Automatic Derivation and Implementation of Signal Processing Algorithms

Sebastian Egner
Philips Research Laboratories
Prof. Hostlaan 4, WY21
5656 AA Eindhoven, The Netherlands
sebastian.egner@philips.com

Jeremy Johnson
Mathematics and Computer Science
Drexel University
Philadelphia, PA 19104
jjohnson@mcs.drexel.edu

David Padua
Computer Science
University of Illinois
Urbana, IL 61801
padua@cs.uiuc.edu

Markus Püschel
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
pueschel@ece.cmu.edu

Jianxin Xiong*
Computer Science
University of Illinois
Urbana, IL 61801
jxiong@cs.uiuc.edu

Abstract

We present a computer algebra framework to automatically derive and implement algorithms for digital signal processing. The two main parts of the framework are AREP, a library for symbolic manipulation of group representations and structured matrices, and SPL, a compiler turning matrix expressions into efficient imperative-style numerical programs.

1 Introduction

Fast implementations of signal transforms are of crucial importance for real-time demands in signal processing. Many digital signal processing (DSP) transforms are mathematically given as a multiplication of a matrix M (the transform) with a vector x (the sampled signal). Examples are the discrete Fourier transform (DFT), trigonometric transforms, and the Hartley and Haar transforms, [17]. A fast algorithm for these transforms can be given by a factorization of M into a product of sparse matrices, [17, 21]. For example, the well-known Cooley/Tukey algorithm, [3], also known as fast Fourier transform (FFT), for computing the DFT can be written as

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes \mathbf{1}_n) \cdot T \cdot (\mathbf{1}_m \otimes \mathrm{DFT}_n) \cdot L,$$

where $\mathbf{1}_n$ denotes the identity matrix, T a diagonal matrix, and L a permutation matrix both depending on m and n.

After expressing an algorithm as a matrix factorization, the factorization must be converted into an efficient implementation, either in a high-level language such as C or FORTRAN, or perhaps directly in assembly or machine code. This is a fairly mechanical process, and, as we will see, can be automated.

In this paper we present a computer algebra framework to automatically derive and implement algorithms for DSP transforms. The two main parts of the framework are AREP, a library for symbolic manipulation of group representations and structured matrices, and SPL, a compiler turning matrix expressions into efficient imperative-style numerical programs.

The algebraic methods of AREP allow the user to interactively explore symmetries of discrete signal transformations and to use these symmetries to obtain fast algorithms. Any such algorithm can then be exported into the SPL language where it can be further manipulated and compiled into FORTRAN or C code. At the SPL level, alternative factorizations can be derived, and additional information needed for the production and optimization of the resulting programs can be inserted.

AREP and SPL are linked by a two-way interface which allows the entire process of deriving and implementing a fast algorithm to be automated, as is outlined in Figure 1. However, AREP may well produce suboptimal algorithms and must be seen as an exploration tool. SPL, on the other hand, aims at high performance code of production quality for DSP applications on various hardware and software platforms. Currently the code produced by the SPL compiler is of comparable efficiency to that of the FFTW package, [8]. The approach as sketched in Figure 1 is similar to the approach for creating VLSI implementations in [2], which indeed marks the origin of AREP. SPL is under development within the SPIRAL project, [12].

The paper is organized as follows. In Section 2 we present an algorithm for deriving a sparse matrix factorization for a certain class of matrices and introduce AREP which implements the procedure. Section 3 introduces the SPL language for expressing matrix factorizations, and the SPL compiler which translates them into efficient FORTRAN or C code. We conclude the paper with a complete example of implementing an 8-point discrete cosine transform (DCT) using AREP and SPL according to Figure 1.

2 AREP: Generating Matrix Factorizations

In this section we will explain how a fast algorithm for a DSP transform, given as a matrix M, can be derived automatically. The basic idea has its roots in [11] and has been

*The work of Jeremy Johnson, David Padua, Markus Püschel, and Jianxin Xiong was supported in part by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting. The work of Markus Püschel was supported by NSF through award 9988296.

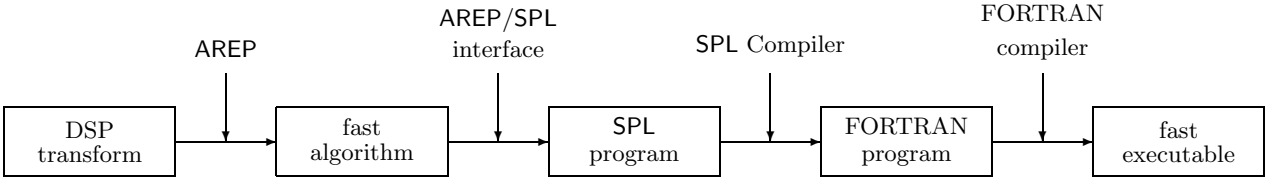

Figure 1: Automatic Generation of Algorithm and Implementation of a DSP transform

further developed in [4, 7, 5, 13, 15]. The approach essentially consists of two steps. In the first step, the "symmetry" of M is computed. The "symmetry" is a pair of group representations representing a certain invariance property of M (cf. Section 2.2). In the second step, the group representations are decomposed stepwise. This gives rise to factorized decomposition matrices and determines a factorization of M (cf. Section 2.3). The factorization represents a fast algorithm for computing the transform M. Intuitively speaking, the "symmetry" captures a large part of the redundancy contained in M, and the decomposition of the representations turns the redundancy into a fast algorithm.

For the convenience of the reader we will first introduce the mathematical notation we are going to use. In Section 2.2 we will define the term "symmetry" of a matrix, and in Section 2.3 we explain how to use it to derive a matrix factorization. In Section 2.4 we give a brief overview on the GAP share package AREP, [6], which contains an implementation of the factorization procedure. We conclude this section with some examples.

2.1 Mathematical Background

In this section we present the basic notation of representations and structured matrices we are going to use. For further information on representation theory we refer the reader to standard books such as [19].

Representations A (complex) *representation* of a group G is a homomorphism

$$\phi:\ G \to \mathrm{GL}_n(\mathbb{C})$$

of G into the group $\mathrm{GL}_n(\mathbb{C})$ of invertible $(n \times n)$-matrices over the complex numbers $\mathbb{C}$. The *degree* of ϕ is n. If $A \in \mathrm{GL}_n(\mathbb{C})$, then $\phi^A:\ g \mapsto A^{-1} \cdot \phi(g) \cdot A$ is the *conjugate* of ϕ by A. If ϕ and ψ are representations of G, then the representation $\phi \oplus \psi:\ g \mapsto \phi(g) \oplus \psi(g) = \left[\begin{smallmatrix} \phi(g) & 0 \\ 0 & \psi(g) \end{smallmatrix}\right]$ is called the *direct sum* of ϕ and ψ. The direct sum of n representations $\phi_1, \ldots, \phi_n$ is defined analogously. The representation ϕ is *irreducible*, if it cannot be conjugated to be a direct sum. Every representation ϕ (over $\mathbb{C}$) can be decomposed into a direct sum of irreducible representations by conjugation with a suitable matrix A (Maschke's Theorem). The matrix A is not uniquely determined and is called a *decomposition matrix* for ϕ. A representation ϕ is called a *permutation* representation, if all images $\phi(g)$ are permutation matrices, and ϕ is called a *monomial* representation, if all images $\phi(g)$ are monomial matrices. A monomial matrix has exactly one non-zero entry in every row and column and is hence a generalization of a permutation matrix.

Matrices We use the following notation to represent matrices. $[\sigma, n] = [\delta_{i\sigma j} \mid i, j = 1 \ldots n]$ is the $(n \times n)$-permutation matrix corresponding to the permutation σ. $\mathbf{1}_n$ denotes the identity matrix of size n, $\mathrm{diag}(L)$ denotes a diagonal matrix with the list L on the diagonal, $[\sigma, L]$ is the monomial matrix $[\sigma, \mathrm{length}(L)] \cdot \mathrm{diag}(L)$, $\oplus, \otimes$ denotes the direct sum and the Kronecker (or tensor) product of matrices, respectively, and $\mathrm{R}_\alpha = \left[\begin{smallmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{smallmatrix}\right]$ is the rotation matrix for angle α.

$$\mathrm{DFT}_n = [\omega_n^{k\ell} \mid k, \ell = 0 \ldots n-1],$$

where $\omega_n = e^{2\pi i/n}$, denotes the discrete Fourier transform of size n. The following example states the interpretation of the DFT in terms of representation theory.

Example 1 *It is a known fact that* DFT_n *maps the cyclic shift in the time-domain into a phase change in the frequency domain. In our notation,*

$$[(1, 2, \ldots, n), n] \cdot \mathrm{DFT}_n = \mathrm{DFT}_n \cdot \mathrm{diag}(1, \omega_n, \ldots, \omega_n^{n-1}).$$

In terms of representation theory, DFT_n *decomposes the permutation representation* $\phi: x \mapsto [(1, 2, \ldots, n), n]$ *of the cyclic group* $G = \mathsf{Z}_n = \langle x \mid x^n = 1 \rangle$ *into the direct sum* $\phi^{\mathrm{DFT}_n} = \rho_1 \oplus \ldots \oplus \rho_n$*, where the irreducible representations are given by* $\rho_k:\ x \mapsto \omega_n^{k-1}$*,* $k = 1 \ldots n$*.*

2.2 Symmetry of a Matrix

The notion of symmetry has a two-fold purpose. First, it catches the redundancy contained in the matrix M; second, it establishes the connection to representation theory, which enables the application of algebraic methods to factorize M, as sketched in Section 2.3.

We consider an arbitrary rectangular matrix $M \in \mathbb{C}^{m \times n}$. A *symmetry* of M is a pair (ϕ_1, ϕ_2) of representations of the same group G satisfying

$$\phi_1(g) \cdot M = M \cdot \phi_2(g), \quad \text{for all } g \in G.$$

We will use a shorthand notation and write $\phi_1 \xrightarrow{M} \phi_2$. We call G a *symmetry group* of M. With this general definition, however, every matrix has arbitrary many symmetries. If, for example, M is an invertible $(n \times n)$-matrix and ϕ is any representation of degree n of a group G, then M has the symmetry (ϕ, ϕ^M). Thus, in order to catch the redundancy contained in M, we will consider certain "types" of symmetry arising from restrictions on the representations ϕ_1, ϕ_2:

1. Mon-irred symmetry: ϕ_1 is monomial, ϕ_2 is a direct sum of irreducible representations.
2. Mon-mon symmetry: ϕ_1 and ϕ_2 are monomial.

In words, the matrix M has a mon-mon symmetry if there are non-trivial monomial matrices L, R such that $L \cdot M = M \cdot R$. Similarly, the matrix M has a mon-irred symmetry

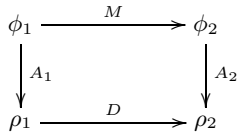

Figure 2: Factorization of M with symmetry (ϕ_1, ϕ_2)

if M is a decomposition matrix for a monomial representation ϕ. The rationale for considering the types of symmetry above will become clear in Section 2.3. Algorithms for finding symmetry are a main topic of [4, 5].

Example 2 *Example 1 shows that the* DFT_n *has the symmetry group* $G = \mathsf{Z}_n = \langle x \mid x^n = 1 \rangle$ *with symmetry* (ϕ_1, ϕ_2)*:*

$$\begin{aligned} \phi_1 &: \ x \mapsto [(1, 2, \ldots, n), n], \\ \phi_2 &: \ x \mapsto \mathrm{diag}(1, \omega_n, \ldots, \omega_n^{n-1}). \end{aligned}$$

Note, that (ϕ_1, ϕ_2) *is a mon-irred symmetry as well as a mon-mon symmetry.*

2.3 Matrix Factorization

Now we explain how to factorize a given matrix M, which has an arbitrary symmetry (ϕ_1, ϕ_2). First, the representations ϕ_1, ϕ_2 are decomposed with matrices A_1, A_2, respectively. This gives rise to two decomposed representations $\rho_1 = \phi_1^{A_1}, \rho_2 = \phi_2^{A_2}$. Second, the matrix $D = A_1^{-1} \cdot M \cdot A_2$ is computed to obtain the commutative diagram in Figure 2.

Altogether, we obtain the factorization

$$M = A_1 \cdot D \cdot A_2^{-1}. \tag{1}$$

¿From representation theory we know that D is a sparse matrix but the question of sparsity remains regarding the matrices A_1 and A_2. The factorization in (1) is useful only if the decomposition matrices A_1 and A_2 can themselves be determined as a product of sparse matrices. This is possible for monomial representations (with certain restrictions on the symmetry group G), as has been developed in [13, 14, 15], and justifies the consideration of the two types of symmetry described in Section 2.2:

1. Mon-irred symmetry: A_1 is a decomposition matrix of a monomial representation, A_2 is the identity, since ϕ_2 is already decomposed.
2. Mon-mon symmetry: A_1 and A_2 are decomposition matrices of monomial representations.

The factorizations of A_1 and A_2 arise from a stepwise decomposition of the corresponding representations along a chain of normal subgroups, [14], which will not be further explained here.

The algorithm for factoring a matrix with symmetry thus reads as follows.

Algorithm 1 *Given a matrix* M *to be factored into a product of sparse matrices.*

1. *Determine a suitable symmetry* (ϕ_1, ϕ_2) *of* M*.*
2. *Decompose* ϕ_1 *and* ϕ_2 *stepwise, and obtain (factorized) decomposition matrices* A_1, A_2*.*
3. *Compute the sparse matrix* $D = A_1^{-1} \cdot M \cdot A_2$*.*

Result: $M = A_1 \cdot D \cdot A_2^{-1}$ *is a factorization of* M *into a product of sparse matrices. This is a fast algorithm for evaluating the linear transformation* $x \mapsto M \cdot x$*.*

Algorithm 1 is implemented within the GAP share package AREP (see Section 2.4).

The following example applies Algorithm 1 to the DFT_4.

Example 3 *Let* $M = \mathrm{DFT}_4$*.* M *has the symmetry* $\phi_1 : x \mapsto [(1, 2, 3, 4), 4]$*,* $\phi_2 : \ x \mapsto \mathrm{diag}(1, \omega_4, \omega_4^2, \omega_4^3)$ *(cf. Example 2).* ϕ_2 *is already decomposed, hence* $A_2 = \mathbf{1}_4$*. Decomposing* ϕ_1 *stepwise yields the decomposition matrix*

$$\begin{aligned} &A_1 = \\ &(\mathrm{DFT}_2 \otimes \mathbf{1}_2) \cdot \mathrm{diag}(1, 1, 1, \omega_4) \cdot (\mathbf{1}_2 \otimes \mathrm{DFT}_2) \cdot [(2, 3), 4]. \end{aligned}$$

We compute $D = A_1^{-1} \cdot M \cdot A_2 = \mathbf{1}_4$ *and get the Cooley/Tukey factorization* $M = A_1$*.*

2.4 AREP

The implementation of Algorithm 1 is part of the package AREP, [6], which has been created as part of the thesis research in [4, 13]. AREP is implemented in the language GAP v3.4.4, [9], a computer algebra system for computational group theory, and has been accepted as a GAP share package. The goal of AREP was to create a package for computing with group representations up to equality, not only up to equivalence, as is done when using characters. In this sense, AREP provides the data types and the infrastructure to do efficient symbolic computation with representations and structured matrices which arise from the decomposition of representations. The central objects in this package are the recursive data types `ARep` and `AMat`.

An `ARep` is a GAP record representing a group representation. The record contains a number of fields which uniquely characterize a representation up to equality, e.g. degree, characteristic, and the represented group always have to be present. There are a number of elementary constructors for creating an `ARep`, e.g. by specifying the images on a set of generators of the group (`ARepByImages`). Furthermore, there are constructors building a structured `ARep` from given `AReps` (e.g. `DirectSumARep`, `InductionARep`). The idea is not to immediately evaluate such a construction, but to build an `ARep` representing it. For example, an `ARep` representing a direct sum has a field `summands` containing the list of summands. Conversion to an (unstructured) matrix representation is performed by calling the appropriate function. There are also functions converting an unstructured, e.g. monomial `ARep`, into a highly structured `ARep`, e.g. a conjugated induction of a representation of degree 1, which is mathematically *identical* to the original one. Permutation and monomial representations have been given special attention in the package since they are efficient to store and to compute with and they were the central object of interest.

The data type `AMat` has been created according to the same principle as `ARep`, as a GAP record representing a matrix. Again, there are elementary constructors to create an `AMat`, e.g. `AMatPerm` takes a permutation, a degree, and a characteristic and builds an `AMat` which efficiently represents a permutation matrix. Higher level constructors recursively build the product, direct sum, tensor product, etc., of `AMats` and are not evaluated unless an evaluation function is invoked. Since an `AMat` is not evaluated and the structure of

the corresponding matrix is maintained it provides an efficient method for storing and manipulating sparse-structured matrices. Matrix functions such as the determinant, trace, and inverse can be evaluated directly on an `AMat` using well-known mathematical rules, such as the determinant of the product of matrices is equal to the product of the determinants.

For a description of further capabilities of AREP we refer the reader to the AREP manual and web page, [6].

2.5 Examples

In this section we apply Algorithm 1 to a number of signal transforms. The following factorizations have been generated *entirely automatically* using AREP. Even the LaTeX-expressions displayed below have been generated. The definitions of the transforms follow [17]. The runtime in all cases was less than 24 seconds CPU time (substantially less time is required for all but the last example) on a Sun Ultra 5 with an UltraSPARC IIi CPU running at 440 MHz, 256 MB RAM, and running SunOS 5.7. The factorizations of the DCT and Hartley transform are due to different monirred symmetries with dihedral symmetry group. For more examples see the website of [6].

DFT: Cooley/Tukey Algorithm 1 finds the Cooley/Tukey factorization of DFT_n as shown in Example 3 for $n = 4$.

DFT: Rader The Rader FFT, [16], computes a DFT_p of prime size p using two DFTs of size $p-1$. The factorization algorithm finds this automatically.

$$
\begin{array}{l}
\mathrm{DFT}_5 = [(4,5),5]\cdot \\
(\mathbf{1}_1 \oplus ((\mathrm{DFT}_2 \otimes \mathbf{1}_2)\cdot \mathrm{diag}(1,1,1,\omega_4)\cdot(\mathbf{1}_2 \otimes \mathrm{DFT}_2)))\cdot \\
[(1,4)(2,5,3),(a,b,c,1,1)]\cdot(\mathbf{1}_3 \oplus \left[\begin{smallmatrix}1 & 4\\ 1 & -1\end{smallmatrix}\right])\cdot[(1,4)(2,3,5),5]\cdot \\
(\mathbf{1}_1 \oplus \tfrac{1}{4}\cdot(\mathbf{1}_2 \otimes \mathrm{DFT}_2)\cdot \mathrm{diag}(1,1,1,-\omega_4)\cdot(\mathrm{DFT}_2 \otimes \mathbf{1}_2))\cdot \\
[(3,4,5),5].
\end{array}
$$

The factorization is due to a mon-mon symmetry with cyclic symmetry group Z_4. Lines 2 and 4 essentially contain a DFT_4, a, b, c are constants (not given here due to lack of space).

DCT, type II The discrete cosine transform DCT (type II) [unscaled] is defined as the matrix

$$\mathrm{DCT}_n = \left[\cos\left(\frac{k(2\ell+1)\pi}{2n}\right) \mid k,\ell = 0\ldots n-1\right].$$

The scaled version of the DCT (type II) multiplies the (k,ℓ) element of the matrix by the scaling factor c_k where $c_k = 1/\sqrt{2}$ for $k = 0$ and $c_k = 1$ elsewhere. Below is the generated factorization for the unscaled DCT_8. A similar factorization can be obtained for the scaled variant.

$$
\begin{array}{l}
\mathrm{DCT}_8 = \\
[(2,5)(4,7)(6,8),8]\cdot(\mathbf{1}_2 \oplus \mathrm{R}_{\frac{3}{8}\pi} \oplus \mathrm{R}_{\frac{15}{16}\pi} \oplus \mathrm{R}_{\frac{21}{16}\pi})\cdot \\
[(2,4,7,3,8),(1,1,1,\sqrt{\tfrac{1}{2}},1,1,1,1)]\cdot \\
((\mathrm{DFT}_2 \otimes \mathbf{1}_3) \oplus \mathbf{1}_2)\cdot[(5,6),8]\cdot \\
(\mathbf{1}_4 \oplus \tfrac{1}{\sqrt{2}}\cdot \mathrm{DFT}_2 \oplus \mathbf{1}_2)\cdot[(2,3,4,5,8,6,7),8]\cdot \\
(\mathbf{1}_2 \otimes ((\mathrm{DFT}_2 \oplus \mathbf{1}_2)\cdot[(2,3),4]\cdot(\mathbf{1}_2 \otimes \mathrm{DFT}_2)))\cdot \\
[(1,8,6,2)(3,4,5,7),8].
\end{array}
$$

Hartley transform The discrete Hartley transform DHT_n is defined as the matrix

$$\mathrm{DHT}_n = \left[\cos\left(\tfrac{2k\ell\pi}{n}\right) + \sin\left(\tfrac{2k\ell\pi}{n}\right) \mid k,\ell = 0\ldots n-1\right].$$

The algorithm finds automatically the following factorization for DHT_8.

$$
\begin{array}{l}
\mathrm{DHT}_8 = [(1,8)(2,4)(3,6)(5,7),8]\cdot \\
(\mathbf{1}_2 \otimes ((\mathbf{1}_2 \otimes \mathrm{DFT}_2)\cdot[(2,3),4]\cdot(\mathrm{DFT}_2 \oplus \mathbf{1}_2)))\cdot \\
[(2,7,6,8,5,4,3),8]\cdot \\
(\mathbf{1}_4 \oplus -\tfrac{1}{\sqrt{2}}\cdot \mathrm{DFT}_2 \oplus \mathbf{1}_2)\cdot[(5,6),8]\cdot((\mathrm{DFT}_2 \otimes \mathbf{1}_3) \oplus \mathbf{1}_2)\cdot \\
[(2,5,3,6,4)(7,8),(1,-1,-\sqrt{2},-\sqrt{2},\sqrt{2},\sqrt{2},-1,-1)]\cdot \\
(\mathbf{1}_6 \oplus \mathrm{DFT}_2)\cdot[(2,5,8,7,3,4),8].
\end{array}
$$

Haar transform The Haar transform HT_{2^k} is defined recursively by:

$$\mathrm{HT}_2 = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix},\ \mathrm{HT}_{2^{k+1}} = \begin{bmatrix}\mathrm{HT}_{2^k} & \otimes\,[1\quad 1]\\ 2^{k/2}\cdot \mathbf{1}_{2^k} & \otimes\,[1\ -1]\end{bmatrix},$$

for $k \geq 1$. A fast algorithm for the Haar transform follows directly from the definition. For $k = 3$ we build the corresponding matrix and input it into the factorization algorithm.

$$
\begin{array}{l}
\mathrm{HT}_8 = [(1,8,6,4,2,7,5,3),8]\cdot \\
(\mathrm{diag}(-\sqrt{2},\sqrt{2}) \oplus \mathbf{1}_4 \oplus \mathrm{DFT}_2)\cdot[(1,5,4,8,6,3,7,2),8]\cdot \\
(\mathbf{1}_2 \otimes ([(1,2),4]\cdot(\mathrm{DFT}_2 \oplus 2\,\mathbf{1}_2)\cdot[(2,3),4]\cdot(\mathbf{1}_2 \otimes \mathrm{DFT}_2)))\cdot \\
[(1,8,4,7)(3,6,2,5),(1,1,1,1,-1,-1,-1,-1)].
\end{array}
$$

The factorization is based on a mon-irred symmetry. The symmetry group has been recognized as an iterated wreath product, [18].

3 SPL: Implementing Matrix Factorizations

In Section 2 we have seen that algorithms for signal transforms can be described by matrix factorizations and that it is possible in many situations to automatically derive such factorizations using the AREP library. A matrix factorization produced by AREP in the form of an `AMat`-object can be converted to an SPL program and compiled by the SPL compiler to produce an efficient numeric program corresponding to the matrix factorization. Moreover, the SPL programming environment includes tools for obtaining alternative matrix factorizations that may lead to improved performance of the resulting numeric code. Performance can be investigated using a GAP interface (other interfaces for MATLAB and Maple have also been developed) that makes it easy to obtain and analyze timings for alternative SPL expressions.

SPL is a domain-specific programming language for expressing and implementing matrix factorizations. It was originally developed to investigate and automate the implementation of FFT algorithms, [1] (there it was called TPL). As such, some of the built-in matrices are biased towards the expression of FFT algorithms, however, it is important to note that SPL is not restricted to the FFT. It contains features that allow the user to introduce notation suitable to any class of matrix expressions.

This section summarizes the SPL language and outlines the features and structure of the SPL compiler and programming environment. A more complete discussion of SPL and the SPL compiler can be found in [22].

3.1 SPL Language

An SPL program is essentially a sequence of mathematical formulas built up from a parameterized set of special

matrices and algebraic operators such as matrix composition, direct sum, and the tensor product. The SPL language uses a prefix notation similar to lisp to represent formulas. For example, the expressions `(compose A B)` and `(tensor A B)` correspond to the matrix product $A \cdot B$ and the tensor product $A \otimes B$ respectively. The language also includes special symbols such as `(F n)` and `(I m)` to represent the discrete Fourier transform matrix F_n and the identity matrix I_m (this notation differs from that used previously). In addition to the built-in symbols the user can assign an SPL formula to a symbol to be used in other expressions. Moreover, new parameterized symbols can be defined so that SPL programs can refer to other sets of parameterized matrices.

For example, the Cooley-Tukey factorization of F_4 from Example 3 is represented by the SPL expression

```
(compose
  (tensor (F 2) (I 2)) (T 4 2)
  (tensor (I 2) (F 2)) (L 4 2)
)
```

The parameterized symbols `(T 4 2)` and `(L 4 2)` correspond to a diagonal matrix called the twiddle factor matrix and a permutation matrix called a stride permutation respectively. In general, the expressions `(T m*n n)` and `(L m*n n)` correspond to matrices T^{mn}_n and L^{mn}_n defined by $T^{mn}_n(e^m_i \otimes e^n_j) = \omega^{ij}_{mn}(e^m_i \otimes e^n_j)$ and $L^{mn}_n(e^m_i \otimes e^n_j) = (e^n_j \otimes e^m_i)$, where e^n_i is the vector with a one in the i-th location and zeros elsewhere and ω_{mn} is a primitive mn-th root of unity.

The power of SPL for expressing alternative algorithms is illustrated by the following list of formulas corresponding to different variants of the FFT, [10, 20, 21]. Each of the formulas was obtained using the Cooley-Tukey factorization and elementary properties of the tensor product. The symbol R_8 refers to the eight-point bit-reversal permutation.

Apply Cooley/Tukey inductively

$$F_8 = (F_2 \otimes I_4)T^8_4(I_2 \otimes F_4)L^8_2$$

Recursive FFT

$$F_8 = (F_2 \otimes I_4)T^8_4(I_2 \otimes ((F_2 \otimes I_2)T^4_2(I_2 \otimes F_2)L^4_2))L^8_2$$

Iterative FFT (Cooley/Tukey)

$$F_8 = (F_2 \otimes I_4)T^8_4(I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T^4_2)(I_4 \otimes F_2)R_8$$

Vector FFT (Stockham)

$$F_8 = (F_2 \otimes I_4)T^8_4L^8_2(F_2 \otimes I_4)(T^4_2 \otimes I_2)(L^4_2 \otimes I_2)(F_2 \otimes I_4)$$

Vector FFT (Korn/Lambiotte)

$$F_8 = (F_2 \otimes I_4)T^8_4L^8_2(F_2 \otimes I_4)(T^4_2 \otimes I_2)L^8_2(F_2 \otimes I_4)L^8_2R_8$$

Parallel FFT (Pease)

$$F_8 = L^8_2(I_4 \otimes F_2)L^8_4T^8_4L^8_2L^8_2(I_4 \otimes F_2) \\ L^8_4(T^4_2 \otimes I_2)L^8_2L^8_2(I_4 \otimes F_2)R_8$$

These formulas are easily translated into SPL programs. The following SPL program corresponds to the formula for the iterative FFT on 8 points.

```
(define R8 (permutation (1 5 3 7 2 6 4 8)))
(compose
  (tensor (F 2) (I 4)) (T 8 4)
  (tensor (I 2) (F 2) (I 2)) (tensor (I 2) (T 4 2))
  (tensor (I 4) (F 2))
  R8
)
```

In general, an SPL program consists of the following constructs. Note that this list refers to SPL 3.28 (the current version) and will be extended in future versions (see http://polaris.cs.uiuc.edu/∼spl)

```
(1) matrix operations

    (tensor formula formula ...)
    (compose formula formula ...)
    (direct_sum formula formula ...)
    (conjugate formula permutation)
    (scale scalar formula)

(2) direct matrix description:

    (matrix (a11 a12 ...) (a21 a22 ...) ... )
    (sparse (i1 j1 a1) (i2 j2 a2) ... )
    (diagonal (d1 d2 ...))
    (permutation (p1 p2 ...))

(3) parameterized matrices:

    (I n)
    (F n)
    (T mn n)
    (L mn n)
    ...

(4) scalar expressions:

    +, -, *, /, %              # arithmetic operators
    pi                         # scalar constants
    exp(), cos(), sin(), ...   # scalar functions
    w(n,k)                     # exp(2*pi*i*k/n)
    w(n)                       # w(n,1)
    ...

(5) symbol definition:

    (define name formula)      # define a formula
    (define name const-expr)   # define a constant
    (primitive name ...)       # introduce new primitive
    (operator name ...)        # introduce new operator
    (template [condition] pattern (i-code-list))
```

In addition to these constructs, lines beginning with ";" are comments. Templates are used to define new parameterized matrices and operators. They also define the semantics of SPL programs and are used to control the generation of code by the SPL compiler. Templates are defined using an language independent syntax for code called i-code.

3.2 Formula Translation and the SPL Compiler

The SPL compiler consists of six stages: (1) parsing, (2) semantic binding, (3) type control, (4) optimization, (5) scheduling, and (6) code generation. The parser builds an abstract syntax tree (AST). The AST is converted to intermediate code (i-code) using templates to define the semantics of different SPL expressions. The i-code is expanded to produce type dependent code (e.g. double precision real or complex) and loops are unrolled depending on compiler parameters. After intermediate code is generated, various optimizations such as constant folding, copy propagation, common sub-expression elimination, and algebraic simplification are performed. Optionally, data dependency analysis

is used to rearrange the code to improve locality and pipelining. Finally, the intermediate code is converted to FORTRAN or C, leaving machine dependent compilation stages (in particular register allocation and instruction scheduling) to a standard compiler – the code generator could easily be modified assembly or machine code directly. Different options to the compiler, command line flags or compiler directives, control various aspects of the compiler, such as the data types, whether loops are unrolled, and whether the optional scheduler is used. Some compiler optimizations and instruction scheduling can also be obtained at the SPL level, by transforming the input formulas.

SPL formulas are compiled to code sequences by applying a compilation scheme to the formula and using recursion on the structure of the formula. For example, a composition $A \cdot B$ is compiled into the sequence $t = B \cdot x; y = A \cdot t$ mapping input signal x into output signal y with intermediate signal vector t. In the same way, the direct sum compiles into operations acting on parts of the input signal in parallel. The tensor product of code sequences for computing A and B can be obtained using the equation $A \otimes B = L^{mn}_m (I_n \otimes A) L^{mn}_n (I_m \otimes B)$.

For example, the code produced an SPL program corresponding to the 4-point Cooley/Tukey algorithm from Example 3 is

```
subroutine F4(y,x)

implicit complex*16(f), integer(r)
implicit automatic(f,r)
complex*16 y(4), x(4)

f0 = x(1) - x(3)
f1 = x(1) + x(3)
f2 = x(2) - x(4)
f3 = x(2) + x(4)
y(3) = f1 - f3
y(1) = f1 + f3
f6 = (0.00000000d0,-1.00000000d0) * f2
y(4) = f0 - f6
y(2) = f0 + f6
end
```

In this example, two compiler directives were added: one giving the name `F4` to the subroutine and one causing complex arithmetic to be used in the resulting code. Looking at this example, one already sees several optimizations that the compiler makes (e.g. multiplications by 1 and -1 are removed). More significantly, multiplication by the permutation matrix L^4_2 is performed as re-addressing in the array accesses. Another important point is that scalar variables were used for temporaries rather than an array. This has significant consequences on the FORTRAN compilers effectiveness at register allocation and instruction scheduling.

Changing the code type to real, `#codetype real`, breaks up complex numbers into real and imaginary parts which gives the chance for further optimizations. In the case above the (complex) multiplication vanishes.

```
subroutine F4(y,x)

implicit real*8(f), integer(r)
implicit automatic(f,r)
real*8 y(8), x(8)

f0 = x(1) - x(5)
f1 = x(2) - x(6)
f2 = x(1) + x(5)
f3 = x(2) + x(6)
f4 = x(3) - x(7)
f5 = x(4) - x(8)
f6 = x(3) + x(7)
f7 = x(4) + x(8)
y(5) = f2 - f6
y(6) = f3 - f7
y(1) = f2 + f6
y(2) = f3 + f7
y(7) = f0 - f5
y(8) = f1 + f4
y(3) = f0 + f5
y(4) = f1 - f4
end
```

In the previous example, we produced straight-line code. The SPL compiler is also capable of producing code with loops. For example, the formula $I_n \otimes A$ has a straightforward interpretation as a loop with n iterations, where each iteration applies A to a segment of the input vector. The SPL compiler is instructed to generate code using this interpretation by the following template in which `ANY` matches any integer and `any` matches any SPL expression. More details on the template mechanism may be found in [22].

```
(template (tensor (I ANY) any)
;; ---- Imm @ Bpq parameters: self(ny,nx), m,B(p,q)
;; ---- compute y = (I tensor B) x
;; $p1 and $p2 refer to the first and second parameters,
;; resp. The fields nx and ny refer to the row and
;; column dimension nx_1 = nx - 1 and ny_1 = ny - 1
 (
  do $p1
     $y(0:1:$p2.ny_1 $p2.ny) =
       call $p2( $x(0:1:$p2.nx_1 $p2.nx) )
  end
 ))
```

The following example shows how to use the SPL compiler to combine straight-line code with loops using formula manipulation and loop unrolling (loop unrolling is controlled by the compiler directive `#unroll`). Using a simple property of the tensor product, $I_{64} \otimes F_2 = I_{16} \otimes (I_2 \otimes F_2)$. This identity is used to control blocking in the code produced by SPL compiler.

```
; compute I_64 @ F_2 using I_16 @ (I_2 @ F_2)
#datatype real
#unroll on
(define I2F2 (tensor (I 2) (F 2)))
#unroll off
#subname I64F2
(tensor (I 16) I2F2)
```

The resulting code is

```
subroutine I64F2(y,x)

implicit real*8(f), integer(r)
implicit automatic(f,r)
real*8 y(64), x(64)

do i0 = 0, 15
   y(4*i0+2) = x(4*i0+1) - x(4*i0+2)
   y(4*i0+1) = x(4*i0+1) + x(4*i0+2)
   y(4*i0+4) = x(4*i0+3) - x(4*i0+4)
   y(4*i0+3) = x(4*i0+3) + x(4*i0+4)
end do
end
```

SPL clearly provides a convenient way of expressing and implementing matrix factorizations; however, it can only be considered as a serious programming tool, if the generated code is competitive with the best code available. One

strength of the SPL compiler is its ability to produce long sequences of straight-line code. In order to obtain maximal efficiency small signal transforms should be implemented with straight-line code thus avoiding the overhead of loop control or recursion. One of the fastest available packages for computing the FFT, FFTW, [8], utilizes this idea; however, the code generation facilities of FFTW are restricted to several FFT algorithms. SPL offers far greater generality yet the efficiency of the resulting code is comparable to FFTW (see [22] for timing comparisons).

3.3 The SPL Programming Environment

The SPL programming environment does not only supply a compiler but it also provides tools for generating SPL programs and interfacing with MATLAB and the computer algebra systems GAP, [9] and Maple. SPL programs can be manipulated using algebraic properties of the operators and symbols involved. Such properties are codified as rewrite rules. All of the FFT variants previously listed are easily generated using the Cooley-Tukey factorization and properties of the tensor product. Using the rewrite rules one can derive alternative SPL programs for computing the same signal transformation. Each formula can then be compiled, timed, and compared so that the one with the best performance is selected. Tools for generating and analyzing SPL programs can be found at [12].

In the next section we show an interface between SPL and AREP that allows us to generate SPL programs from GAP. In addition to producing SPL programs from GAP, we can read SPL programs into GAP and use the symbolic computation capabilities of GAP to verify the correctness of SPL programs. Finally, we have an interface between MATLAB and SPL which allows us to generate and time SPL programs from MATLAB. Using the interface we can generate many different SPL programs and analyze their performance (e.g. using MATLAB's plotting features).

4 Example: Automatic Implementation of an 8-point DCT

In this section we derive an algorithm and the corresponding implementation of an 8-point DCT according to the procedure sketched in Figure 1. Note that the DCT algorithm derived by AREP and the generated Fortran program require the same number of arithmetic operations as the best algorithms known (29 additions and 12 multiplications, e.g., [17]).

Step 1: Deriving a sparse matrix factorization Within GAP (using AREP) we create the matrix $M = \mathrm{DCT}_8$ and decompose it with an implementation of Algorithm 1. Each symmetry type, as presented in Section 2.2, has its own implementation of Algorithm 1. Here we use a "perm-irred" symmetry, a special case of a mon-irred symmetry. The output is an `AMat` (cf. Section 2) which represents the factorization already shown in Section 2.5. In order to factor the DCT using perm-irred symmetry it is necessary to first transpose the DCT and then take the transpose of the resulting AMat. The transpose of the DCT of type II is the DCT of type III. Thus the example calculation is performed using a DCT of type III and the resulting factorization is then transposed to obtain a factorization of the DCT of type II. Note that the representation of the DCT and its factorization is exact since the occurring cosines can be expressed as differences of cyclotomic numbers which are provided in GAP (`E(n)`$= \omega_n$).

```
gap> RequirePackage("arep");
gap> M :=  MatSPL(Transform("DCT3",8));
gap> A := MatrixDecompositionByPermIrredSymmetry(M);
gap> A := TransposedAMat(A);
AMatPerm((2,5)(4,7)(6,8), 8) *
DirectSumAMat(
  IdentityPermAMat(2),
  RotationAMat(3/8),
  RotationAMat(15/16),
  RotationAMat(21/16)
) *
AMatMon( Mon(
  (2,4,7,3,8),
  [ 1, 1, 1, 1/2*E(8)-1/2*E(8)^3, 1, 1, 1, 1 ]
) ) *
DirectSumAMat(
  TensorProductAMat(
    DFTAMat(2),
    IdentityPermAMat(3)
  ),
  IdentityPermAMat(2)
) *
AMatPerm((5,6), 8) *
DirectSumAMat(
  IdentityPermAMat(4),
  (1/2*E(8)-1/2*E(8)^3) * DFTAMat(2),
  IdentityPermAMat(2)
) *
AMatPerm((2,3,4,5,8,6,7), 8) *
TensorProductAMat(
  IdentityPermAMat(2),
  DirectSumAMat(
    DFTAMat(2),
    IdentityPermAMat(2)
  ) *
  AMatPerm((2,3), 4) *
  TensorProductAMat(
    IdentityPermAMat(2),
    DFTAMat(2)
  )
) *
AMatPerm((1,8,6,2)(3,4,5,7), 8)
```

This is the AMat representation of the factorization of DCT_8 presented in the AREP example section. The factorization can be verified by expanding the AMat (convert to a matrix) and comparing it to the defining matrix.

```
gap> MatAMat(A) = TransposedMat(M);
true
```

Step 2: Translating into a SPL program The `AMat` A is translated into a SPL program and exported into the file `dct.spl`. The translation is performed in two steps using an intermediate data structure (not explained here). In order to use the following commands, several the files containing the GAP programs for AREP-SPL interface must be loaded (these files may be obtained from [12]).

```
gap> S := SPLAMat(A);
gap> ExportSPL("dct.spl", S);
```

The file `dct.spl` contains the SPL program

```
( compose
  ( permutation ( 1 5 3 7 2 8 4 6 ) )
  ( direct_sum
    ( I 2 )
    ( R 3/8 )
```

```
    ( R 15/16 )
    ( R 21/16 )
  )
  ( permutation ( 1 4 8 7 5 6 3 2 ) )
  ( diagonal ( 1 1 1 sqrt(1/2) 1 1 1 1 ) )
  ( direct_sum
    ( tensor
      ( F 2 )
      ( I 3 )
    )
    ( I 2 )
  )
  ( permutation ( 1 2 3 4 6 5 7 8 ) )
  ( direct_sum
    ( I 4 )
    ( compose
      ( diagonal ( sqrt(1/2) sqrt(1/2) ) )
      ( F 2 )
    )
    ( I 2 )
  )
  ( permutation ( 1 3 4 5 8 7 2 6 ) )
  ( tensor
    ( I 2 )
    ( compose
      ( direct_sum
        ( F 2 )
        ( I 2 )
      )
      ( permutation ( 1 3 2 4 ) )
      ( tensor
        ( I 2 )
        ( F 2 )
      )
    )
  )
  ( permutation ( 8 1 4 5 7 2 3 6 ) )
)
```

Note that version 3.28 of the SPL compiler does not directly support the rotation matrices `R`. Therefore, it is necessary to first expand the rotations before attempting to compile the resulting SPL program. This can be done using the command `PrepareForExportingSPL`.

Step 3: Compiling to a FORTRAN program The SPL compiler translates the SPL program into a FORTRAN program performing different optimizations (see Section 3). The option "`-R`" creates a program for real input vectors.

```
spl -R dct.spl > dct.f
```

The file `dct.f` contains a FORTRAN 77 program:

```
      subroutine sub(y,x)

      implicit real*8(f), integer(r)
      implicit automatic(f,r)
      real*8 y(8), x(8)

      f0 = x(8) - x(1)
      f1 = x(8) + x(1)
      f2 = x(4) - x(5)
      f3 = x(4) + x(5)
      f4 = f1 - f3
      f5 = f1 + f3
      f6 = x(7) - x(2)
      f7 = x(7) + x(2)
      f8 = x(3) - x(6)
      f9 = x(3) + x(6)
      f10 = f7 - f9
      f11 = f7 + f9
      f12 = f8 - f6
      f13 = f8 + f6
      f14 = 0.7071067811865476d0 * f13
      f15 = 0.7071067811865476d0 * f12
      f16 = f5 - f11
      y(1) = f5 + f11
      f18 = f0 - f15
      f19 = f0 + f15
      f20 = f2 - f14
      f21 = f2 + f14
      y(5) = 0.7071067811865476d0 * f16
      f23 = f10 + f4
      f24 = (-0.5411961001461969d0) * f10
      f25 = 0.9238795325112867d0 * f23
      f26 = 1.3065629648763766d0 * f4
      y(3) = f24 + f25
      y(7) = f26 - f25
      f29 = f18 + f20
      f30 = (-1.175875602419359 1d0) * f18
      f31 = 0.1950903220161286d0 * f29
      f32 = (-0.7856949583871018d0) * f20
      y(2) = f30 + f31
      y(8) = f32 - f31
      f35 = f21 + f19
      f36 = 0.2758993792829431d0 * f21
      f37 = (-0.8314696123025452d0) * f35
      f38 = (-1.3870398453221475d0) * f19
      y(4) = f36 + f37
      y(6) = f38 - f37

      end
```

Similarly, a C program could have been generated using the command

```
spl -R -xlanguage=c dct.spl > dct.c
```

The correctness of the resulting FORTRAN or C programs can be verified by comparing it to the code generated for the defining matrix of the eight-point DCT. The command `CompareExternallySPL` compares the output of two SPL programs computed from a random input vector (it is also possible to test on a complete basis). The output of the comparison is `true` if the output vectors are within a specified numerical threshold. In addition, the norm (max norm by default) of the difference of the output vectors is returned.

```
Se := PrepareForExportingSPL(S);
Sd := TerminateSPL(Transform("DCT2",8));
gap> CompareExternallySPL(Se,Sd);
[ true, "8.881784e-16" ]
```

Step 4: Compiling to an executable A standard FORTRAN compiler produces executable machine dependent code, taking care of the special properties of the underlying hardware (such as number and type of registers etc).

```
f77 -c dct.f
```

The final result is a module to apply the DCT_8 to a real input signal of FORTRAN type `double precision`.

The performance of the compiled SPL program can be measured from within gap using the function `MeasureSPL`, which returns runtime in nanoseconds. The program obtained from the factored DCT matrix ran in 120 nanoseconds (on the Sun Ultra 5 described previously) compared to 286 nanoseconds for the program generated from the defining matrix.

```
gap> MeasureSPL(Se);
120
gap> MeasureSPL(Sd);
286
```

5 Future Work

AREP and SPL are both subjects of ongoing research, [12]. The main question arising from the factorizations shown in Section 2.5 is to understand the group theoretical properties of the transforms in terms of signal processing. The goal of SPL is to develop into a mature programming environment and library of highly efficient portable and adaptable DSP algorithms. Production of code for multiprocessor architectures and special purpose hardware will also be considered.

References

[1] Auslander, L., Johnson, J. R., and Johnson, R. W. Automatic implementation of FFT algorithms. Tech. Rep. 96-01, Dept. of Math. and Computer Science, Drexel University, Philadelphia, PA, June 1996. Presented at the DARPA ACMP PI meeting.

[2] Beth, T., Klappenecker, A., Minkwitz, T., and Nückel, A. *The ART behind IDEAS*, vol. 1000 of *LNCS*. Springer, 1995, pp. 141–158.

[3] Cooley, J. W., and Tukey, J. W. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. of Computation 19* (1965), 297–301.

[4] Egner, S. *Zur Algorithmischen Zerlegungstheorie Linearer Transformationen mit Symmetrie*. PhD thesis, Universität Karlsruhe, Informatik, 1997.

[5] Egner, S., and Püschel, M. Fast Discrete Signal Transforms and Monomial Representations of Finite Groups. Submitted for publication.

[6] Egner, S., and Püschel, M. *AREP – Constructive Representation Theory and Fast Signal Transforms*. `http://www.ece.cmu.edu/~smart/arep/arep.html`, 1998. GAP share package and manual.

[7] Egner, S., and Püschel, M. Automatic Generation of Fast Discrete Signal Transforms. *IEEE Trans. on Signal Processing* (2001). To appear September.

[8] Frigo, M., and Johnson, S. G. FFTW: An adaptive software architecture for the FFT. In *ICASSP '98* (1998), vol. 3, pp. 1381–1384. `http://www.fftw.org`.

[9] The GAP Team. *GAP – Groups, Algorithms, and Programming*. University St. Andrews, Scotland, 1997. `http://www-gap.dcs.st-and.ac.uk/~gap/`.

[10] Johnson, J., Johnson, R., Rodriguez, D., and Tolimieri, R. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures . *IEEE Trans. Circuits Sys. 9* (1990).

[11] Minkwitz, T. *Algorithms Explained by Symmetry*, vol. 900 of *LNCS*. Springer, 1995, pp. 157–167.

[12] Moura, J. M. F., Johnson, J., Johnson, R., Padua, D., Prasanna, V., Püschel, M., and Veloso, M. M. SPIRAL: Portable Library of Optimized SP Algorithms, 1998. `http://www.ece.cmu.edu/~spiral/`.

[13] Püschel, M. *Konstruktive Darstellungstheorie und Algorithmengenerierung*. PhD thesis, Universität Karlsruhe, Informatik, 1998. Translated in [14].

[14] Püschel, M. Constructive Representation Theory and Fast Signal Transforms. Tech. Rep. Drexel-MCS-1999-1, Drexel Univ., Philadelphia, 1999. Translation of [13].

[15] Püschel, M. Decomposing Monomial Representations of Solvable Groups. Tech. Rep. Drexel-MCS-1999-2, Drexel Univ., Philadelphia, 1999. Submitted for publication.

[16] Rader, C. M. Discrete Fourier Transforms When the Number of Data Samples is Prime. *Proceedings of the IEEE 56* (1968), 1107–1108.

[17] Rao, K. R., and Yip, P. *Discrete Cosine Transform*. Academic Press, 1990.

[18] Rockmore, D. A Wreath Product Approach to Signal Processing, 1999. Talk at IMACS-ACA.

[19] Serre, J. *Linear Representations of Finite Groups*. Springer, 1977.

[20] Tolimieri, R., An, M., and Lu, C. *Algorithms for Discrete Fourier Transforms and Convolution*, 2nd ed. Springer, 1997.

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

[22] Xiong, J., Johnson, J., Johnson, R., and Padua, D. SPL: A Language and Compiler for DSP Algorithms. In *Proc. PLDI* (2001).