Automatic derivation and implementation of fast convolution algorithms

Jeremy R. Johnson*, Anthony F. Breitzman

Department of Computer Science, Drexel University, Philadelphia, PA 19104, USA

Received 11 August 2000; accepted 11 June 2002

Abstract

This paper surveys algorithms for computing linear and cyclic convolution. Algorithms are presented in a uniform mathematical notation that allows automatic derivation, optimization, and implementation. Using the tensor product and Chinese remainder theorem, a space of algorithms is defined and the task of finding the best algorithm is turned into an optimization problem over this space of algorithms. This formulation led to the discovery of new algorithms with reduced operation count. Symbolic tools are presented for deriving and implementing algorithms.

Keywords: Cyclic convolution; Convolution algorithms

1. Introduction

Convolution is arguably one of the most important computations in signal processing. It also has applications outside of signal processing including the efficient computation of prime length Fourier transforms, polynomial multiplication, and large integer multiplication. Efficient implementations of convolution algorithms are therefore always in demand.

The careful study of convolution algorithms began with Winograd’s investigation of the complexity of convolution and related problems. Winograd (1977, 1980) proved a lower bound on the number of multiplications required for convolution, and used the Chinese remainder theorem (CRT) to construct optimal algorithms that achieve the minimum number of multiplies. Unfortunately, to reach the theoretical minimum in multiplications often requires an inordinate number of additions that may defeat any advantages gained by decreasing the number of multiplies. These results spurred further study in the design and

* Corresponding author. Tel.: +1-215-895-2893; fax: +1-215-895-0545. E-mail address: jjohnson@cs.drexel.edu (J.R. Johnson).

0747-7171/S - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2002.06.001
implementation of “fast” convolution algorithms. The research on this problem over the last 25 years is summarized in the books by Nussbaumer (1982), Burrus and Parks (1985), Blahut (1992) and Tolimieri et al. (1997).

Much of the research has focused on techniques for reducing the number of additions by using near-optimal rather than optimal multiplication counts. Other authors, beginning with Agarwal and Cooley (1977), have focused on using Winograd’s techniques to implement convolution algorithms for certain small sizes. These “small algorithms” are then combined to compute larger convolutions using various “prime factor” algorithms. This approach has had the greatest success in the application to computing prime size discrete Fourier transforms (DFTs) via Rader’s (1968) theorem and prime factor fast Fourier transforms (FFTs) (Burrus and Eschenbacher, 1981; Temperton, 1985).

In spite of all this work there remain many questions concerning these algorithms. The main unanswered question involves the determination of the practicality of the Winograd-based algorithms (a Winograd-based algorithm is understood to mean an algorithm constructed using the techniques introduced by Winograd (1977, 1980)) over the full range of input sizes. The problem of optimal design of convolution algorithms, given the various available techniques, remains an open question. One reason for this is the difficulty of implementing Winograd-based algorithms for general sizes. Another difficulty arises in trying to systematically compare “all” possible combinations of convolution algorithms and techniques. Such a systematic investigation is necessary in order to obtain an optimal implementation.

The main goal of this paper is to create a foundation for this investigation. In order to carry out this study, an infrastructure was developed for automatically deriving and implementing convolution algorithms. Previous work has been done in this direction; however most of these efforts have produced unoptimized straight-line code (Agarwal and Cooley, 1977; Cooley, 1989). More recent work by Selesnick and Burrus (1996), has automated the generation of convolution algorithms without using straight-line code. Their work highlighted the structure in prime-power Winograd algorithms and showed how to utilize this structure to generate structured code. However the code produced was for Matlab and does not produce an optimized implementation. Moreover, they do not provide tools to experiment with algorithmic choices nor arbitrary sizes. These limitations do not allow previous work to be used to systematically answer the performance questions discussed in this paper.

In this paper a research project is discussed that builds mainly on the work of Agarwal and Cooley (1977) and Selesnick and Burrus (1996), and aims to address the above issues. The main contributions are a uniform mathematical description of a large family of convolution algorithms, a software infrastructure based on this mathematical foundation, for generating, optimizing, and implementing convolution algorithms, and the discovery of new algorithms with reduced operation counts. This effort also builds upon techniques for automating the implementation of FFT algorithms developed by Johnson et al. (1990) and Auslander et al. (1996) and is part of the SPIRAL project (Moura et al., 1998) whose aim is to automate the design, optimization and implementation of signal processing algorithms.

This paper is organized as follows. Section 2 provides the mathematical tools needed to derive and classify convolution algorithms, and Section 3 describes a family of convolution algorithms from the literature using a consistent and precise algebraic formulation. This
provides a means to easily combine algorithms in ways not previously considered. Using this formulation, a space of algorithms is defined and the task of finding the best algorithm is turned into an optimization problem over this space of algorithms. Section 4 outlines the optimization problem and presents some of the improved algorithms that were found. Finally, in Section 5 a suite of tools is presented for implementing and experimenting with the Winograd-based convolution algorithms. An overview of the design and implementation along with several illustrative examples is provided.

2. Mathematical preliminaries

This section reviews the mathematical tools that are used in deriving convolution algorithms.

2.1. Three perspectives on convolution

Convolution can be viewed from three different perspectives: as a sum, a polynomial product, and a matrix operation. This allows polynomial algebra to be used to derive algorithms and the corresponding matrix algebra to be used to manipulate and implement algorithms.

The linear convolution (denoted \( u \ast v \)) of the vectors \( u = (u_0, \ldots, u_{M-1}) \) and \( v = (v_0, \ldots, v_{N-1}) \) is a vector of size \( M + N - 1 \). If both vectors are of the same size, \( M = N \), the linear convolution is said to be of size \( N \).

**Definition 2.1 (Linear Convolution).** Let \( u = (u_0, \ldots, u_{M-1}) \) and \( v = (v_0, \ldots, v_{N-1}) \). The \( i \)th component of \( u \ast v \) is equal to

\[
(u \ast v)_i = \sum_{k=0}^{i} u_{i-k} v_k, \quad 0 \leq i < 2N.
\]

If the vectors \( u = (u_0, \ldots, u_{M-1}) \) and \( v = (v_0, \ldots, v_{N-1}) \) are mapped to the polynomials

\[
u(x) = \sum_{i=0}^{M-1} u_i x^i \quad \text{and} \quad v(x) = \sum_{j=0}^{N-1} v_j x^j,
\]

then \( u \ast v \) is mapped to the polynomial \( u(x)v(x) \).

Linear convolution sum is also equivalent to the following matrix vector multiplication.

\[
u = \begin{bmatrix}
u_0 \\
u_1 & u_0 \\
\vdots & \vdots & \ddots \\
u_{M-1} & \vdots & \ddots & u_0 \\
& \ddots & \vdots & u_1 \\
& & \ddots & \vdots & u_{M-1} \\
& & & \ddots & \vdots \\
& & & & u_{M-1}
\end{bmatrix}v.
\]

(2)
Cyclic convolution of two vectors of size $N$ results in a new vector of length $N$, and is obtained from linear convolution by reducing the indices $i-k$ and $k$ in (1) modulo $N$.

**Definition 2.2** (Cyclic Convolution). Let $u = (u_0, \ldots, u_{N-1})$ and $v = (v_0, \ldots, v_{N-1})$. The $i$th component of the cyclic convolution of $u$ and $v$, denoted by $u \odot v$, is equal to

$$
(u \odot v)_i = \sum_{k=0}^{N-1} u_{(i-k) \mod N} v_k, \quad 0 \leq i < N.
$$

Cyclic convolution is obtained by multiplying the polynomials corresponding to $u$ and $v$ and taking the remainder modulo $x^N - 1$. It can also be recast in terms of matrix algebra, as the product of a circulant matrix $\text{Circ}_N(u)$, times the vector $v$.

$$
u_0 \ u_{N-1} \ u_{N-2} \ \ldots \ u_1
\begin{bmatrix}
\vdots
\end{bmatrix}
\begin{bmatrix}
u_0 \ u_{N-1} \ u_{N-2} \ \ldots \ u_1
\end{bmatrix} = v.
$$

This matrix is called a circulant matrix because the columns of the matrix are all obtained by cyclically rotating the first column.

A circulant matrix is generated by the shift matrix

$$S_N = \begin{bmatrix}
0 & \ldots & 0 & 0 & 1 \\
1 & 0 & \ldots & 0 & 0 \\
0 & 1 & 0 & \ldots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & \ldots & 0 & 1 & 0
\end{bmatrix},
$$

which is so named because when it is applied to a vector it cyclically shifts the elements. It is easy to verify that

$$
\text{Circ}_N(u) = \sum_{i=0}^{N-1} u_i S_N^i.
$$

### 2.2. Polynomial algebra

Elementary properties of polynomial algebras, in particular the CRT, can be used to derive convolution algorithms, and the regular representation can be used to convert from the polynomial view of convolution to the matrix view. Let $f(x)$ be a polynomial with coefficients in a field $F$, and let $F[x]/f(x)$ denote the quotient algebra of polynomials modulo $f(x)$. Typically $F$ will be the complex, $\mathbb{C}$, or real, $\mathbb{R}$, numbers depending on the convolution inputs; however, when deriving algorithms using the CRT the rationals, $\mathbb{Q}$ or an extension of the rationals will be used depending on the required factorization of $f(x)$. Linear convolution corresponds to multiplication in the polynomial algebra $F[x]$, and cyclic convolution corresponds to multiplication in $F[x]/x^N - 1$. 
The regular representation, $\rho$, of the algebra $\mathbb{F}[x]/f(x)$ is the mapping from $\mathbb{F}[x]/f(x)$ into the algebra of linear transformations of $\mathbb{F}[x]/f(x)$ defined by

$$\rho(A(x))B(x) = A(x)B(x) \pmod{f(x)},$$

where $A(x)$ and $B(x)$ are elements of $\mathbb{F}[x]/f(x)$. Once a basis for $\mathbb{F}[x]/f(x)$ is selected, the regular representation associates matrices with polynomials. Assume that $\deg(f(x)) = N$. The dimension of $\mathbb{F}[x]/f(x)$ is $N$, and $\{1, x, x^2, \ldots, x^{N-1}\}$ is a basis for $\mathbb{F}[x]/f(x)$.

With respect to this basis, $\rho(x) = C_f$, the companion matrix of $f(x)$, and the regular representation of $\mathbb{F}[x]/f(x)$ is the matrix algebra generated by $C_f$. In particular, when $f(x) = x^N - 1$, $\rho(x)$ is $S_N$ and the regular representation of $\mathbb{C}[x]/(x^n - 1)$ is the algebra of circulant matrices.

2.2.1. Chinese remainder theorem

The polynomial version of the Chinese remainder provides a decomposition of a polynomial algebra, $\mathbb{F}[x]/f(x)$ into a direct product of polynomial algebras.

**Theorem 2.1 (CRT).** Assume that $f(x) = f_1(x) \cdots f_t(x)$ in $\mathbb{F}$ where $\gcd(f_i(x), f_j(x)) = 1$ for $i \neq j$. Then

$$\mathbb{F}[x]/f(x) \cong \mathbb{F}[x]/f_1(x) \times \cdots \times \mathbb{F}[x]/f_t(x).$$

Moreover, the isomorphism is given constructively by a system of orthogonal idempotents $e_1(x), \ldots, e_t(x)$ where $e_i(x)e_j(x) \equiv 0 \pmod{f_i(x)}$ when $i \neq j$, $e_i(x)e_i(x) \equiv 1 \pmod{f_i(x)}$, and $e_1(x) + \cdots + e_t(x) \equiv 1 \pmod{f_i(x)}$. If

$$A(x) = A_1(x)e_1(x) + \cdots + A_t(x)e_t(x),$$

then

$$A(x) \equiv A_i(x) \pmod{f_i(x)}.$$

A more general version of this theorem with a proof can be found in Lang (1984), and Lauer (1982) shows how to compute the idempotents using the extended Euclidean algorithm.

The CRT implies that the regular representation of $\mathbb{F}[x]/f(x)$ can be decomposed into a direct sum of the regular representations of $\mathbb{F}[x]/f_i(x)$.

**Theorem 2.2 (Matrix Version of the CRT).** Let $R$ be the linear transformation, from the CRT, that maps $\mathbb{F}[x]/f(x)$ onto $\mathbb{F}[x]/f_1(x) \times \cdots \times \mathbb{F}[x]/f_t(x)$: $R(A(x)) = (A(x) \mod f_1(x), \ldots, A(x) \mod f_t(x))$. Then

$$R\rho(A) = (\rho(A_1) \oplus \cdots \oplus \rho(A_t))R.$$

**Proof.**

$$R\rho(A)B = R(AB) = (A_1B_1, \ldots, A_tB_t) = (\rho(A_1) \oplus \cdots \oplus \rho(A_t))(B_1, \ldots, B_t) = (\rho(A_1) \oplus \cdots \oplus \rho(A_t))RB.$$

Since $B$ is arbitrary the equation in the theorem is true. □
Example 1. Let \( f(x) = x^4 - 1 \), and let \( f_1(x) = x - 1 \), \( f_2(x) = x + 1 \), and \( f_3(x) = x^2 + 1 \) be the irreducible rational factors of \( f(x) \). Let \( A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 \) be an element of \( \mathbb{Q}(x)/f(x) \) (coefficients could come from any extension of \( \mathbb{Q} \)). Since \( A(x) \mod f_1(x) = a_0 + a_1 + a_2 + a_3 \), \( A(x) \mod f_2(x) = a_0 - a_1 + a_2 - a_3 \), and \( A(x) \mod f_3(x) = (a_0 - a_2) + (a_1 - a_3)x \),

\[
R = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 \\
1 & 0 & -1 & 0 \\
0 & 1 & 0 & -1
\end{bmatrix},
\]

with \( R(a_0, a_1, a_2, a_3)^T = (A \mod f_1, A \mod f_2, A \mod f_3) \). It is easy to verify that \( e_1(x) = 1/4(1 + x + x^2 + x^3) \), \( e_2(x) = 1/4(1 - x + x^2 - x^3) \), and \( e_3(x) = 1/2(1 - x) \) are a system of orthogonal idempotents. Therefore,

\[
R^{-1} = \begin{bmatrix}
1/4 & 1/4 & 1/2 & 0 \\
1/4 & -1/4 & 0 & 1/2 \\
1/4 & 1/4 & -1/2 & 0 \\
1/4 & -1/4 & 0 & -1/2
\end{bmatrix}.
\]

Consequently,

\[
R \begin{bmatrix}
a_0 & a_3 & a_2 & a_1 \\
a_1 & a_0 & a_3 & a_2 \\
a_2 & a_1 & a_0 & a_3 \\
a_3 & a_2 & a_1 & a_0
\end{bmatrix} R^{-1} = \begin{bmatrix}
a_0 + a_1 + a_2 + a_3 & 0 & 0 & 0 \\
0 & a_0 - a_1 + a_2 - a_3 & 0 & 0 \\
0 & 0 & a_0 - a_2 & a_3 - a_1 \\
0 & 0 & a_1 - a_3 & a_0 - a_2
\end{bmatrix}.
\]

2.2.2. Tensor product

The tensor product provides another important tool for deriving convolution algorithms. For this paper it is sufficient to consider the tensor product of finite dimensional algebras. Let \( U \) and \( V \) be vector spaces. A bilinear mapping \( \beta \) is a map from \( U \times V \rightarrow W \) such that

\[
\beta(\alpha_1u_1 + \alpha_2u_2, v) = \alpha_1\beta(u_1, v) + \alpha_2\beta(u_2, v)
\]

\[
\beta(u, \alpha_1v_1 + \alpha_2v_2) = \alpha_1\beta(u, v_1) + \alpha_2\beta(u, v_2).
\]

It is easy to verify that convolution is a bilinear mapping. More generally, multiplication in any algebra is a bilinear mapping due to the distributive property.

A vector space \( T \) along with a bilinear map \( \theta : U \times V \rightarrow U \otimes V \) is called a tensor product if it satisfies the properties:

1. \( \theta(U \times V) \) spans \( T \).
2. Given another vector space \( W \) and a bilinear mapping \( \varphi : U \times V \rightarrow W \) there exists a linear map \( \lambda : T \rightarrow W \) with \( \varphi = \theta \circ \lambda \).
The tensor product, denoted by $U \otimes V$, exists and is unique (see Lang, 1984). If $U$ and $V$ are finite dimensional and \{u_1, \ldots, u_m\} and \{v_1, \ldots, v_n\}$ are bases for $U$ and $V$, then \{u_1 \otimes v_1, \ldots, u_1 \otimes v_n, \ldots, u_m \otimes v_1, \ldots, u_m \otimes v_n\} is a basis for $U \otimes V$. It follows that the dimension of $U \otimes V$ is $mn$.

Let $A$ and $B$ be algebras and let $A \otimes B$ be the tensor product of $A$ and $B$ as vector spaces. Let $A_1, A_2 \in A_1$ and $B_1, B_2 \in A_2$, then $A \otimes B$ becomes an algebra with multiplication defined by $(A_1 \otimes B_1)(A_2 \otimes B_2) = A_1A_2 \otimes B_1B_2$. It is clear from this definition, that the regular representation $\rho(A \otimes B)$ is equal to $\rho(A) \otimes \rho(B)$.

When $A_1$ and $A_2$ are matrix algebras the tensor product coincides with the Kronecker product of matrices.

**Definition 2.3** (Kronecker Product). Let $A$ be an $m_1 \times n_1$ and $B$ be an $m_2 \times n_2$ matrix. The Kronecker product of $A$ and $B$, $A \otimes B$ is the $m_1m_2 \times n_1n_2$ block matrix whose $(i, j)$ block, for $0 \leq i < m_1$ and $0 \leq j < n_1$ is equal to $a_{i,j}B$.

**Example 2.**

$$\mathbb{F}[x, y]/(f(x), g(y)) \cong \mathbb{F}[x]/f(x) \otimes \mathbb{F}[y]/g(y).$$

Consider the bilinear map $\mathbb{F}[x]/f(x) \otimes \mathbb{F}[y]/g(y) \rightarrow \mathbb{F}[x, y]/(f(x), g(y))$ defined by $(A(x), B(y)) \rightarrow A(x)B(y)$. This map is onto since the collection of binomials $x^iy^j$ span $\mathbb{F}[x, y]/(f(x), g(y))$. Property 2 of the tensor product follows by setting $\lambda(x' y') = \varphi(x', y')$ for any other bilinear map $\varphi$.

If $\deg(f) = m$ and $\deg(g) = n$, then \{1, x, \ldots, x^{m-1}\} is a basis for $\mathbb{F}[x]/f(x)$ and \{1, y, \ldots, y^{n-1}\} is a basis for $\mathbb{F}[y]/g(y)$. With respect to these bases, $\rho(x) = C_f$ and $\rho(y) = C_g$. Using the basis $x^iy^j = x^i \otimes y^j$ for $0 \leq i < m, 0 \leq j < n$, $\rho(x \otimes y) = \rho(x) \otimes \rho(y) = C_f \otimes C_g$. In particular, $\mathbb{F}[x]/(x^m - 1) \otimes \mathbb{F}[y]/(y^n - 1)$ corresponds to two-dimensional convolution and the regular representation has a block circulant structure. For example, when $m = n = 2$, the regular representation is given by

$$a_0(I_2 \otimes I_2) + a_1(I_2 \otimes S_2) + a_2(S_2 \otimes I_2) + a_3(S_2 \otimes S_2) = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_3 & a_0 & a_1 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix}.$$  

### 2.3. Bilinear algorithms

A bilinear algorithm (Winograd, 1980) is a canonical way to describe algorithms for computing bilinear mappings. The purpose of this section is to provide a formalism for the constructions in Winograd (1980) that can be used in the computer manipulation of convolution algorithms. Similar notation has been used by other authors (Johnson et al., 1991; Tolimieri et al., 1997).

**Definition 2.4** (Bilinear Algorithm). A bilinear algorithm is a bilinear mapping denoted by the triple $(C, A, B)$ of matrices, where the column dimension of $C$ is equal to the row dimensions of $A$ and $B$. When applied to a pair of vectors $u$ and $v$ the bilinear algorithm $(C, A, B)$ computes $C(Au \bullet Bv)$, where $\bullet$ represents component-wise multiplication of vectors.
2.3.1 Operations on bilinear algorithms

Example 3. Consider a two-point linear convolution

\[
\begin{bmatrix}
u_0 \\
u_1
\end{bmatrix} \ast \begin{bmatrix}
v_0 \\
v_1
\end{bmatrix} = \begin{bmatrix}
u_0 v_0 \\
u_0 v_1 + u_1 v_0 \\
u_1 v_1
\end{bmatrix}.
\]

This can be computed with three instead of four multiplications using the following algorithms.

1. \(t_0 \leftarrow u_0 v_0\);
2. \(t_2 \leftarrow u_1 v_1\);
3. \(t_1 \leftarrow (u_0 + u_1)(v_0 + v_1) - t_0 - t_2\).

The desired convolution is given by the vectors whose components are \(t_0, t_1, \text{ and } t_2\). This algorithm is equivalent to the bilinear algorithm

\[
tc_2 = \begin{bmatrix}
1 & 0 & 0 \\
-1 & 1 & -1 \\
0 & 0 & 1
\end{bmatrix} \cdot \begin{bmatrix}
1 & 0 \\
1 & 1 \\
0 & 1
\end{bmatrix} = \begin{bmatrix}
1 & 0 \\
1 & 1 \\
0 & 1
\end{bmatrix}.
\]

2.3.1 Operations on bilinear algorithms

Let \(B_1 = (C_1, A_1, B_1)\) and \(B_2 = (C_2, A_2, B_2)\) be two bilinear algorithms. The following operations are defined for bilinear algorithms.

1. **[Direct sum]** \(B_1 \oplus B_2 = (C_1 \oplus C_2, A_1 \oplus A_2, B_1 \oplus B_2)\).
2. **[Tensor product]** \(B_1 \otimes B_2 = (C_1 \otimes C_2, A_1 \otimes A_2, B_1 \otimes B_2)\).
3. **[Product]** Assuming compatible row and column dimensions, \(B_1 B_2 = (C_2 C_1, A_1 A_2, B_1 B_2)\).

As a special case of the product of two bilinear algorithms, let \(P\) and \(Q\) be matrices and assume compatible row and column dimensions.

\[PB_1 Q = (PC_1, A_1 Q, B_1 Q)\] 

These operations provide algorithms to compute the corresponding bilinear maps.

**Lemma 2.1** (Tensor Product of Bilinear Mappings). Let \(B_1 = (C_1, A_1, B_1)\) and \(B_2 = (C_2, A_2, B_2)\) be two bilinear algorithms that compute \(\beta_1 : U_1 \times V_1 \longrightarrow W_1\) and \(\beta_2 : U_2 \times V_2 \longrightarrow W_2\) respectively. Then \(B_1 \otimes B_2\) computes the bilinear mapping \(\beta_1 \otimes \beta_2 : U_1 \otimes U_2 \times V_1 \otimes V_2 \longrightarrow W_1 \otimes W_2\) defined by \(\beta_1 \otimes \beta_2(u_1 \otimes v_1, u_2 \otimes v_2) = \beta_1(u_1, v_1) \otimes \beta_2(u_2, v_2)\).

**Proof.**

\[
B_1 \otimes B_2(u_1 \otimes u_2, v_1 \otimes v_2) = (C_1 \otimes C_2)((A_1 \otimes A_2)(u_1 \otimes u_2))
\]

\[
\bullet (B_1 \otimes B_2)(v_1 \otimes v_2))
\]

\[
= (C_1 \otimes C_2)(A_1 u_1 \otimes A_2 u_2) \bullet (B_1 v_1 \otimes B_2 v_2)
\]

\[
= (C_1 \otimes C_2)((A_1 u_1 \bullet B_1 v_1) \otimes (A_2 u_2 \bullet B_2 v_2))
\]

\[
= (C_1 (A_1 u_1 \bullet B_1) v_1) \otimes (C_2 (A_2 u_2 \bullet B_2) v_2))
\]
by factoring $A$ (convolution of size $N$. Then Theorem 2.3 (Bilinear Algorithm Corresponding to the CRT). Assume that $f(x) = f_1(x) \cdots f_t(x)$ in $\mathbb{F}[x]$, where $\gcd(f_i(x), f_j(x)) = 1$ for $i \neq j$, and let $(C_i, A_i, B_i)$ be a bilinear algorithm to multiply elements of $\mathbb{F}[x]/f_i(x)$. Then there exists an invertible matrix $R$ such that the bilinear algorithm

$$R^{-1} \left( \bigoplus_{i=1}^t (C_i, A_i, B_i) \right) R$$

computes multiplication in $\mathbb{F}[x]/f(x)$.

In filtering applications it is often the case that one of the inputs to be cyclically convolved is fixed. Fixing one input in a bilinear algorithm leads to a linear algorithm. When this is the case, one part of the bilinear algorithm can be precomputed and the precomputation does not count towards the cost of the algorithm. Let $(C, A, B)$ be a bilinear algorithm for cyclic convolution and assume that the first input is fixed. Then the computation $(C, A, B)(u, v)$ is equal to $(C \, \text{diag}(Au) \, B)v$, where diag$(Au)$ is the diagonal matrix whose diagonal elements are equal to the vector $Au$.

In most cases the $C$ portion of the bilinear algorithm is much more costly than the $A$ or $B$ portion of the algorithm, so it would be desirable if this part could be precomputed. Given a bilinear algorithm for a cyclic convolution, the matrix exchange property allows one to exchange the $C$ and $A$ or $B$ matrices.

Theorem 2.4 (Matrix Exchange). Let $J_N$ be the anti-identity matrix of size $n$ defined by $J_N : i \mapsto n-1-i$ for $i = 0, \ldots, N-1$, and let $(C, A, B)$ be a bilinear algorithm for cyclic convolution of size $N$. Then $(J_N B^T, A, C^T J_N)$, where $(\cdot)^T$ denotes matrix transposition, is a bilinear algorithm for cyclic convolution of size $N$.

Proof. Since $J_N S_N J_N = S_N^T$ and $J_N^{-1} = J_N$, $\text{Circ}_N(u) = J_N \text{Circ}_N(u)^T J_N$. Therefore,

$$u \otimes v = \text{Circ}_N(u)v$$

$$= (J_N \text{Circ}_N(u)^T J_N)v$$

$$= (J_N (C \text{diag}(Au) B)^T J_N)v$$

$$= (J_N B^T \text{diag}(Au) C^T J_N)v$$

$$= (J_N B^T, A, C^T J_N)(u, v). \square$$

2.4. Linear algorithms and matrix factorizations

Many fast algorithms for computing $y = Ax$ for a fixed matrix $A$ can be obtained by factoring $A$ into a product of structured sparse matrices. Such algorithms can be
represented by formulas containing parameterized matrices and a small collection of operators such as matrix composition, direct sum, and the tensor product.

An important example is provided by the FFT which is obtained from a factorization of the DFT matrix. Let \( \text{DFT}_n = [\omega_{nk}^j]_{0 \leq k, l < n}, \omega_n = \exp(2\pi i/n) \), then

\[
\text{DFT}_{rs} = (\text{DFT}_r \otimes I_s) T^{rs}_r (I_r \otimes \text{DFT}_s) L^{rs}_r,
\]

where \( I_n \) is the \( n \times n \) identity matrix, \( L^{rs}_r \) is the \( rs \times rs \) stride permutation matrix

\[
L^{rs}_r : j \mapsto j \cdot r \mod rs - 1, \quad rs = rs - 1 \mapsto rs - 1,
\]

and \( T^{rs}_r \) is the diagonal matrix of twiddle factors,

\[
T^{rs}_r = \bigoplus_{j=0}^{s-1} \text{diag}(\omega_n^0, \ldots, \omega_n^{r-1})^j, \quad \omega_n = e^{2\pi i/n}, \quad i = \sqrt{-1}.
\]

For example,

\[
\text{DFT}_4 = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & -1 & i
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & -1 & 0 \\
0 & 1 & 0 & -1
\end{bmatrix} \cdot \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \cdot \begin{bmatrix}
1 & 1 & 0 & 0 \\
1 & -1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & -1
\end{bmatrix} \cdot \bigoplus_{j=0}^{3} \text{diag}(\omega_n^0, \ldots, \omega_n^{3})^j
\]

\[
= (\text{DFT}_2 \otimes I_2) \cdot T^4_2 \cdot (I_2 \otimes \text{DFT}_2) \cdot L^4_2.
\]

See Johnson et al. (1990), Tolimieri et al. (1997) and VanLoan (1992) for a more complete discussion.

The tensor product satisfies the following basic properties, where indicated inverses exist, and matrix dimensions are such that all products make sense.

1. \((\alpha A) \otimes B = A \otimes (\alpha B) = \alpha(A \otimes B)\).
2. \((A + B) \otimes C = (A \otimes C) + (B \otimes C)\).
3. \(A \otimes (B + C) = (A \otimes B) + (A \otimes C)\).
4. \(1 \otimes A = A \otimes 1 = A\).
5. \(A \otimes (B \otimes C) = (A \otimes B) \otimes C\).
6. \((A \otimes B)^T = A^T \otimes B^T\).
7. \((A \otimes B)(C \otimes D) = AC \otimes BD\).
8. \(A \otimes B = (I_{m_1} \otimes B)(A \otimes I_{m_2}) = (A \otimes I_{n_2})(I_{n_1} \otimes B)\).
9. \((A_1 \otimes \cdots \otimes A_t)(B_1 \otimes \cdots \otimes B_t) = (A_1 B_1 \otimes \cdots \otimes A_t B_t)\).

10. \((A_1 \otimes B_1) \cdots (A_t \otimes B_t) = (A_1 \cdots A_t \otimes B_1 \cdots B_t)\).

11. \((A \otimes B)^{-1} = A^{-1} \otimes B^{-1}\).

12. \(I_m \otimes I_n = I_{mn}\).

All of these identities follow from the definition or simple applications of preceding properties (see Horn and Johnson, 1991).

The following additional properties will be required.

**Theorem 2.5 (Commutation Theorem).** Let \(A\) be an \(m_1 \times n_1\) matrix and let \(B\) be an \(m_2 \times n_2\) matrix. Then
\[
L_{m_1 m_2}(A \otimes B)L_{n_1 n_2} = (B \otimes A).
\]

More generally, if \(A_i, i = 1, \ldots, t\) is an \(n_i \times n_i\) matrix, and \(\sigma\) is a permutation of the indices \(\{1, \ldots, t\}\), there is a permutation matrix \(P_\sigma\) such that
\[
P_\sigma^{-1}(A_1 \otimes \cdots \otimes A_t)P_\sigma = A_{\sigma(1)} \otimes \cdots \otimes A_{\sigma(t)}.
\]

The proof of the commutation theorem can be found in Johnson et al. (1990), and the following property easily follows from the commutation theorem.

**Theorem 2.6 (Distributive Property of the Tensor Product).** Let \(A\) be an \(m \times n\) matrix and let \(B_i, i = 1, \ldots, t\) be an \(m_i \times n_i\) matrix. Then
\[
A \otimes (B_1 \oplus \cdots \oplus B_t) = L_{mn}(A \otimes B_1) \oplus \cdots \oplus (A \otimes B_t)
\]
\[
A \otimes (B_1 \oplus B_2) = L_{mnm_1}^{m_1}L_{mnm_2}^{m_2} \oplus (A \otimes B_1) \oplus (A \otimes B_2).
\]

### 3. Survey of convolution algorithms and techniques

This section surveys algorithms for linear and cyclic convolution in a form that is convenient for automatic generation. All of the algorithms are presented using the uniform mathematical notation of bilinear algorithms and are derived systematically using polynomial algebra and properties of the tensor product. Algorithms implicitly refer to bilinear algorithms, and operations on bilinear algorithms use the definitions in Section 2.3.

#### 3.1. Linear convolution

**3.1.1. Standard algorithm**

In a few rare cases, the standard method of multiplying polynomials learned in high school might be the best choice for a linear convolution algorithm. This can be turned into a bilinear algorithm of matrices in the obvious way.
Example 4. A $3 \times 3$ linear convolution given by the standard algorithm is:

$$s_{b3} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{bmatrix},$$

$$= (s_{b3}[C], s_{b3}[A], s_{b3}[B]).$$

3.1.2. Toom–Cook algorithm

The Toom–Cook algorithm (Toom, 1963; Cook, 1966; Knuth, 1981) uses evaluation and interpolation to compute the product of two polynomials. To compute the product $h(x) = f(x)g(x)$, where $f$ and $g$ are $N - 1$ degree polynomials, first evaluate each polynomial at $2^{N-1}$ distinct values $\alpha_i$. Next compute the $2^{N-1}$ multiplications $h(\alpha_i) = f(\alpha_i)g(\alpha_i)$. Finally, use the $2^{N-1}$ points $(\alpha_i, h(\alpha_i))$ and the Lagrange interpolation formula to recover $h(x)$.

$$h(x) = \sum_{j=0}^{2N-2} h(\alpha_i) \prod_{k \neq j} \frac{x - \alpha_k}{\alpha_j - \alpha_k}.$$ 

This algorithm can be expressed as a bilinear algorithm using the following notation.

Definition 3.1 (Bar Notation). Let $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ then $\overline{A(x)}$ denotes the equivalent vector $[a_0 \ a_1 \ \ldots \ a_n]^T$.

Definition 3.2 (Vandermonde Matrix).

$$V[\alpha_0, \ldots, \alpha_n] = \begin{bmatrix}
1 & \alpha_0 & \alpha_0^2 & \cdots & \alpha_0^n \\
1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \alpha_n & \alpha_n^2 & \cdots & \alpha_n^n
\end{bmatrix}.$$ 

The matrix $V$ applied to the vector of coefficients of $f(x)$ is equal to the vector containing the evaluations $f(\alpha_0), f(\alpha_1), \ldots, f(\alpha_n)$, and applying $V^{-1}$ to the vector of evaluations returns the original coefficients. Therefore $V^{-1}$ corresponds to interpolation and can be computed using Lagrange’s formula. The following theorem summarizes these observations.

Theorem 3.1 (Toom–Cook Algorithm). The bilinear algorithm $(V^{-1}, V', V')$, where $V'$ is the $(2N-1) \times N$ matrix containing the first $N$ columns of $V[\alpha_0, \ldots, \alpha_{2N-1}]$, computes the $N$-point linear convolution of two vectors.
This theorem is a special case of Theorem 2.3 and follows from the CRT applied to \( f(x) = \prod_{i=0}^{2N-1} (x - \alpha_i) \). The matrix \( R \) in this case is the Vandermonde matrix \( V[\alpha_0, \ldots, \alpha_{2N-1}] \).

The Toom–Cook algorithm reduces the number of "general" multiplications from \( N^2 \) (computed by definition) to \( 2N-1 \) at the cost of more additions. A general multiplication is one that cannot be precomputed at compile time, or reduced to a series of additions at run-time. For small input sizes when there are sufficiently many convenient evaluation points such as 0, 1, \(-1\), \(\infty\), then the reduction in general multiplications corresponds to a reduction in actual multiplications. If \( f(x) = f_0 + f_1 x + \cdots + f_k x^k \), with \( f_k \) non-zero, then \( f(\infty) = f_k \).

Example 3 corresponds to the Toom–Cook algorithm using evaluation points 0, 1, \(\infty\), and the following 3-point example uses evaluation points 0, 1, \(-1\), 2, \(\infty\).

Example 5. A 3 \times 3 linear convolution given by the Toom–Cook algorithm is:

\[
\begin{bmatrix}
1 & 0 & 0 \\
-1/2 & 1 & -1/3 \\
1/2 & -1/2 & -1/6 \\
0 & 0 & 0
\end{bmatrix}
= (tc_3[C], tc_3[A], tc_3[B]).
\]

Note further that the algorithm can be improved to use fewer operations by using:

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
1 & -1 & 1 \\
1 & 2 & 4 \\
0 & 0 & 1
\end{bmatrix}
= (tc_3[C], tc_3[A], tc_3[B]).
\]

3.1.3. Combining linear convolutions

The tensor product can be used to combine small linear convolution algorithms into larger ones in an efficient manner. This is important, because the tensor product of smaller convolution algorithms will generally use fewer operations than a direct larger convolution algorithm. For example combining a Toom–Cook algorithm of size 2 with a Toom–Cook algorithm of size 3, creates a linear convolution of size 6 that uses many fewer operations than a Toom–Cook convolution of size 6.

Theorem 3.2 (Tensor Product of Linear Convolutions). Let \( \mathcal{L}_m \) and \( \mathcal{L}_n \) be bilinear algorithms for linear convolution of sizes \( m \) and \( n \) respectively. Then \( O_{m,n}(\mathcal{L}_m \otimes \mathcal{L}_n) \) is a bilinear algorithm for linear convolution of size \( mn \), where \( O_{m,n} \) is a sparse
The proof is most easily seen from the polynomial interpretation of convolution. Let \( a(x) \) and \( b(x) \) be polynomials of degree \( mn - 1 \), and let

\[
A(x, y) = \sum_{i=0}^{n-1} A_i(x)y^i \quad \text{and} \quad B(x, y) = \sum_{j=0}^{n-1} B_j(x)y^j,
\]

where \( A_i(x) \) and \( B_j(x) \) are polynomials of degree \( m - 1 \). Substituting \( y = x^m \), \( a(x) = A(x, x^m) \) and \( b(x) = B(x, x^m) \). Consequently, if \( C(x, y) = A(x, y)B(x, y) \), then \( c(x) = C(x, x^m) \). By Lemma 2.1 and Example 2, \( \mathcal{L}_m \otimes \mathcal{L}_n \) computes \( C(x, y) \). The matrix \( O_{m,n} \) corresponds to the reduction obtained from substituting \( y = x^m \) into \( C(x, y) \). □

Example 6.

\[
O_{2,3} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 &
\]

The following generalization is obtained using induction and simple properties of the tensor product.

**Theorem 3.3.** Let \( N = n_1, \ldots, n_t \) and let \( \mathcal{L}_{n_{i}} \), \( 0 \leq i < t \) be linear convolution algorithms of size \( n_{i} \), then \( O_{n_1 \cdots n_t} (\mathcal{L}_{n_1} \otimes \cdots \otimes \mathcal{L}_{n_t}) = \mathcal{L}_{n_1 \cdots n_t} \), where \( O_{n_1 \cdots n_t} \) is a sparse \( (2n_1 - 1) \cdots (2n_t - 1) \times (2N - 1) \) matrix defined by \( O_{n_1 \cdots n_t} = O_{n_1, n_2 \cdots n_{t}} (I_{2n_1-1} \otimes O_{n_2, \ldots, n_t}) \).

### 3.2. Linear convolution via cyclic convolution

In the next section, fast algorithms for performing cyclic convolution, and more generally for multiplying two polynomials modulo a polynomial are discussed. Tolimieri et al. (1997) points out that linear convolution can be obtained from generalized cyclic convolution corresponding to polynomial multiplication modulo a polynomial. For example, if \( g(x) = g_0 + g_1 x + g_2 x^2 \) and \( h(x) = h_0 + h_1 x + h_2 x^2 \), then \( g(x)h(x) \) can be computed by first convolving \( g \) and \( h \) via a 4-point cyclic convolution and then adding the vector \( g_2 h_2 m(x) \) where \( m(x) = x^4 - 1 \). The following theorem expresses Tolimieri’s method in terms of bilinear algorithms.
Theorem 3.4 (Linear from Cyclic). Let \( g(x), h(x) \) be polynomials of degree \( n - 1 \) and \( m(x) = x^{2n-2} + \sum_{i=0}^{2n-3} m_i x^i \), be a monic polynomial of degree \( 2n - 2 \). Assume that \((C_m, A_m, B_m)\) is a bilinear algorithm that computes \( g(x)h(x) \mod m(x) \). Then the bilinear algorithm \((C, A, B)\) computes \( f(x)g(x) \), where

\[
C = \begin{bmatrix}
1 & m_0 \\
& \ddots & \ddots & \ddots \\
& & 1 & m_{2n-3} \\
& & & 1 & 1
\end{bmatrix},
\]

\[
A = \begin{bmatrix}
A_m & 1 \\
& \ddots & \ddots & \ddots \\
& & 1 & 1 \\
& & & 1
\end{bmatrix},
\]

\[
B = \begin{bmatrix}
B_m & 1 \\
& \ddots & \ddots & \ddots \\
& & 1 & 1 \\
& & & 1
\end{bmatrix}.
\]

Proof. Let \( c(x) = \frac{g(x)h(x)}{m(x)} \). Therefore, \( f(x)g(x) = c(x) + q(x)m(x) \), and since \( m(x) \) is monic and of degree \( 2n - 2 \), \( g(x)h(x) = c(x) + g_n h_n m(x) \).

\[
(C, A, B)(g, h) = \begin{bmatrix}
1 & m_0 \\
& \ddots & \ddots & \ddots \\
& & 1 & m_{2n-3} \\
& & & 1 & 1
\end{bmatrix} \begin{bmatrix}
C_m(A_m g \bullet B_m h) \\
& 0 \\
& \ddots \\
& & C_m(A_m g \bullet B_m h)
\end{bmatrix} \\
= \frac{g(x)h(x) \mod m(x) + g_n h_n m(x)}{g(x)h(x)}. \qed
\]

3.3. Cyclic convolution

Convolution modulo \( f(x) \) refers to polynomial multiplication modulo a third polynomial. Algorithms for convolution modulo \( f(x) \) can be obtained from linear convolution algorithms by multiplying by a matrix, which corresponds to computing the remainder in division by \( f(x) \). Let \( M(f(x)) \) denote the reduction matrix defined by \( M(f(x))A(x) = A(x) \mod f(x) \). The exact form of \( M(f(x)) \) depends on the degree of \( A(x) \). If \((C, A, B)\) is a bilinear algorithm for linear convolution, then \((M(f(x))C, A, B)\) is a bilinear algorithm for convolution modulo \( f(x) \).

Example 7. Composing

\[
M(x^2 - 1) = \begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 0
\end{bmatrix}.
\]
with the Toom–Cook bilinear algorithm of (6) gives the bilinear algorithm

\[
(M(x^2 - 1)C_2, A_2, B_2) = \begin{pmatrix}
1 & 0 & 1 \\
-1 & 1 & -1 \\
1 & 1 & 1 \\
0 & 1 & 0
\end{pmatrix}
\]

for 2-point cyclic convolution.

### 3.3.1. Convolution theorem

The well-known convolution theorem provides a bilinear algorithm for computing cyclic convolution.

**Theorem 3.5** (Convolution Theorem). The bilinear algorithm \((\text{DFT}^{-1}_N, \text{DFT}_N, \text{DFT}_N)\) computes \(N\)-point cyclic convolution.

**Proof.** Let \(\omega_N\) be a primitive \(N\)th root of unity, then \(x^N - 1 = \prod_{i=0}^{N-1} (x - \omega_i^j)\). Since, \(V[1, \omega_N, \ldots, \omega_N^{N-1}] = \text{DFT}_N\), the convolution theorem follows from Theorem 2.3.

When \(N = RS\), \(x^N - 1 = \prod_{i=0}^{S-1} (x^R - \omega_S^j)\). Applying the CRT to this factorization leads to the following theorem which allows the DFT to be combined with other convolution algorithms.

**Theorem 3.6.** Let \(N = RS\) and let \(C_i, i = 0, \ldots, S-1\), be bilinear algorithms to multiply two polynomials modulo \(x^R - \omega_S^j\). Then

\[
(\text{DFT}^{-1}_S \otimes I_R) \left( \bigoplus_{i=0}^{S-1} C_i \right) (\text{DFT}_S \otimes I_R)
\]

is a bilinear algorithm to compute \(N\)-point cyclic convolution.

**Proof.** Let \(f(x)\) be a polynomial of degree \(N - 1\) and write \(f(x) = \sum_{j=0}^{S-1} f_j(x)x^{Rj}\), where \(\deg(f_j(x)) < R\). Then \(f(x) \mod x^R - \omega_S^j\) \(= \sum_{j=0}^{S-1} f_j(x)\alpha_S^j\). Therefore, the matrix \(R = [R_0 R_1 \ldots R_{S-1}]^T\) with \(R_i f = f(x) \mod x^R - \omega_S^j\) is equal to \(\text{DFT}_S \otimes I_R\).

Note that multiplication modulo \(x^R - \alpha\) can easily be transformed into cyclic convolution. Observe that if \(\beta^R = \alpha\), and \(h(x) = f(x)g(x) \mod x^R - \alpha\), then

\[
h_{\beta}(x) = h(\beta x) = f(\beta x)g(\beta x) \mod (\beta x)^R - \alpha
\]

\[
= f(\beta x)g(\beta x) \mod (\beta x)^R - \alpha
\]

\[
= f(\beta x)g(\beta x) \mod \alpha(x^R - 1).
\]

Therefore, \(h(x) = h_{\beta}(x/\beta)\).

Applying this observation and the previous theorem leads to the following construction related to the FFT shown in (7).

**Theorem 3.7.** Let \(C_R\) be a bilinear algorithm to compute \(R\)-point cyclic convolution, and let \(F_S = ((\text{DFT}_S \otimes I_R), T^N_R(\text{DFT}_S \otimes I_R), T^N_R(\text{DFT}_S \otimes I_R))\). Then \((I_S \otimes C_R)F_S\) computes \(N\)-point cyclic convolution.
3.3.2. Winograd

Winograd’s (1977) algorithm for computing cyclic convolution follows from the CRT when applied to the irreducible rational factors of the polynomial $X^N - 1$. The irreducible rational factors of $x^N - 1$ are called cyclotomic polynomials.

**Definition 3.3** (Cyclotomic Polynomials). The cyclotomic polynomial $\Phi_N(x)$ can be defined recursively from the formula

$$x^N - 1 = \prod_{d | N} \Phi_d(x).$$

Alternatively

$$\Phi_N(x) = \prod_{\gcd(j, N) = 1} (x - \omega_j^N),$$

where $\omega_N$ is a primitive $N$th root of unity. It follows that $\deg(\Phi_N(x)) = \phi(N)$, where $\phi$ is the Euler $\phi$ function. It is well known (Lang, 1984) that $\Phi_N(x)$ has integer coefficients and is irreducible over the rationals.

Applying Theorem 2.3 to $x^N - 1 = \prod_{d | N} \Phi_d(x)$ leads to the following algorithm.

**Theorem 3.8** (Winograd Convolution Algorithm). Let $C_f$ denote a bilinear algorithm that multiplies elements of $\mathbb{C}[x]/f(x)$. Then

$$C = R^{-1} \left( \bigoplus_{d | n} C_{\Phi_d(x)} \right) R$$

where $R = [R_{d_1} R_{d_2} \ldots R_{d_k}]^T$ and $R_{d_i} \bar{f} = f(x) \mod \Phi_d(x)$ is a bilinear algorithm for $N$-point cyclic convolution.

Using the 2-point cyclic convolution algorithm in Example 7 and the cyclotomic polynomials $\Phi_1(x) = (x - 1)$, $\Phi_2(x) = (x + 1)$, and $\Phi_4(x) = (x^2 + 1)$ yields the following 4-point cyclic convolution algorithm.

**Example 8.**

$$R_4^{-1} \left[ \begin{array}{ccc} 1 & 1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & -1 \end{array} \right] \cdot \left[ \begin{array}{ccc} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{array} \right] R_4,$$

where $R_4$ is the matrix $R$ in Example 1.

The results in the next section provide a more efficient method for computing $R_4$. 
3.3.3. CRT-based cyclic convolution algorithms for prime powers

Selesnick and Burrus (1996) have shown that when $N = p^k$ is a prime power, the Winograd algorithm has additional structure. This structure follows from the properties

\[ \Phi_{p^k}(x) = x^{p^k-1} + \cdots + x + 1 \]

\[ \Phi_{p^l}(x) = \Phi_p(x^{p^k-1}). \]

The composition structure of $\Phi_{p^k}(x)$ provides an efficient way to compute $R_{p^k}$.

**Theorem 3.9.** Let $R_{p^k} = [R_0, R_p, \ldots, R_{p^k}]$ be the $p^k \times p^k$ reduction matrix where $R_p f(x) \equiv f(x) \mod \Phi_p(x)$ for $f(x)$ of degree $p^k - 1$. Then

\[ R_{p^k} = \begin{bmatrix} 1_p \otimes R_{p^{k-1}} \\ G_p \otimes I_{p^{k-1}} \end{bmatrix}, \]

where $G_n$ is the $(n-1) \times n$ matrix:

\[ G_n = \begin{bmatrix} 1 & -1 \\ \vdots & \ddots & \ddots & \ddots \\ & & 1 & -1 \end{bmatrix}, \]

and $1_n$ is the $1 \times n$ matrix filled with 1’s. Moreover, $R_{p^k} = (R_{p^{k-1}} \oplus I_{(p-1)p^{k-1}})(R_p \otimes I_{p^{k-1}})$.

**Proof.** First observe that if $f(x) = f_0 + f_1 x + \cdots + f_{m-1} x^{m-1} + x^m$ and $A(x) = \sum_{i=0}^{m} a_i x^i$, then $A(x) \mod f(x) = \sum_{i=0}^{m-1} (a_i - f_i)x^i$. Therefore reduction of $A(x)$ modulo $f(x)$ is given by

\[ R = \begin{bmatrix} 1 & \cdots & -f_0 \\ \vdots & \ddots & \ddots & \ddots \\ & & 1 & -f_{m-1} \end{bmatrix}. \]

When $f(x) = 1 + x + \cdots + x^{n-1}$ the matrix $G_n$ is obtained. Second observe that if $A(x) = \sum_{i=0}^{m} a_i x^{ni}$, where $\deg(A_i) < n$, then $A(x) \mod f(x^n) = \sum_{i=0}^{m} (A_i(x) - f_i A_{m}(x)) x^{ni}$. Therefore reduction of $A(x)$ modulo $f(x^n)$ is given by $R \otimes I_n$, and reduction modulo $\Phi_{p^k}(x) = \Phi_p(x^{p^k-1})$ is given by $G_p \otimes I_{p^{k-1}}$. Finally, since $x^{p^k-1} \mod \Phi_{p^{k-1}} = 1$, reduction of $A(x)$ modulo $\{ \Phi_{p^i}(x), i = 0, \ldots, k \}$ is given by $1_p \otimes R_{p^{k-1}}$. These observations prove the first part of the theorem. The factorization in the second part is obtained using the multiplicative property of the tensor product. \( \square \)

A simple block matrix multiplication provides the following computation of the inverse of $R_{p^k}$.

**Theorem 3.10.**

\[ R_{p^k}^{-1} = 1/p \left( 1_p \otimes R_{p^{k-1}}^{-1} \right) V_p \otimes I_{p^{k-1}}, \]
where \( V_n \) is the \( n \times (n - 1) \) matrix

\[
\begin{bmatrix}
-1 & -1 & -1 & \ldots & -1 \\
-1 & -1 & -1 & \ldots & -1 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
-1 & \ldots & -1 & n - 1 & -1 \\
-1 & \ldots & -1 & -1 & n - 1 \\
-1 & \ldots & -1 & -1 & -1 \\
\end{bmatrix}
\]

Moreover, \( R_p^{-1} = (R_p^{-1} \odot I_p^{p-1}) (R_p^{-1} \odot I_{(p-1)p^{p-1}}) \).

**Example 9.** A bilinear algorithm for a cyclic convolution of size 27 is \((C, A, B)\), where:

\[
C = R_3^{-1}
\begin{bmatrix}
1 & M(x^6 + x^3 + 1) & M(x^{18} + x^9 + 1) \\
M(x^2 + x + 1) & L_2[A] & L_{18}[A] \\
L_2[A] & L_6[A] & R_3 \\
\end{bmatrix}
\]

\[
A = R_3^{-1}
\begin{bmatrix}
L_2[A] & L_6[A] & L_{18}[A] \\
\end{bmatrix}
\]

\[
B = R_3^{-1}
\begin{bmatrix}
L_2[B] & L_6[B] & L_{18}[B] \\
\end{bmatrix}
\]

where \( L_n = (L_n[A], L_n[B]) \) is a bilinear algorithm for a linear convolution of size \( n \) of any method, and

\[
R_3 =
\begin{bmatrix}
1 & 1 & 1 \\
1 & 0 & -1 \\
0 & 1 & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 \\
1 & 0 & -1 \\
0 & 1 & -1 \\
\end{bmatrix}
\odot I_3
\begin{bmatrix}
1 & 1 & 1 \\
1 & 0 & -1 \\
0 & 1 & -1 \\
\end{bmatrix}
\odot I_9
\]

### 3.3.4. The Agarwal–Cooley and split-nesting algorithms

The Agarwal and Cooley (1977) algorithm uses the tensor product to create a larger cyclic convolution from smaller cyclic convolutions. The split-nesting algorithm, due to Nussbaumer (1982), follows directly from Agarwal–Cooley using simple properties of the tensor product.

The Agarwal–Cooley algorithm follows from the fact that when \( \gcd(m, n) = 1 \), the algebra \( \mathbb{F}[x]/(x^{mn} - 1) \) is isomorphic to \( \mathbb{F}[y, z]/(y^m - 1, z^n - 1) \), which by Example 2 is isomorphic to \( \mathbb{F}[y]/(y^m - 1) \odot \mathbb{F}[z]/(z^n - 1) \). The isomorphism is obtained by mapping \( x \) to \( yz \) which maps \( x^i \) to \( y^{(i \mod m)} z^{(i \mod n)} \). Using the reordering required by this mapping and Lemma 2.1 leads to the following theorem which shows how to build an \( mn \)-point cyclic convolution algorithm from the tensor product of \( m \)-point and \( n \)-point cyclic convolution algorithms.

**Theorem 3.11** (Agarwal–Cooley Algorithm). Assume \( \gcd(m, n) = 1 \) and let \( C_m = (C_m, A_m, B_m) \) and \( C_n = (C_n, A_n, B_n) \) be bilinear algorithms for cyclic convolution of size \( m \) and \( n \). Let \( Q_{m,n}^{-1} \) be the permutation that maps \( i \) to \( (i \mod m)n + (i \mod n) \). Then \( Q_{m,n}^{-1}(C_m \odot C_n) Q_{m,n} \) computes a cyclic convolution of size \( mn \).
Note that the permutation $Q_{m,n}$ is defined by the mapping $i + j \mapsto i e_m + j e_n \mod mn$, $0 \leq i < m$, $0 \leq j < n$, where $e_m \equiv 1 \mod m$, $e_m \equiv 0 \mod n$, $e_n \equiv 0 \mod m$, $e_n \equiv 1 \mod n$, are the idempotents defining the CRT mapping for the integers $m$ and $n$.

Let $R_{m}^{-1} \left( \bigoplus_{i=0}^{k_1} C_{m_i} \right) R_{m}$ and $R_{n}^{-1} \left( \bigoplus_{j=0}^{k_2} C_{n_j} \right) R_{n}$ be bilinear algorithms to compute $m$ and $n$-point Winograd cyclic convolutions. Then combining Agarwal–Cooley with the Winograd algorithm yields the bilinear algorithm

$$Q_{m,n}^{-1} \left( R_{m}^{-1} \left( \bigoplus_{i=0}^{k_1} C_{m_i} \right) R_{m} \right) \otimes \left( R_{n}^{-1} \left( \bigoplus_{j=0}^{k_2} C_{n_j} \right) R_{n} \right) Q_{m,n}$$

(11)

for computing an $mn$-point cyclic convolution, (provided $\gcd(m,n) = 1$). Using the multiplicative property of the tensor product, this is equal to

$$Q_{m,n}^{-1} \left( R_{m}^{-1} \otimes R_{n}^{-1} \right) \left( \bigoplus_{i=0}^{k_1} C_{m_i} \otimes \bigoplus_{j=0}^{k_2} C_{n_j} \right) (R_{m} \otimes R_{n}) Q_{m,n}.$$  

(12)

Rearranging this equation into a double sum of tensor products leads to the “split-nesting algorithm” which was first derived by Nussbaumer (1982), who observed that it requires fewer additions then (11). The following theorem describes this transformation.

**Theorem 3.12** (Split Nesting). Let $C = \bigoplus_{i=0}^{s-1} C_i$ and $D = \bigoplus_{j=0}^{t-1} D_j$. Then

$$C \otimes D = P^{-1} \left( \bigoplus_{i=0}^{s-1} \bigoplus_{j=0}^{t-1} C_i \otimes D_j \right) P,$$

where $P$ is a permutation.

**Proof.** Using the first part of Theorem 2.6,

$$C \otimes D = \bigoplus_{i=0}^{s-1} \bigoplus_{j=0}^{t-1} C_i \otimes D_j = \bigoplus_{i=0}^{s-1} \left( \bigoplus_{j=0}^{t-1} C_i \otimes D_j \right).$$

Using the second part of Theorem 2.6, the previous equation is equal to

$$\bigoplus_{i=0}^{s-1} P_i^{-1} \left( \bigoplus_{j=0}^{t-1} C_i \otimes D_j \right) P_i,$$

which is equal to

$$P^{-1} \left( \bigoplus_{i=0}^{s-1} \bigoplus_{j=0}^{t-1} C_i \otimes D_j \right) P,$$

where $P = \bigoplus_{i=0}^{s-1} P_i$. □
Example 10. Let

\[ C_4 = R_4^{-1}(1 \oplus 1 \oplus C_2)R_4 \quad \text{and} \quad C_{27} = R_{27}^{-1}(1 \oplus D_2 \oplus D_6 \oplus D_{18})R_{27}, \]

where \( C_2 = M(x^2 + 1)C_2, \) \( D_2 = M(x^2 + x + 1)C_2, \) \( D_6 = M(x^6 + x^3 + 1)C_6, \) \( D_{18} = M(x^{18} + x^9 + 1)C_{18}, \) be the algorithms for cyclic convolution on four and 27 points given in Examples 8 and 9. By Agarwal–Cooley,

\[ Q_{4,27}^{-1}(R_4^{-1}(1 \oplus 1 \oplus C_2)R_4) \otimes (R_{27}^{-1}(1 \oplus D_2 \oplus D_6 \oplus D_{18})R_{27})Q_{4,27} \]

is an algorithm for cyclic convolution on 108 points. The split-nesting theorem transforms this algorithm into

\[ (Q_{4,27}^{-1}(R_4^{-1} \otimes R_{27}^{-1}))P^{-1}(1 \oplus D_2 \oplus D_6 \oplus D_{18}) \oplus (1 \oplus D_2 \oplus D_6 \oplus D_{18}) \oplus (C_2 \oplus C_2 \oplus D_2 \oplus D_6 \oplus D_6 \oplus D_2 \oplus D_{18})) \]

\[ P(R_4 \otimes R_{27})Q_{4,27} \]

where \( P = I_{27} \oplus I_{27} \oplus P_3 \) and \( P_3 = (I_2 \oplus L_2^4 \oplus L_2^{12} \oplus L_2^{36})L_{27}^{54}. \)

4. Optimization and operation counts

In this section different algorithms are compared using operation count. The concise algorithmic notation presented in this paper, makes it easy to compare and understand the tradeoffs of different convolution algorithms. In addition, new algorithms can be constructed with smaller operation counts than the best previously known results.

Starting with standard (computation by definition) and the Toom–Cook algorithms, the CRT Theorem 2.3 and the tensor product (Theorems 3.2 and 3.11) can be used to derive different algorithms for computing linear and cyclic convolution. Depending on which components are used and the order in which the constructions are applied, algorithms with different computational cost are obtained. Additional algorithmic choices can be obtained by rearranging the factors in the algorithm using properties of the tensor product and other algebraic manipulations.

The set of algorithms that can be obtained from these constructions defines a space of convolution algorithms, and for a given size finding the algorithm in the space with minimal cost becomes a well-defined optimization problem. In this paper, operation count is used as the cost function since it provides exact results and is easy to compare with previous work. However, using the automated algorithm generation and implementation outlined in Section 5 a similar optimization problem can be carried out using run-time for the cost function. This is being done as part of the SPIRAL project (Moura et al., 1998).

The following theorems can be used to determine the number of operations for the different algorithms in this space.

**Theorem 4.1.**

1. The number of operations required to apply \( R_{p,k} \) or \( R_{p,k}^T \) to an arbitrary vector using the factorization in Theorem 3.9 is \( 2(p^k - 1). \)
2. The number of operations required to apply $O_{n_1 \cdots n_t}$, defined in Theorem 3.2, to an arbitrary vector is $(2n_1 - 1) \cdots (2n_t - 1) - n_1 \cdots n_t \leq n_1 \cdots n_t$.

**Theorem 4.2** (Cost of the Tensor Product). Let $A$ be a $k_1 \times l_1$ matrix requiring $a_1$ additions and $m_1$ multiplications to apply a vector. Let $B$ be a $k_2 \times l_2$ matrix requiring $a_2$ additions and $m_2$ multiplications to apply a vector. Then

1. $A \oplus B$ can be applied to a vector using $a_1 + a_2$ additions and $m_1 + m_2$ multiplications.
2. $A \otimes B$ can be applied to a vector using $l_1 a_2 + k_2 a_1$ additions and $l_1 m_2 + k_2 m_1$ multiplications.

**Proof.** The result for the direct sum is obvious and the result for the tensor product follows from the factorization $A \otimes B = (A \otimes I_{k_2})(I_{l_1} \otimes B)$, and the commutation theorem which implies that up to a permutation $(A \otimes I_{k_2})$ is equal to $(I_{k_2} \otimes A)$. When $A$ and $B$ are rectangular matrices, using this factorization, the number of operations for $A \otimes B$ is not the same as for $B \otimes A$. □

When constructing convolution algorithms from smaller convolution algorithms using the CRT and the tensor product, the following strategies can be used to reduce operation count.

1. Order tensor products so as to minimize the operations given in Theorem 4.2.
2. Use optimal subalgorithms depending on the particular size involved. Beware that in general the tensor product of optimal algorithms is not optimal. Thus dynamic programming cannot be guaranteed to find optimal algorithms.

These optimizations and the comparison of different convolution algorithms are illustrated using a cyclic convolution of size $2^2 \cdot 3^3 = 108$. It is assumed that the convolution is used in a filtering application and that the matrix-exchange technique will be applied. Therefore the number of operations to convolve two vectors via a bilinear algorithm $(C, A, B)(x, y)$ to be performed at run-time is just the number of operations in $A^T$, $B$, plus $M$ multiplications required to multiply $By$ with the diagonal matrix created by the matrix exchange algorithm.

Table 1 lists the operation counts for linear convolution algorithms of sizes 4, 6, 9, 18, and 27 built from the Toom–Cook algorithms of sizes 2 and 3 and the standard algorithm of size 3 using the tensor product. The standard algorithm of size 2 is not considered since it never leads to an algorithm with lower operation count. The number of operations are easily computed using Theorem 4.2. For example, the number of operations required by $Lin_{18c}$ is equal to $3 \cdot \text{ops}(sb_3) + 3 \cdot \text{ops}(tc_2) + 9 \cdot 3$. There are several key points to be made from Table 1. First, there are significant differences in operation counts, depending on the way an algorithm is constructed. The data in the table shows that the combination of optimal algorithms is not necessarily optimal. For example, the optimal algorithm of size 18, $Lin_{18c}$, is not built from optimal algorithms for sizes 6 and 3 nor is it built from optimal algorithms for sizes 9 and 2. This shows that dynamic programming cannot always find optimal algorithms. Finally, notice that the Standard Bilinear algorithm is useful in this computation, even though it is ignored by most authors because asymptotically it is a bad choice compared to the Toom–Cook algorithm. It can be shown that asymptotically the
number of operations to compute $\text{Lin}_N$ using the tensor product of $tc_3$ is $N^\log_2(2k-1)$, however, this asymptotic result is not important for the range of sizes considered in this paper (for large input sizes, the convolution theorem used with the FFT, which is $O(N \log(N))$, should be used).

Cyclic convolution algorithms can be built from linear convolutions directly by reducing linear convolutions of the same size or indirectly using Winograd's algorithm. Winograd’s
algorithm replaces larger tensor products with a direct sum of smaller tensor products at the cost of a linear amount of extra additions. Using Winograd’s algorithm, cyclic convolution of size 4 can be built from linear convolutions of sizes 1 and 2, and cyclic convolution of size 27 can be built from linear convolutions of sizes 1, 2, 6, and 18. Table 3 shows the operation counts for cyclic convolution of sizes 4 and 27 using Winograd’s algorithm built from optimal linear convolutions found in Table 1. These results can be compared to the direct approach by comparing to the operation counts for the best linear algorithms of sizes 4 and 27 in Table 1.

Cyclic convolution of size 108 can be built, using Agarwal–Cooley, from cyclic convolution algorithms of sizes 4 and 27. Table 3 shows the operation counts for cyclic convolution of size 108 is shown in Table 3. Since matrix exchange is used, this can be computed with the same number of operations as the optimal algorithm of size 36. The idea of optimizing tensor products independent of dimension (tensor products of linear convolutions correspond to multi-dimensional convolution) simply using the optimal linear convolution of the given size for all of the summands in the split-nesting algorithm leads to an improved algorithm that will be called improved split nesting. The resulting operation count for computing cyclic convolution of size 108 is shown in Table 3.
Table 2
Operation counts for linear convolution

<table>
<thead>
<tr>
<th>Method</th>
<th>B</th>
<th>B</th>
<th>A^T</th>
<th>A^T</th>
<th>Diag</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lin_{12a} = O_{3,2,2}(sb_3 \otimes tc_2 \otimes tc_2)</td>
<td>15</td>
<td>0</td>
<td>84</td>
<td>0</td>
<td>81</td>
<td>180</td>
</tr>
<tr>
<td>Lin_{12b} = O_{2,2,2}(tc_2 \otimes sb_3 \otimes tc_2)</td>
<td>33</td>
<td>0</td>
<td>102</td>
<td>0</td>
<td>81</td>
<td>216</td>
</tr>
<tr>
<td>Lin_{12c} = O_{2,2,2}(tc_2 \otimes tc_2 \otimes sb_3)</td>
<td>45</td>
<td>0</td>
<td>114</td>
<td>0</td>
<td>81</td>
<td>240</td>
</tr>
<tr>
<td>Lin_{12d} = O_{2,2,2}(tc_2 \otimes tc_2 \otimes tc_3)</td>
<td>53</td>
<td>0</td>
<td>86</td>
<td>0</td>
<td>45</td>
<td>184</td>
</tr>
<tr>
<td>Lin_{12e} = O_{2,2,2}(tc_2 \otimes tc_3 \otimes tc_2)</td>
<td>63</td>
<td>0</td>
<td>96</td>
<td>0</td>
<td>45</td>
<td>204</td>
</tr>
<tr>
<td>Lin_{12f} = O_{3,2,2}(tc_3 \otimes tc_2 \otimes tc_2)</td>
<td>78</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>45</td>
<td>234</td>
</tr>
<tr>
<td>Lin_{36a} = O_{3,3,2,2}(sb_3 \otimes sb_3 \otimes tc_2 \otimes tc_2)</td>
<td>45</td>
<td>0</td>
<td>738</td>
<td>0</td>
<td>729</td>
<td>1512</td>
</tr>
<tr>
<td>Lin_{36b} = O_{3,2,3,3}(sb_3 \otimes tc_2 \otimes sb_3 \otimes tc_2)</td>
<td>99</td>
<td>0</td>
<td>792</td>
<td>0</td>
<td>729</td>
<td>1620</td>
</tr>
<tr>
<td>Lin_{36c} = O_{3,3,2,3}(sb_3 \otimes tc_2 \otimes tc_2 \otimes sb_3)</td>
<td>135</td>
<td>0</td>
<td>828</td>
<td>0</td>
<td>729</td>
<td>1692</td>
</tr>
<tr>
<td>Lin_{36d} = O_{3,3,2,3}(sb_3 \otimes tc_2 \otimes tc_2 \otimes tc_3)</td>
<td>159</td>
<td>0</td>
<td>528</td>
<td>0</td>
<td>405</td>
<td>1092</td>
</tr>
<tr>
<td>Lin_{36e} = O_{3,3,2,3}(sb_3 \otimes tc_2 \otimes tc_3 \otimes tc_2)</td>
<td>189</td>
<td>0</td>
<td>558</td>
<td>0</td>
<td>405</td>
<td>1152</td>
</tr>
<tr>
<td>Lin_{36f} = O_{3,3,2,3}(sb_3 \otimes tc_3 \otimes tc_2 \otimes tc_2)</td>
<td>234</td>
<td>0</td>
<td>603</td>
<td>0</td>
<td>405</td>
<td>1242</td>
</tr>
<tr>
<td>Lin_{36g} = O_{2,3,3,3}(tc_3 \otimes sb_3 \otimes sb_3 \otimes tc_2)</td>
<td>261</td>
<td>0</td>
<td>954</td>
<td>0</td>
<td>729</td>
<td>1944</td>
</tr>
<tr>
<td>Lin_{36h} = O_{2,2,3,3}(tc_3 \otimes sb_3 \otimes tc_2 \otimes sb_3)</td>
<td>297</td>
<td>0</td>
<td>990</td>
<td>0</td>
<td>729</td>
<td>2016</td>
</tr>
<tr>
<td>Lin_{36i} = O_{2,3,3,3}(tc_3 \otimes sb_3 \otimes tc_2 \otimes tc_2)</td>
<td>249</td>
<td>0</td>
<td>618</td>
<td>0</td>
<td>405</td>
<td>1272</td>
</tr>
<tr>
<td>Lin_{36j} = O_{2,3,3,3}(tc_3 \otimes sb_3 \otimes tc_3 \otimes tc_2)</td>
<td>279</td>
<td>0</td>
<td>648</td>
<td>0</td>
<td>405</td>
<td>1332</td>
</tr>
<tr>
<td>Lin_{36k} = O_{2,3,3,3}(tc_3 \otimes sb_3 \otimes sb_3 \otimes sb_3)</td>
<td>405</td>
<td>0</td>
<td>1098</td>
<td>0</td>
<td>729</td>
<td>2232</td>
</tr>
<tr>
<td>Lin_{36l} = O_{2,2,3,3}(tc_3 \otimes tc_3 \otimes sb_3 \otimes sb_3)</td>
<td>309</td>
<td>0</td>
<td>678</td>
<td>0</td>
<td>405</td>
<td>1392</td>
</tr>
<tr>
<td>Lin_{36m} = O_{2,3,3,3}(tc_3 \otimes tc_3 \otimes tc_3 \otimes sb_3)</td>
<td>477</td>
<td>0</td>
<td>846</td>
<td>0</td>
<td>405</td>
<td>1728</td>
</tr>
<tr>
<td>Lin_{36n} = O_{2,2,3,3}(tc_3 \otimes tc_3 \otimes tc_3 \otimes tc_3)</td>
<td>349</td>
<td>0</td>
<td>538</td>
<td>0</td>
<td>225</td>
<td>1112</td>
</tr>
<tr>
<td>Lin_{36o} = O_{2,3,3,3}(tc_3 \otimes tc_3 \otimes sb_3 \otimes tc_2)</td>
<td>531</td>
<td>0</td>
<td>900</td>
<td>0</td>
<td>405</td>
<td>1836</td>
</tr>
<tr>
<td>Lin_{36p} = O_{2,3,3,3}(tc_3 \otimes tc_3 \otimes tc_2 \otimes sb_3)</td>
<td>567</td>
<td>0</td>
<td>936</td>
<td>0</td>
<td>405</td>
<td>1908</td>
</tr>
<tr>
<td>Lin_{36q} = O_{2,3,3,3}(tc_3 \otimes tc_3 \otimes tc_3 \otimes tc_2)</td>
<td>599</td>
<td>0</td>
<td>588</td>
<td>0</td>
<td>225</td>
<td>1212</td>
</tr>
<tr>
<td>Lin_{36r} = O_{2,3,3,3}(tc_3 \otimes tc_3 \otimes tc_3 \otimes tc_3)</td>
<td>429</td>
<td>0</td>
<td>618</td>
<td>0</td>
<td>225</td>
<td>1272</td>
</tr>
<tr>
<td>Lin_{36s} = O_{3,2,3,2}(tc_3 \otimes sb_3 \otimes tc_2 \otimes tc_2)</td>
<td>612</td>
<td>0</td>
<td>981</td>
<td>0</td>
<td>405</td>
<td>1998</td>
</tr>
<tr>
<td>Lin_{36t} = O_{3,2,3,2}(tc_3 \otimes sb_3 \otimes tc_3 \otimes tc_2)</td>
<td>666</td>
<td>0</td>
<td>1035</td>
<td>0</td>
<td>405</td>
<td>2106</td>
</tr>
<tr>
<td>Lin_{36u} = O_{3,2,3,2}(tc_3 \otimes tc_3 \otimes tc_3 \otimes tc_2)</td>
<td>702</td>
<td>0</td>
<td>1071</td>
<td>0</td>
<td>405</td>
<td>2178</td>
</tr>
<tr>
<td>Lin_{36v} = O_{3,2,3,2}(tc_3 \otimes tc_3 \otimes tc_3 \otimes tc_3)</td>
<td>474</td>
<td>0</td>
<td>663</td>
<td>0</td>
<td>225</td>
<td>1362</td>
</tr>
<tr>
<td>Lin_{36w} = O_{3,2,3,2}(tc_3 \otimes tc_3 \otimes tc_3 \otimes tc_2)</td>
<td>504</td>
<td>0</td>
<td>693</td>
<td>0</td>
<td>225</td>
<td>1422</td>
</tr>
<tr>
<td>Lin_{36x} = O_{3,2,3,2}(tc_3 \otimes tc_3 \otimes tc_3 \otimes tc_3)</td>
<td>549</td>
<td>0</td>
<td>738</td>
<td>0</td>
<td>225</td>
<td>1512</td>
</tr>
</tbody>
</table>

5. Implementation of convolution algorithms

In this section, a Maple (Char et al., 1991) package for implementing the algorithms discussed in this paper is described. The codification of convolution in terms of bilinear algorithm built from parameterized matrices allows Maple’s symbolic and algebraic computation facilities to be used to derive and manipulate these algorithms. The infrastructure provided by the package allows for generation, manipulation, testing, and combining various convolution algorithms all within an interactive environment. The
Table 3
Operation counts for cyclic convolution examples of sizes 4, 27, and 108

<table>
<thead>
<tr>
<th>Method</th>
<th>B</th>
<th>B</th>
<th>AT</th>
<th>AT</th>
<th>Muls</th>
<th>Muls</th>
<th>Muls</th>
<th>Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyc4</td>
<td>7</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Cyc27</td>
<td>113</td>
<td>0</td>
<td>252</td>
<td>0</td>
<td>166</td>
<td>531</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgCool4,27</td>
<td>1614</td>
<td>0</td>
<td>2336</td>
<td>0</td>
<td>830</td>
<td>4780</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgCool27,4</td>
<td>754</td>
<td>0</td>
<td>1476</td>
<td>0</td>
<td>830</td>
<td>3060</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SplitNest4,27</td>
<td>780</td>
<td>0</td>
<td>1502</td>
<td>0</td>
<td>830</td>
<td>3112</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SplitNest27,4</td>
<td>702</td>
<td>0</td>
<td>1424</td>
<td>0</td>
<td>830</td>
<td>2956</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ImprovedSplitNest108</td>
<td>672</td>
<td>0</td>
<td>1394</td>
<td>0</td>
<td>830</td>
<td>2896</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

algorithms generated by the package can be exported to a domain-specific language called signal processing language (SPL) and then translated into efficient C or FORTRAN code by the SPL compiler (Xiong et al., 2001). By combining the strengths of Maple and the SPL compiler the benefits of existing algebraic computation tools can be exploited without the need to embed high-performance compiler technology into a computer algebra system.

The resulting environment allows the systematic application of the algebraic theory developed over the years to produce correct and efficient programs. Numerous algorithmic choices can be tried, allowing the user to rapidly test various optimizations to find the best combination of algorithms for a particular size convolution on a particular computer. Furthermore, automatic code generation and algebraic verification provides the ability to construct non-trivial examples with confidence that the resulting code is correct.

5.1. SPL language

This section briefly outlines the SPL language. Further details are available in Xiong et al. (2001), where, in addition, an explanation of how SPL programs are translated to programs is provided.

SPL provides a convenient way of expressing matrix factorizations, and the SPL compiler translates matrix factorizations into efficient programs for applying the matrix expression to an input vector. SPL programs consist of

1. **formulas**, which are symbolic expressions representing matrix factorizations.
2. **constant expressions** for entries appearing in formulas.
3. **define statements** for assigning names to expressions.
4. **compiler directives** for providing type information and controlling compilation.

SPL formulas are built from general matrix constructions, parameterized symbols denoting families of special matrices, and matrix operations such as matrix composition, direct sum, and the tensor product. The elements of a matrix can be real or complex numbers. In SPL, these numbers can be specified as scalar constant expressions, which may contain function invocations and symbolic constants like pi. For example, 12, 1.23, 5*pi, sqrt(5), and
(\cos(2*\pi/3.0), \sin(2*\pi/3)) are valid scalar SPL expressions. All constant scalar expressions are evaluated at compile-time. SPL uses a prefix notation similar to lisp to represent formulas.

To illustrate the syntax and constructs in SPL the factorization for $\mathbb{R}^{3 \times 3}$ in Example 9 is shown in Fig. 1. This uses a define statement to refer to $\mathbb{R}^3$, which is defined using a general matrix construction. The factorization of $\mathbb{R}^{3 \times 3}$ illustrates the matrix operations composition, direct sum, and the tensor product, and the use of a parameterized symbol for the identity matrix. The program also uses directives to indicate that the type is real, the subroutine produced should be called R27, and that the code for R3 should be unrolled, while the code generated for R27 should use loops.

The SPL compiler would produce a subroutine (either C or FORTRAN, depending on the compiler flags used) called R27 which takes an input vector, $x$, of size 27 and produces an output vector, $y$, of size 27 equal to $\mathbb{R}^{3 \times 3}x$. The structure of the program is determined by the matrix factorization. Let $A_3 = R_3 \oplus I_{24}$, $A_2 = (R_3 \otimes I_3) \oplus I_{18}$, and $A_1 = R_3 \otimes I_9$. The code is obtained by looping over code for $R_3$ where each application of $R_3$ is applied to a different segment of size 3 of the input vector $x$. Similar constructions are used for the other factors. The translation process is controlled by a template mechanism which specifies how to construct code for matrix constructions, parameterized matrices, and matrix operations. Various optimizations may be applied, but this simple example illustrates the basic process used by the SPL compiler.

An initial set of templates are provided for common matrices and operations. However, it is possible to define new general matrix constructions, parameterized symbols, and matrix operations by providing additional templates. A useful construction that arises in convolution algorithms that is not available by default is the stack operator. Let $A$ and $B$ be $m \times n$ and $p \times n$ matrices respectively, then (stack $A$ $B$) is the $(m + p) \times n$ matrix

$$
\begin{bmatrix}
A \\
B
\end{bmatrix}
$$

Given a program to apply $A$ to a vector and a program to apply $B$ to a vector, a program to apply (stack $A$ $B$) to a vector is obtained by applying $A$ to the input and storing the result in the first $m$ elements of the output and applying $B$ to the input and storing the result
in the remaining \( p \) elements of the output. The following SPL template enables the SPL compiler to construct code for \((\text{stack } A \ B)\) using this approach.

\[
\text{template (stack any any)}
\begin{align*}
[p1.\text{nx} & = = p2.\text{nx}] \\
& \\
& \text{\$y(0:1:\$p1.\text{ny}_1) = call \$p1( \$x(0:1:\$p1.\text{nx}_1) )} \\
& \text{\$y($p1.\text{ny}_1:1:\$p0.\text{ny}_1) = call \$p2( \$x(0:1:\$p1.\text{nx}_1) )} \\
&
\end{align*}
\]

The first part of the template is the pattern \((\text{stack any any})\) and guard \([p1.\text{nx} \ == \ == p2.\text{nx}]\) which matches \((\text{stack } A \ B)\) where \( A \) and \( B \) match any SPL formulas with equal input dimensions. The code for \( A \) and \( B \) is referenced by \$/B0\p1 and \$/B0\p2, and the fields \( \text{nx} \) and \( \text{ny} \) refer to the input and output dimensions (\(1\) subtracts one). The input and output vectors are implicitly defined as \( x \) and \( y \). The notation \text{base:stride:end} specifies the set of indices ranging from \text{base} through \text{end} in increments of \text{stride}. The call statement expands the code for the matching SPL formulas with given inputs and outputs.

5.2. SPL Maple package

Programming directly in SPL is a cumbersome process. Therefore, an interactive version of SPL is provided in Maple. In this environment, it is much easier to add new features and to extend the language, and it is possible to write simple scripts, using Maple’s algebraic computation engine to generate SPL code. In particular, SPL was extended to include bilinear computations in addition to linear computations, all of the parameterized matrices and bilinear algorithms discussed in this paper were added, and Maple’s polynomial algebra was used to generate SPL objects obtained from the CRT. This section briefly describes the design and implementation of the Maple package. A complete description along with source code is available in Breitzman and Johnson (2002).

The implementation centers around the concept of an SPL object, which corresponds to a multi-linear computation. SPL objects have a name and fields for the number of inputs and the input and output dimensions. In addition, there may be a list of parameters which may be set to Maple expressions such as an integer, list, or polynomial or other SPL objects. Since parameters include both primitive data types and SPL objects, an SPL object can be used to represent general matrix constructions, parameterized matrices, or operators. There are methods to construct an SPL object, evaluate an SPL object to a matrix or a triple of matrices in the case of bilinear algorithms, apply an SPL object, count the number of arithmetic operations used by an SPL object, and export an SPL object. Once exported an SPL object can be compiled by the SPL compiler.

SPL objects can be bound or unbound. An unbound object is a named, parameterized, multi-linear map which does not have a specified method of computation (i.e. it does not have an apply method). Unbound objects can be bound by using a provided bind function. The bind function allows SPL objects to be defined in terms of other SPL objects. Parameterized matrices and operators may be defined using other parameterized matrices.
and operators. Since the SPL objects defining an SPL object may themselves be unbound, bind may need to be applied recursively. It is possible to specify the number of levels that bind is to be applied. Alternatively an unbound object may be bound by using the parameters to index into a hash table containing different computation methods.

Using unbound symbols has several advantages: (1) the size of an SPL expression can be significantly shorter when symbols are not always expanded, (2) it is easier to see the structure in a complicated formula if subformulas are named, (3) parameterized matrices and operators not available to the SPL compiler can be used provided a bind function is available that defines them using formulas supported by the compiler, and (4) an SPL expression can be constructed whose components are unspecified and therefore, alternative computation methods can be used when applying an SPL object. The last point can be used to apply the optimization techniques presented in Section 4 (e.g. the improved split-nesting algorithm).

To illustrate how bind can be used to define an operator using existing operators and parameterized matrices, the stack operator discussed in the previous section will be defined. In this case the operator is extended to take an arbitrary number of operands. Let $A_i, i = 1, \ldots, t$ be an $m_i \times n$ matrix, and observe that

\[(\text{stack } A_1 \ldots A_t) = \begin{bmatrix} A_1 \\ \vdots \\ A_t \end{bmatrix} = \begin{bmatrix} A_1 & \cdots & A_t \end{bmatrix} (e_t^T \otimes I_n),\]

where $e_t$ is the $1 \times t$ matrix containing all ones. The SPL operator stackMatrix in Fig. 2 uses this idea. The methods and fields of an SPL object that do not depend on a particular instance of the object are stored in a symbol table, $\text{symTabl}$, indexed by the object’s name in order to save space. Default print methods are available. The constructor, called SPLStackMatrix, creates a Maple table to store the object, fills in the dynamic fields, and does some error checking. The bind function bindStack uses the function stackk to construct the SPL formula described above. It uses the parameterized matrix $(\text{SPLOnes } m \times n)$, which corresponds to the $m \times n$ matrix whose elements are all equal to 1. This symbol can be defined using $\text{SPLMatrix}([\text{seq}([\text{seq}(1,j=1..n)],i=1..m)])$.

5.3. Convolution package

The convolution package is a collection of parameterized matrices, operators, and bilinear algorithms corresponding to the algorithms presented in Section 3. There are symbols for the parameterized matrices needed in the different algorithms: Vandermonde matrix (Definition 3.2), Overlap matrix (Theorem 3.2), reduction modulo a polynomial, polynomial version of the CRT (Theorem 2.3), DFT matrix (Section 2.4), $R_{p,k}$ and $G_n$ matrices along with their inverses (Theorems 3.9 and 3.10), CRT permutation (Theorem 3.11). There is an operator for creating general bilinear algorithms. There are symbols for linear convolution using the standard and Toom–Cook algorithms, and an operator for combining linear convolution using Theorem 3.2. There are symbols for cyclic convolution using the standard algorithm, Toom–Cook, and the convolution theorem. There are operators for combining cyclic convolutions using the Agarwal–Cooley
Fig. 2. Maple implementation of SPLStackMatrix.

construction in Theorem 3.11 and for combining bilinear algorithms using Winograd’s construction based on the CRT. There are symbols for special cases of Winograd’s construction using Toom–Cook or the standard algorithm for computation modulo the factors of $x^N - 1$, and there is a special symbol for the prime power structure introduced by Selesnick and Burrus. There are generic symbols for linear and cyclic convolution
that are defined using a hash table of algorithms. Default values are placed in the hash table as needed; however, if more efficient algorithms are found, they can be placed in the hash table instead. Finally, there are operators corresponding to matrix exchange (Theorem 2.4) and for evaluating a bilinear algorithm at one input to obtain a linear algorithm.

These symbols and operators can be defined directly from the definitions given in the various theorems using bind functions that construct the corresponding parameterized SPL formulas. Figs. 3 and 4 show how this is done for several sample parameterized matrices and operators respectively. Fig. 5 shows how to use some of the algorithms discussed above, to create SPL code within the Maple package. The SPL code can then be compiled via the SPL compiler available at http://www.spiral.net.

6. Conclusion

In this paper a survey of algorithms for linear and cyclic convolution based on the ideas of Winograd and others was presented. All of the algorithms are expressed in a precise algebraic notation consisting of parameterized matrices, matrix operators, and bilinear algorithms. This allows different algorithms to be easily combined and implementations to be automatically generated directly from the mathematical description. In this framework, optimizing the implementation becomes a search problem over a well-defined space of mathematical formulas. A preliminary investigation has found improved algorithms with
reduced operation counts. In future work, a more systematic search for optimal algorithms will be performed using both operation counts and actual run-times.

References


Rader, C.M., 1968. Discrete Fourier transform when the number of data samples is prime. Proc. IEEE 56 (6), 1107–1108.


