be used with multiple data types, which produces different associated matrices. For example, in 2-way 64-bit integer mode and 4-way 32-bit integer mode, `_mm_unpacklo_epi64` is respectively represented by

$$M^2_{\texttt{\_mm\_unpacklo\_epi64}} = \begin{pmatrix} 1 & \cdot \\ \cdot & 1 \end{pmatrix} \quad \text{and} \quad M^4_{\texttt{\_mm\_unpacklo\_epi64}} = \begin{pmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{pmatrix}.$$

`_mm_unpacklo_epi64` can also be used in 8-way 16-bit and 16-way 8-bit integer mode as well as in 2-way 64-bit and 4-way 32-bit floating-point mode.

Parameterized instructions. We treat parameterized instructions as one instruction instance per possible parameter value. For instance, `_mm_shuffle_ps` is parameterized by four 2-bit constants, leading to 256 instruction instances. We assume that all parameters are fixed at compile time, even if the instruction set does support variable parameters (as AltiVec's `vec_perm`).

Building matrices from ISA definition. Our system generates the matrices for the given instruction set automatically. To do this, we first collect the instruction description from ISA and compiler manuals and basically copy them verbatim into a database. Each instruction is represented by a record including the vector length ν, the *semantics* function that takes up to three lists (two input vectors and one parameter vector) and produces a list, and the *parameters* list that contains all possible parameter values. Unary instructions ignore the second input and unparameterized instructions ignore the third input. For instance, `_mm_shuffle_ps` is represented by

```
Intel_SSE2.4_x_float._mm_shuffle_ps := rec(
  v := 4,
  semantics := (x, y, p) -> [x[p[1]], x[p[2]], y[p[3]], y[p[4]]],
  parameters := Cartesian([[1..4],[1..4], [1..4], [1..4]])
);
```

The matrix generation is straightforward by "applying" the instruction to the canonical base vectors; the results are the columns of the desired matrix. More formally, if $e^\nu_i \in \mathbb{R}^\nu$ is the canonical basis vector with the "1" at the ith position and $0^\nu \in \mathbb{R}^\nu$ the zero vector, then the matrix M_p for an instruction with semantics function $s(\cdot, \cdot, \cdot)$ and parameter p is given by

$$M^{\nu}_{\texttt{instr},p} = \left(s(e^{\nu}_0, 0^{\nu}, p)|\ldots|s(e^{\nu}_{\nu-1}, 0^{\nu}, p)|s(0^{\nu}, e^{\nu}_0, p)|\ldots|s(e^{\nu}_{0^{\nu},\nu-1}, p)\right).$$

3.2 Translating Matrix Formulas into Vector Programs

In Table 2, we summarized how matrix formulas are recursively translated into scalar code. To obtain vector programs for formulas representing permutations, we expand this table with three cases: instruction matrices, vector permutations, and half-vector permutations. Then we define the class of all formulas that we translate into vector programs and define a cost measure for these formulas.

Instruction matrices. If a matrix, such as $M^4_{\texttt{\_mm\_unpacklo\_ps}}$, corresponds to a vector instruction it is translated into this instruction.

Vector permutations. If a permutation is of the form $P \otimes \mathrm{I}_\nu$, P a permutation matrix, it permutes blocks of data of size ν. Hence, we translate $P \otimes \mathrm{I}_\nu$ into vector code by first translating P into scalar code, and then replacing the scalar data type to the corresponding vector data type.

For example, $y = (\mathrm{L}^4_2 \otimes \mathrm{I}_4)\, x$ is implemented for 4-way 32-bit floating point SSE2 in two steps. First, $y = \mathrm{L}^4_2\ x$ is translated into the scalar program.

```
float x[2], y[2]; y[0] = x[0]; y[1] = x[2]; y[2] = x[1]; y[3] = x[3];
```

Then the scalar data type `float` is replaced by the vector data type `__m128` to get the final program

```
__m128 x[2], y[2]; y[0] = x[0]; y[1] = x[2]; y[2] = x[1]; y[3] = x[3];
```

Half-vector permutation. Permutations $P \otimes \mathrm{I}_{\nu/2}$ are implemented using the same instructions `i1` and `i2` that implement, if possible, $\mathrm{L}^4_2 \otimes \mathrm{I}_{\nu/2}$. Otherwise, $P \otimes \mathrm{I}_{\nu/2}$ cannot be implemented.

Vectorized matrix formulas. We define *vectorized matrix formulas* $\langle \mathrm{vmf} \rangle$ as matrix formulas that can be translated into vector programs as explained above. The exact form depends on the vector extension and mode used. Formally, in BNF

$$\begin{aligned}\langle \mathrm{vmf} \rangle ::= {} & \langle \mathrm{vmf} \rangle \langle \mathrm{vmf} \rangle \mid \mathrm{I}_m \otimes \langle \mathrm{vmf} \rangle \mid \begin{pmatrix} \langle \mathrm{vmf} \rangle \\ \langle \mathrm{vmf} \rangle \end{pmatrix} \mid \langle \mathrm{perm} \rangle \otimes \mathrm{I}_\nu \mid \\ & \langle \mathrm{perm} \rangle \otimes \mathrm{I}_{\nu/2} \text{ if } \mathrm{L}^4_2 \otimes \mathrm{I}_{\nu/2} \text{ possible} \mid M_{\texttt{instr}} \text{ with } \texttt{instr} \text{ in ISA} \\ \langle \mathrm{perm} \rangle ::= {} & \mathrm{L}^{mn}_m \mid \mathrm{I}_m \otimes \langle \mathrm{perm} \rangle \mid \langle \mathrm{perm} \rangle \otimes \mathrm{I}_m \mid \langle \mathrm{perm} \rangle \langle \mathrm{perm} \rangle \end{aligned}$$

Cost measure. We define a cost measure for vectorized matrix formulas recursively through (5)–(11). (6) assigns a constant cost $c_{\texttt{instr}}$ to each instruction `instr`. (7) states that permutations of vectors are for free, as they do not incur any vector shuffle instructions. (9)–(11) makes our cost measure additive with respect to matrix operators "$\cdot$", "$\begin{pmatrix}\cdot\\ \cdot\end{pmatrix}$", and"$\otimes$". The instructions `i1` and `i2` in (8) are the same that implement $\mathrm{L}^4_2 \otimes \mathrm{I}_{\nu/2}$.

$$\text{Cost}_{\text{ISA},\nu}(P) = \infty, \quad P \text{ not a } \langle \text{vmf} \rangle \tag{5}$$

$$\text{Cost}_{\text{ISA},\nu}(M^{\nu}_{\texttt{instr}}) = c_{\texttt{instr}} \tag{6}$$

$$\text{Cost}_{\text{ISA},\nu}(P \otimes \text{I}_{\nu}) = 0, \quad P \text{ permutation} \tag{7}$$

$$\text{Cost}_{\text{ISA},\nu}\left(P \otimes \text{I}_{\nu/2}\right) = \lfloor n/2 \rfloor c_{\texttt{i1}} + \lceil n/2 \rceil c_{\texttt{i2}}, \quad P \; 2n \times 2n \text{ permutation} \tag{8}$$

$$\text{Cost}_{\text{ISA},\nu}(A\,B) = \text{Cost}_{\text{ISA},\nu}(A) + \text{Cost}_{\text{ISA},\nu}(B) \tag{9}$$

$$\text{Cost}_{\text{ISA},\nu}\left(\begin{pmatrix} A \\ B \end{pmatrix}\right) = \text{Cost}_{\text{ISA},\nu}(A) + \text{Cost}_{\text{ISA},\nu}(B) \tag{10}$$

$$\text{Cost}_{\text{ISA},\nu}(\text{I}_m \otimes A) = m\text{Cost}_{\text{ISA},\nu}(A) \tag{11}$$

To minimize the instruction count $c_{\texttt{instr}} = 1$ is chosen. Using values of $c_{\texttt{instr}}$ that depend on `instr` allows for fine-tuning of the instruction selection process when multiple solutions with minimal instruction count exist. For example, for SSE2 we set $c_{\texttt{instr}} = 1$ for binary instructions and $c_{\texttt{instr}} = 0.9$ to unary instructions. This slightly favors unary instructions which require one register less. Other refinements are possible.

4 Generating Vectorized Permutation Programs

Our goal is to generate efficient vector programs that implement stride permutations $\text{L}^{n\nu}_{k}$. The parameters of the stride permutation imply that we permute data that can be stored in an array of n SIMD vectors and that $k \mid n\nu$.

Problem statement. *Input:* The permutation $\text{L}^{n\nu}_{k}$ to be implemented, the vector length ν, and a list of vector instruction instances `instr` for the ISA considered and their associated costs $c_{\texttt{instr}}$.

Output: A vectorized matrix formula for $\text{L}^{n\nu}_{k}$ with minimized cost and the implementation of the formula.

Our algorithm for solving the problem uses a rewriting system that is used in tandem with a dynamic programming search using backtracking. For important cases the solution is proven optimal.

4.1 Rewriting Rule Set

We use a rewriting system [14] to recursively translate the given stride permutation $\text{L}^{n\nu}_{k}$ into a vectorized matrix formula. In each rewriting step, the rewriting system finds one of the rules (12)–(22) and suitable parameters (a subset of k, ℓ, m, n, r, `instr`, `i1`, and `i2`) for that rule so that its left side matches a subformula in the current matrix formula. This matching subformula is then replaced by the right side of the rule.

Note that there may be degrees of freedom, as the matching of the left side may, for instance, involve factorizing the integer kmn into three integers k, m, and n, or involve the picking of a suitable, non-unique instruction `instr`. Also, it is not guaranteed that one of the rules is applicable, which may lead to dead ends. However, Section 4.3 shows that under relatively weak conditions (that are met by most current vector extension modes) there exists a solution for any $\text{L}^{n\nu}_{k}$.

The best solution has no obvious closed form; we use a dynamic programming search with backtracking to find it. The remainder of this section discusses the rewriting rule set while Section 4.2 discusses the search.

Recursive rules. Rules (12)–(17) are recursive rules. (12) is the entry rule, normalizing $\mathrm{L}^{n\nu}_k$ into the shape $\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r$ to simplify pattern matching. (13)–(16) mirror identities (1)–(4) and have the factorization kmn as degree of freedom. (17) extracts candidates for vector instruction matrices and may be followed by application of (20)–(22).

$$\mathrm{L}^{mn}_m \to \mathrm{I}_1 \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_1 \tag{12}$$

$$\mathrm{I}_\ell \otimes \mathrm{L}^{kmn}_n \otimes \mathrm{I}_r \to \left(\mathrm{I}_\ell \otimes \mathrm{L}^{kn}_n \otimes \mathrm{I}_{mr}\right)\left(\mathrm{I}_{\ell k} \otimes \mathrm{L}^{mn}_n \mathrm{I}_r\right) \tag{13}$$

$$\mathrm{I}_\ell \otimes \mathrm{L}^{kmn}_n \otimes \mathrm{I}_r \to \left(\mathrm{I}_\ell \otimes \mathrm{L}^{kmn}_{kn} \otimes \mathrm{I}_r\right)\left(\mathrm{I}_\ell \otimes \mathrm{L}^{kmn}_{mn} \otimes \mathrm{I}_r\right) \tag{14}$$

$$\mathrm{I}_\ell \otimes \mathrm{L}^{kmn}_{km} \otimes \mathrm{I}_r \to \left(\mathrm{I}_{k\ell} \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r\right)\left(\mathrm{I}_\ell \otimes \mathrm{L}^{kn}_k \otimes \mathrm{I}_m\right) \tag{15}$$

$$\mathrm{I}_\ell \otimes \mathrm{L}^{kmn}_{km} \otimes \mathrm{I}_r \to \left(\mathrm{I}_\ell \otimes \mathrm{L}^{kmn}_k \otimes \mathrm{I}_r\right)\left(\mathrm{I}_\ell \otimes \mathrm{L}^{kmn}_m \otimes \mathrm{I}_r\right) \tag{16}$$

$$\mathrm{I}_{k\ell} \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r \to \mathrm{I}_k \otimes \left(\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r\right) \quad \text{if } \ell mnr \in \{\nu, 2\nu\} \tag{17}$$

Base cases. Rules (18)–(22) translate constructs $\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r$ into vectorized matrix formulas. Rule (19) is only applied if $\mathrm{L}^4_2 \otimes \mathrm{I}_{\nu/2}$ can be done using two instructions.

$$\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_{r\nu} \to \left(\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r\right) \otimes \mathrm{I}_\nu \tag{18}$$

$$\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_{r\nu/2} \to \left(\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r\right) \otimes \mathrm{I}_{\nu/2} \tag{19}$$

$$\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r \to M^\nu_{\texttt{instr}} \quad \text{if } \exists\ \texttt{instr}\colon \mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r = M^\nu_{\texttt{instr}} \tag{20}$$

$$\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r \to M^\nu_{\texttt{i1}} M^\nu_{\texttt{i2}} \quad \text{if } \exists\ \texttt{i1}, \texttt{i2}\colon \mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r = M^\nu_{\texttt{i1}} M^\nu_{\texttt{i2}} \tag{21}$$

$$\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r \to \begin{pmatrix} M^\nu_{\texttt{i1}} \\ M^\nu_{\texttt{i2}} \end{pmatrix} \quad \text{if } \exists\ \texttt{i1}, \texttt{i2}\colon \mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r = \begin{pmatrix} M^\nu_{\texttt{i1}} \\ M^\nu_{\texttt{i2}} \end{pmatrix} \tag{22}$$

The right-hand side of (18) can be implemented solely using vector assignments (see Section 3.2). (19) introduces half-vector permutations which are necessary if mn is not a two-power. (20) matches if a (necessarily unary) vector instruction (instance) `instr` exists, which implements the left-hand side. As example,

```
y = _mm_shuffle_ps(x, x, _MM_SHUFFLE(0,2,1,3));
```

implements $y = (\mathrm{I}_1 \otimes \mathrm{L}^4_2 \otimes \mathrm{I}_1)\, x$ for 4-way single-precision floating-point SSE2. (21) matches if two (necessarily unary) vector instruction (instances) `i1` and `i2` exist, which implement its left-hand side when applied consecutively. As example,

```
y = _mm_shufflehi_epi16(_mm_shufflelo_epi16(x,
      _MM_SHUFFLE(0,2,1,3)), _MM_SHUFFLE(0,2,1,3));
```

implements $y = (\mathrm{I}_2 \otimes \mathrm{L}^4_2 \otimes \mathrm{I}_2)\, x$ for 16-way 8-bit integer SSE2. (22) matches if two (necessarily binary) vector instruction (instances) `i1` and `i2` exist, which implement its left-hand side when applied to the input in parallel. As example,

```
y[0] = _mm_unpacklo_epi64(x[0], x[1]);
y[1] = _mm_unpacklo_epi64(x[0], x[1]);
```

implements $y = (\mathrm{I}_1 \otimes \mathrm{L}^4_2 \otimes \mathrm{I}_4)\, x$ for 8-way 16-bit integer SSE2.

Base case library. To speed up the pattern matching required in (20)–(22), we perform a one-time initialization for each new vector architecture (instruction set and mode), and build a base case library that caches the instruction sequences that implement $\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r$ for all values ℓ, m, n, and r with $\ell mnr \in \{\nu, 2\nu\}$ or stores that no such instruction(s) exist. We build this table in a five-step procedure.

- First we create the matrices associated with each instance of each instruction (for all modes and parameters).
- Next we filter out all matrices that have more than one "1" per column, as these matrices cannot be used to build permutations.
- To support (19), we search for a pair of binary instructions that implement $\mathrm{L}^4_2 \otimes \mathrm{I}_{\nu/2}$.
- To support (20), we find all unary instruction (instances) that implement $\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r$ with $\nu = \ell mnr$ and save them in the cache.
- To support (21), we find all sequences of two unary instruction (instances) that implement $\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r$ with $\nu = \ell mnr$ and save them in the cache.
- To support (22), we find all pairs binary instruction (instances) that implement $\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r$ with $2\nu = \ell mnr$ and save them in the cache.

4.2 Dynamic Programming Search

The rule set (12)–(22) contains recursive and base rules with choices. We want to find a (not necessarily unique) vectorized matrix formula for $\mathrm{L}^{n\nu}_k$ with minimal cost. We use dynamic programming with backtracking, which finds the optimal solution within the space of possible solutions spanned by the rewriting rules.

Dynamic Programming (DP). For a formula F, let $\mathrm{E}(F)$ be the set of formulas that can be reached by applying one rewriting step using (12)–(22). Assume $A \in \mathrm{E}(F)$ is not yet a vectorized matrix formula. We define $\mathrm{X}(A)$ as the optimal vectorized matrix formula, computed recursively together with its cost, or cost $= \infty$ is it does not exist. DP computes $\mathrm{X}(F)$ as

$$\mathrm{X}(F) = \arg\min \left\{ \mathrm{Cost}_{\mathrm{ISA},\nu} \left(\mathrm{X}(A) \right) | A \in \mathrm{E}(F) \right\}. \tag{23}$$

All computed optimal costs and associated formulas are stored in a table. DP is started by evaluating $\mathrm{Cost}_{\mathrm{ISA},\nu} \left(\mathrm{X}(\mathrm{L}^{n\nu}_k) \right)$.

Backtracking. Not all formulas $\mathrm{I}_\ell \otimes \mathrm{L}^{mn}_m \otimes \mathrm{I}_r$ with $\ell mnr \in \{\nu, 2\nu\}$ can be necessarily translated into a vectorized matrix formula using (20)–(22). Thus, randomly picking elements $A \in \mathrm{E}(F)$ during the rewriting process may not yield a vectorized matrix formula at termination; hence, DP needs to backtrack and in the worst case will generates all formulas that can be obtained using our rewriting rules.

Existence and optimality of the solution are discussed in Section 4.3.

Cycles in rewriting. (14) and (16) produce an infinite cycle. To avoid that problem, we actually run two DPs—once without (14) and once without (16)—and take the minimum value of both answers.

Runtime of algorithm. The generation of vectorized base cases consists of two components: one-time generation of the base case library, and a DP for each stride permutation to be generated.

- *Base case library.* For an instruction set extension mode with n instruction instances, $O(n^2)$ matrix comparisons are required to build the base case library. On a current machine the actual runtime is a few seconds to minutes.
- *DP.* Let $n\nu = \prod_{i=0}^{k-1} p_i^{r_i}$ be the prime factorization of $n\nu$. For a stride permutation $\mathrm{L}_k^{n\nu}$, DP with backtracking is in exponential in $\sum_i r_i$. However, k and r_i are small as we are only handling basic blocks. On a current machine the actual runtime is a few seconds.

4.3 Existence and Optimality

Since we model both permutations and instructions using matrices, we can use mathematics to answer existence and optimality questions. Specifically, we give conditions under which our rewriting system finds a solution, i.e., a vectorized matrix formula, at all.

Further, we show vectorized matrix formulas for $\mathrm{L}_\nu^{\nu^2}$ generated for all modes of SSE2 and the Cell BE and establish their optimality. We also discuss the optimality of solutions for $\mathrm{L}_2^{2\nu}$, and $\mathrm{L}_\nu^{2\nu}$. These three permutations are the ones needed, for example, in the short-vector Cooley-Tukey FFT [4]; $\mathrm{L}_\nu^{\nu^2}$ is an ubiquitous algorithmic building block and crucial in matrix-matrix and matrix-vector multiplication for some vector extensions.

Existence of solution. Under the most general conditions, our algorithm does not necessarily return a solution. However, under relatively weak conditions imposed on the ISA, a solution can be guaranteed. The conditions are met by most current SIMD extensions. One notable exception is 16-way 8-bit integer in SSE2, for which the second condition does not hold.

- For $\nu \mid k \mid n$, a $\langle \mathrm{vmf} \rangle$ for $\mathrm{L}_2^{2\nu}$ must exist.
- For $\nu \nmid k$ or $k \nmid n$, $\langle \mathrm{vmf} \rangle$ for $\mathrm{L}_2^{2\nu}$, L_2^{ν}, and $\mathrm{L}_2^4 \otimes \mathrm{I}_{\nu/2}$ must exist.

The proof explicitly constructs a (suboptimal) vectorized formula using rules (1)–(4). We omit the details.

Optimality of generated implementations. Floyd [7] derived the exact number of block transfers required to transpose an arbitrary matrix on a two-level memory where the small memory can hold two blocks. We can apply his theorem to our situation by identifying binary vector instructions with the two-element memory in his formulation. The number of block transfer operations then yields a lower bound on the number of binary vector instructions required to perform

a stride permutation. Specifically, if $\mathcal{C}_\nu(P)$ is the minimal number of vector shuffle instructions required to perform P, then

$$\mathcal{C}_\nu(\mathrm{L}^{2\nu}_k) \geq 2, \quad \text{for } k \neq 1, 2\nu, \quad \text{and} \quad \mathcal{C}_\nu(\mathrm{L}^{\nu^2}_\nu) \geq \nu \log_2 \nu. \tag{24}$$

For example, for $\mathrm{L}^{\nu^2}_\nu$ our method generates the following vectorized matrix formulas. On SSE2 and on Cell the corresponding instructions counts match the lower bounds on (24) for all modes and are hence optimal.

$$\begin{aligned}
\mathrm{L}^{16}_4 &= \big(\mathrm{L}^8_4 \otimes \mathrm{I}_2\big)\big(\mathrm{I}_2 \otimes \mathrm{L}^8_4\big) \\
\mathrm{L}^{64}_8 &= \big(\mathrm{I}_4 \otimes (\mathrm{L}^4_2 \otimes \mathrm{I}_4)\big)\big(\mathrm{L}^8_4 \otimes \mathrm{I}_8\big)\big(\mathrm{I}_4 \otimes (\mathrm{L}^8_4 \otimes \mathrm{I}_2)\big)\big((\mathrm{I}_2 \otimes \mathrm{L}^4_2) \otimes \mathrm{I}_8\big)\big(\mathrm{I}_4 \otimes \mathrm{L}^{16}_8\big) \\
\mathrm{L}^{256}_{16} &= \big(\mathrm{I}_8 \otimes (\mathrm{L}^4_2 \otimes \mathrm{I}_8)\big)\big(\mathrm{L}^{16}_8 \otimes \mathrm{I}_{16}\big)\big(\mathrm{I}_8 \otimes (\mathrm{L}^8_4 \otimes \mathrm{I}_4)\big)\big((\mathrm{I}_4 \otimes \mathrm{L}^4_2) \otimes \mathrm{I}_{16}\big) \\
&\quad \big((\mathrm{I}_2 \otimes \mathrm{L}^4_2 \otimes \mathrm{I}_2) \otimes \mathrm{I}_{16}\big)\big(\mathrm{I}_8 \otimes (\mathrm{L}^{16}_8 \otimes \mathrm{I}_2)\big)\big((\mathrm{I}_4 \otimes \mathrm{L}^4_2) \otimes \mathrm{I}_{16}\big)\big(\mathrm{I}_8 \otimes \mathrm{L}^{32}_{16}\big)
\end{aligned}$$

The formula for L^{64}_8 yields the following implementation in 8-way 16-bit integer SSE2. All variables are of type `__m128i`.

```
t3 = _mm_unpacklo_epi16(X[0], X[1]); t4 = _mm_unpackhi_epi16(X[0], X[1]);
t7 = _mm_unpacklo_epi16(X[2], X[3]); t8 = _mm_unpackhi_epi16(X[2], X[3]);
t11 = _mm_unpacklo_epi16(X[4], X[5]);t12 = _mm_unpackhi_epi16(X[4], X[5]);
t15 = _mm_unpacklo_epi16(X[6], X[7]);t16 = _mm_unpackhi_epi16(X[6], X[7]);
t17 = _mm_unpacklo_epi32(t3, t7);    t18 = _mm_unpackhi_epi32(t3, t7);
t19 = _mm_unpacklo_epi32(t4, t8);    t20 = _mm_unpackhi_epi32(t4, t8);
t21 = _mm_unpacklo_epi32(t11, t15);  t22 = _mm_unpackhi_epi32(t11, t15);
t23 = _mm_unpacklo_epi32(t12, t16);  t24 = _mm_unpackhi_epi32(t12, t16);
Y[0] = _mm_unpacklo_epi64(t17, t21); Y[1] = _mm_unpackhi_epi64(t17, t21);
Y[2] = _mm_unpacklo_epi64(t18, t22); Y[3] = _mm_unpackhi_epi64(t18, t22);
Y[4] = _mm_unpacklo_epi64(t19, t23); Y[5] = _mm_unpackhi_epi64(t19, t23);
Y[6] = _mm_unpacklo_epi64(t20, t24); Y[7] = _mm_unpackhi_epi64(t20, t24);
```

Further, $\mathrm{L}^{2\nu}_\nu$ can be implemented optimally on all considered vector architectures using 2 binary vector instructions. However, $\mathrm{L}^{2\nu}_2$ *cannot* be implemented optimally on 8-way and 16-way SSE2 due to restrictions in the instruction set.

5 Experimental Results

We generated and evaluated vectorized permutations for a single core of a 2.66 GHz Intel Core2 Duo and one SPE of a 3.2 GHz IBM Cell BE processor. We used the Intel C++ compiler 9.1 for SSE and the GNU C compiler 4.1.1 for the Cell BE. The small sizes of the code generated by our approach makes it infeasible to compare our generated programs to any optimized matrix transposition library.

Implementation in Spiral. We implemented our approach as part of Spiral [2], a program generator that autonomously implements and optimizes DSP

transforms. In particular, Spiral generates high performance vectorized DFT implementations [4]. These implementations require vectorized basic blocks for stride permutations $L_\nu^{\nu^2}$, $L_2^{2\nu}$, and $L_\nu^{2\nu}$, which were hand-supplied in [4]. Using the approach presented in this paper, we automate this last manual step to enable automatic porting to new vector architectures.

Stand-alone stride permutation. In the first experiment, we generated implementations for $y = L_\nu^{\nu^2}\, x$ for SSE2 2-way, 4way, 8-way, and 16-way, as well as one 4-way Cell SPU. We compare our generated vectorized shuffle-based implementation against the one based on vector gathers (see Section 2.1). The shuffle-based implementations require ν vector loads, ν vector stores, and $\nu \log_2 \nu$ shuffle operations. The gather-based implementations require ν vector gathers and ν vector stores. We measured the cycles required for the data to get permuted from L1 cache to L1 cache, measuring many iterations to compensate for the timing overhead and to get a throughput measure.

Table 3 summarizes the results. In this setting, the shuffle-based implementation is much cheaper than the gather-based implementation. The reason is that sustained subword memory access is particularly costly on modern CPUs, which are optimized for wide words.

Table 3. Runtime and number of instructions needed for the stride permutations $y = L_\nu^{\nu^2}\, x$ when implemented using vector shuffles (generated by our method) or gather-based (the usual approach)

	Core2 SSE2				*Cell SPU*	
	$\nu = 2$	$\nu = 4$	$\nu = 8$	$\nu = 16$	$\nu = 2$	$\nu = 4$
vector shuffle						
shuffle instructions	2	8	24	64	2	8
move instructions	4	8	16	32	6	10
cycles	4	13	35	106	15	22
vector gather						
gather instructions	4	28	128	384	14	62
store instructions	2	4	8	16	2	4
cycles	15	60	94	407	32	112

Permutations within numerical kernels. In the second experiment we investigated the impact of our generated vectorized permutations versus vector gathers inside DFT kernels. For proper evaluation, we used Spiral-generated DFT kernels using the split complex format; these kernels are very fast (equal or better than FFTW 3.1.2 and Intel IPP 5.1) since they consist exclusively of vector arithmetic, vector memory access, and stride permutations $L_\nu^{\nu^2}$.

For $n = k\nu^2 \leq 128$, $1 \leq k \leq 8$, a split complex DFT_n requires between $\frac{3}{\nu} n \log_2 n$ and $\frac{8}{\nu} n \log_2 n$ vector arithmetic operations and k stride permutations

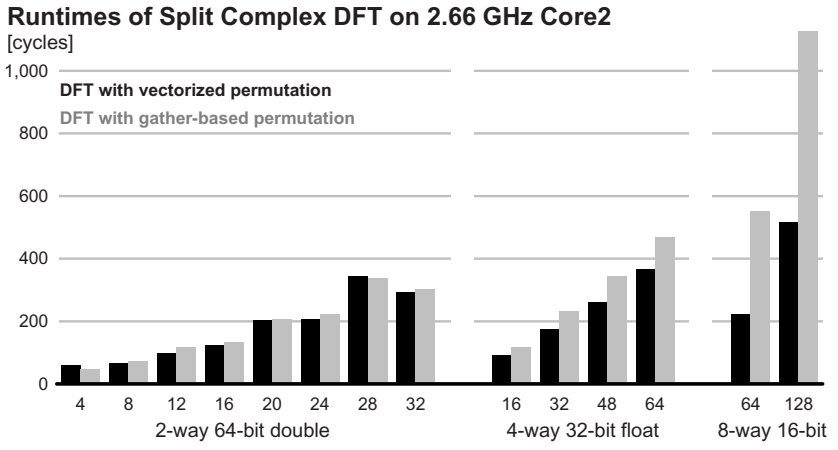


Fig. 1. Vectorized split complex DFT for various small sizes

$L_\nu^{\nu^2}$. Hence, the majority of vector instructions are arithmetic operations, but the number of vector shuffles and vector gathers still make up between 5% and 15% and between 15% to 50% of all instructions in their respective implementations. The overhead is largest for long vector lengths ν and small problem sizes n.

Figure 1 shows the cycle counts of Spiral-generated FFT code in both cases. For 2-way double-precision SSE2 the difference is negligible. For 4-way single-precision SSE2, the difference is up to 35%, due to a relative higher vector shuffle operations count and since expensive 4-way shuffle instructions are relatively more expensive. In the 8-way case these arguments become even more pronounced and the shuffle-based implementation is more than twice as fast as the gather-based implementation.

6 Conclusion

In this paper we show how to generate efficient vector programs for small stride permutations, which are important building blocks for numerical kernels. Even though this is a very specific class, we think we put forward an interesting approach that may have broader applicability. Namely, we have shown how to model vector instructions as matrices and then use matrix algebra for both generating and optimizing algorithm and implementation for the desired permutation and analyzing the quality of the result. On the practical side, our method enables us to quickly generate the building blocks that Spiral needs to generate FFTs for a given vector architecture. This enables us to port Spiral to new vector architectures without creative human effort.

References

1. van Loan, C.: Computational Framework of the Fast Fourier Transform. SIAM, Philadelphia (1992)
2. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE 93(2), 232–275 (2005); Special issue on Program Generation, Optimization, and Adaptation
3. Franchetti, F., Voronenko, Y., Püschel, M.: A rewriting system for the vectorization of signal transforms. In: Proc. High Performance Computing for Computational Science (VECPAR) (2006)
4. Franchetti, F., Püschel, M.: Short vector code generation for the discrete Fourier transform. In: Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS), pp. 58–67 (2003)
5. Nuzman, D., Rosen, I., Zaks, A.: Auto-vectorization of interleaved data for SIMD. In: Proc. Programming Language Design and Implementation (PLDI), pp. 132–143 (2006)
6. Johnson, J.R., Johnson, R.W., Rodriguez, D., Tolimieri, R.: A methodology for designing, modifying, and implementing FFT algorithms on various architectures. Circuits Systems Signal Processing 9, 449–500 (1990)
7. Floyd, R.W.: Permuting information in idealized two-level storage. Complexity of Computer Calculations, 105–109 (1972)
8. Vitter, J.S., Shriver, E.A.M.: Algorithms for parallel memory I: Two-level memories. Algorithmica 12(2/3), 110–147 (1994)
9. Suh, J., Prasanna, V.: An efficient algorithm for out-of-core matrix transposition. IEEE Transactions on Computers 51(6), 420–438 (2002)
10. Lu, Q., Krishnamoorthy, S., Sadayappan, P.: Combining analytical and empirical approaches in tuning matrix transposition. In: Proc. Parallel Architectures and Compilation Techniques (PACT), pp. 233–242 (2006)
11. Zima, H., Chapman, B.: Supercompilers for parallel and vector computers. ACM Press, New York (1990)
12. Ren, G., Wu, P., Padua, D.: Optimizing data permutations for SIMD devices. In: Proc. Programming Language Design and Implementation (PLDI), pp. 118–131 (2006)
13. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: A language and compiler for DSP algorithms. In: Proc. Programming Language Design and Implementation (PLDI), pp. 298–308 (2001)
14. Dershowitz, N., Plaisted, D.A.: Rewriting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, pp. 535–610. Elsevier, Amsterdam (2001)