Synthesizing Efficient Out-of-Core Programs for Block Recursive Algorithms using Block-Cyclic Data Distributions†

Zhiyong Li  John H. Reif  Sandeep K.S. Gupta
Department of Computer Science,  Department of Computer Science,
Duke University,  Colorado State University,
Durham, N.C.  27708-0129  Ft. Collins, CO 80523-1873

Abstract

In this paper, we present a framework for synthesizing I/O efficient out-of-core programs for block recursive algorithms, such as the fast Fourier transform (FFT) and block matrix transposition algorithms. Our framework uses an algebraic representation which is based on tensor products and other matrix operations. The programs are optimized for the striped Vitter and Shriver’s two-level memory model in which data can be distributed using various cyclic(B) distributions in contrast to the normally used physical track distribution cyclic(Bd), where Bd is the physical disk block size.

We first introduce tensor bases to capture the semantics of block-cyclic data distributions of out-of-core data and also data access patterns to out-of-core data. We then present program generation techniques for tensor products and matrix transposition. We accurately represent the number of parallel I/O operations required for the synthesized programs for tensor products and matrix transposition as a function of tensor bases and data distributions. We introduce an algorithm to determine the data distribution which optimizes the performance of the synthesized programs. Further, we formalize the procedure of synthesizing efficient out-of-core programs for tensor product formulas with various block-cyclic distributions as a dynamic programming problem.

We demonstrate the effectiveness of our approach through several examples. We show that the choice of an appropriate data distribution can reduce the number of passes to access out-of-core data by as large as eight times for a tensor product, and the dynamic programming approach can largely reduce the number of passes to access out-of-core data for the overall tensor product formulas.

Keywords: Parallel I/O, program synthesis, data distribution, tensor product, block recursive algorithm, fast Fourier transform.

1 Introduction

Due to the rapid increase in the performance of processors and communication networks in the last two decades, the cost of memory access has become the main bottleneck in achieving high-performance for many applications. Modern computers, including parallel computers, use a sophisticated memory hierarchy consisting of, for example, caches, main memory, and disk arrays, to narrow the gap between the processor and memory system performance. However, the efficient use of this deep memory hierarchy is becoming more and more challenging. For out-of-core applications, such as computational fluid dynamics and seismic data


1Surface mail: Department of Computer Science, Duke University, Box 90129, Durham, N.C. 27708-0129. E-mail: {zhiyong, reif}@cs.duke.edu, gupta@ca.colorado.edu; Telephone: (919)660-6530 (O), (919)383-4235 (H).
processing, which involve a large volume of data, the task of efficiently using the I/O subsystem becomes extremely important. This has spurred a large interest in various aspects of out-of-core applications, including language support, out-of-core compilers, parallel file systems, out-of-core algorithms, and out-of-core program synthesis [2, 23, 7, 4].

Program synthesis (or automatic program generation) has a long history in computer science [19]. In the recent past, tensor (Kronecker) product algebra has been successfully used to synthesize programs for the class of block recursive algorithms for various architectures such as vector, shared memory and distributed memory machines [13, 10, 5], and for memory hierarchies such as cache and single disk systems [17, 16]. We have recently enhanced this program synthesis framework for multiple disk systems with the fixed physical track data distribution [11] as captured by the two-level disk model proposed by Vitter and Shriver [24].

In this paper, we present a framework of using tensor products to synthesize programs for block recursive algorithms for the striped Vitter and Shriver’s two-level memory model which permits various block-cyclic distributions of the out-of-core data on the disk array. The framework presented in this paper generalizes the framework presented in [11]. We use the algebraic properties of the tensor products to capture the semantics of block-cyclic data distributions cyclic(B), where B is the logical block size, on the disk array. We formalize the procedure of synthesizing efficient out-of-core programs for the tensor product formulas with various data distributions as a dynamic programming problem. We illustrate the effectiveness of this dynamic programming approach through an example out-of-core FFT program. We further investigate the implications of various block-cyclic distributions cyclic(B) on the performance of out-of-core block recursive algorithms, such as the fast Fourier transform (FFT) and block matrix transposition algorithm [22, 6, 15]. The performance results show that:

1. The choice of data distribution has a large influence on the performance of the synthesized programs,

2. Our simple algorithm for selecting the appropriate data distribution size is very effective, and

3. The dynamic programming approach can always reduce the number of passes to access out-of-core data.

The paper is organized as follows. Section 2 introduce tensor products and discusses formulation of block recursive algorithms using tensor products and other matrix operations. In Section 3, we introduce a two-level computation model and present the semantics of data distributions and data access patterns. Section 4 presents an overview of our out-of-core program synthesis framework. In Section 5 and Section 6, we summarize the performance results and show the effectiveness of using various block-cyclic data distributions. Performance results are presented in Section 8. Finally, conclusions are provided in Section 7.

2 An Overview of the Tensor Product

In this section, we illustrate the formulation of block recursive algorithms using tensor products. We begin with some preliminary definitions which are essential for understanding the rest of the paper.
2.1 Preliminaries

The tensor product is useful in expressing the block structure in a matrix. Let $A$ be an $m \times n$ matrix and $B$ be a $p \times q$ matrix. The tensor product $A \otimes B$ is a block matrix obtained by replacing each element $a_{i,j}$ by the matrix $a_{i,j}B$, i.e.,

$$A^{m,n} \otimes B^{p,q} = \begin{bmatrix} a_{0,0}B^{p,q} & \cdots & a_{0,n-1}B^{p,q} \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B^{p,q} & \cdots & a_{m-1,n-1}B^{p,q} \end{bmatrix}.$$ 

The above tensor product can be factorized as follows:

$$A^{m,n} \otimes B^{p,q} = (A^{m,n} \otimes I_p) (I_n \otimes B^{p,q}) = (I_m \otimes B^{p,q}) (A^{m,n} \otimes I_q),$$

where $I_n$ represents the $n \times n$ identity matrix. A tensor factorization can be used to efficiently compute $Y^{mp}$ obtained by applying $C^{mp,nq} = (A^{m,n} \otimes B^{p,q})$ to vector $X^{nq}$, i.e., $Y^{mp} = C^{mp,nq}(X^{nq})$. For example, direct application of $C^{mp,nq}$ to $X^{nq}$ requires $O(mpqn)$ scalar operations. However, the following algorithm based on the tensor factorization of $C^{mp,nq}$: $Z^{mq} = (A^{m,n} \otimes I_q)(X^{nq})$; $Y^{mp} = (I_m \otimes B^{p,q})(Z^{mq})$, requires only $O(qmn + mpq)$ scalar operations.

A tensor product involving an identity matrix can be implemented as parallel operations. For example, consider the application of $(I_m \otimes A^{p,n})$ to $X^{mn}$, i.e.,

$$\begin{bmatrix} A^{p,n} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & A^{p,n} \end{bmatrix} \begin{bmatrix} X^{mn}(0:n-1) \\ X^{mn}(n:2n-1) \\ \vdots \\ X^{mn}((m-1)n:mn-1) \end{bmatrix} = \begin{bmatrix} A^{p,n}X^{mn}(0:n-1) \\ A^{p,n}X^{mn}(n:2n-1) \\ \vdots \\ A^{p,n}X^{mn}((m-1)n:mn-1) \end{bmatrix}.$$ 

This can be interpreted as $m$ copies of $A^{p,n}$ acting in parallel on $m$ disjoint segments of $X^{mn}$. However, to interpret the application $(A^{p,n} \otimes I_m)$ to $X^{mn}$ as parallel operations we need to understand stride permutations (a.k.a. shuffle permutations).

The stride permutation $L^m_n$ of a vector $X^{mn}$ is a vector $Y^{mn}$, where $Y^{mn} = [X^{mn}(0:mn-1:n);X^{mn}(1:mn-1:n);\ldots;X^{mn}(m-1:mn-1:n)]$, i.e., the first $m$ elements of $Y^{mn}$ are $X^{mn}(0:mn-1:n)$, which represents elements of $X^{mn}$ at stride $n$ starting with element 0. The next $m$ elements are elements of $X^{mn}$ at stride $n$, starting with element 1. The stride permutation $L^m_n$ can be represented as an $mn \times mn$ transformation. For example, the effect of applying $L^6_5$ to $X^6$ can be expressed in matrix form as follows:

$$L^6_5X^6 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_1 \\ x_3 \\ x_5 \end{bmatrix}.$$ 

Stride permutations can also be defined in terms of a permutation of the tensor product of vector bases. A vector basis $e^m_i$, $0 \leq i < m$, is a column vector of length $m$ with a one at position $i$ and zeros elsewhere. The tensor product of vector bases is called a tensor basis. A tensor basis $e_i^m \otimes \cdots \otimes e_i^m$ can be linearized into a vector basis $e^{m_1 \cdots m_i}_{m_1 + \cdots + i-1 m_1 + i}$. Equivalently, a vector basis $e_i^M$ can be factorized into a tensor product of
vector bases \( e_i^m \otimes \cdots \otimes e_i^m \), where \( M = m_1 \cdots m_t \) and \( i_k = (i \text{ div } M_{k+1}) \text{ mod } M_k \), \( M_k = \prod_{i=1}^t m_i \), \( M_{t+1} = 1 \). For example, \( e_k^{12} = e_1^2 \otimes e_1^2 \otimes e_5^2 \). The stride permutation \( L_m^{m,n} \) can now be defined as:

\[
L_m^{m,n} (e_i^m \otimes e_j^n) = e_j^n \otimes e_i^m.
\]

This gives the relationship between the indexing of the input and the output vectors. By linearizing the input tensor basis \( e_i^m \otimes e_j^n \) to \( e_{m+j} \), we get the indexing function of the input vector to be \( m+n \). Similarly, the indexing function of the output vector is obtained by linearizing the output tensor basis to be \( jm+i \). Therefore, the effect of applying the stride permutation \( L_m^{m,n} \) to an input vector is that the element at index \( m+n \) of the input vector is stored in location at index \( jm+i \) of the output vector.

Using stride permutations, an application of \( A^{p,n} \otimes I_m \) to \( X^{m,n} \) can also be interpreted as \( m \) parallel applications of \( A^{p,n} \) to disjoint segments of \( X^{m,n} \) by using the identity \( L_m^{m,n} (A^{p,n} \otimes I_m) = (I_m \otimes A^{p,n}) L_m^{m,n} \) as follows:

\[
L_m^{m,n} (Y^{pm}) = (I_m \otimes A^{p,n})(L_m^{m,n}(X^{m,n})), \text{ i.e.,}
\]

\[
\begin{bmatrix}
Y^{pm}(0: mn - 1: m) \\
Y^{pm}(1: mn - 1: m) \\
\vdots \\
Y^{pm}(m - 1: mn - 1: m)
\end{bmatrix} =
\begin{bmatrix}
A^{p,n}X^{m,n}(0: mn - 1: m) \\
A^{p,n}X^{m,n}(1: mn - 1: m) \\
\vdots \\
A^{p,n}X^{m,n}(m - 1: mn - 1: m)
\end{bmatrix}.
\]

However, the inputs for each application of \( A^{p,n} \) are accessed at a stride of \( m \) and the outputs are also stored at a stride of \( m \).

The properties of tensor products can be used to transform the tensor product representation of an algorithm into another equivalent form, which can take the advantage of the parallel operations discussed above. For example, by using the following tensor product factorizations,

\[
A^{m,n} \otimes B^{p,q} = (A^{m,n} \otimes I_p)(I_n \otimes B^{p,q}) = (I_m \otimes B^{p,q})(A^{m,n} \otimes I_q);
\]

(1)

\( A \otimes B \) can be implemented by first applying \( q \) parallel applications of \( A \) and then \( m \) parallel applications of \( B \). Several other key properties of tensor products are listed below [13]:

1. \( A \otimes B \otimes C = A \otimes (B \otimes C) = (A \otimes B) \otimes C \);
2. \( (A \otimes B)(C \otimes D) = AC \otimes BD \); assume that the ordinary multiplications \( AC \) and \( BD \) are defined.
3. \( \prod_{i=0}^{n-1}(I_m \otimes A_i) = I_m \otimes (\prod_{i=1}^{n-1} A_i) \);
4. \( \prod_{i=0}^{n-1}(A_i \otimes I_m) = (\prod_{i=1}^{n-1} A_i) \otimes I_m \).

Property 2 is also called factor grouping. It transforms the multiplication of two tensor products into one tensor product by first multiplying the matrices \( A \) with \( C \) and \( B \) with \( D \), respectively.

### 2.2 Tensor Product Formulation of Block Recursive Algorithms

A block recursive algorithm is obtained from a recursive tensor factorization of a computation matrix. For example, FFT algorithms are derived by tensor factorization of the discrete Fourier transform (DFT) matrix.

---

1. We ignore the dimensions of matrices whenever they are clear from the context.
The algorithms obtained from tensor factorization are computationally more efficient than those that directly use the unfactorized matrix. For example, computing the DFT of a vector of size \( N \) by directly multiplying it by an \( N \times N \) DFT matrix requires \( O(N^2) \) operations compared to only \( O(N \log(N)) \) operations using an FFT algorithm. Some other examples of block recursive algorithms are Strassen’s matrix multiplication [12, 14], convolution [9], and fast sine/cosine transforms [18].

A tensor product formulation of a block recursive algorithm has the following generic form:

\[
\prod_{j=1}^{k} (I_{r_j} \otimes A_{v_j} \otimes I_{c_j}),
\]

where \( A_{v_j} \) is a \( v_j \times v_j \) square linear transformation, \( \prod_{i=1}^{k} F_i \) denotes \( F_k \cdots F_1 \), and \( r_j v_j c_j = r_iv_i c_i \), for \( 1 \leq i, j \leq k \). The computation performed at each step \( j \) is \( U_j = (I_{r_j} \otimes A_{v_j} \otimes I_{c_j})(V_j) \). Due to presence of identity terms, it is easy to express each computation step using parallel operations. However, the task of harnessing this inherent parallelism in each computation step with the goal of minimizing the parallel I/O operations is non-trivial. We next present tensor product formulations of two FFT algorithms which are used as examples in this paper.

**Fast Fourier Transform**

The tensor product formulations of various FFT algorithms are presented in [13, 18]. These formulations are obtained by different tensor factorizations of the discrete Fourier transform matrix. Although all of these algorithms are computationally equivalent, they have different computational structures and different data access patterns. For example, consider the following tensor product formulation of the radix-2 decimation-in-time Cooley-Tukey FFT:

\[
F_{2^n} = \left( \prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(I_{2^{n-i}} \otimes T_{2^{i-1}}^{2^n}) \right) R_{2^n},
\]

\[
R_{2^n} = \prod_{i=1}^{n} (I_{2^{i-1}} \otimes L_2^{2^{n-i+1}}), \text{ and } F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.
\]

\( T_{2^{i-1}} \) represents a diagonal matrix of constants and \( R_{2^n} \) permutes the input sequence to a bit-reversed order. As can be seen from Eq. (3), for an FFT on \( 2^n \) points, there are \( n \) steps in the computation after performing the initial bit-reversal permutation. At each step, the data array from the previous step is scaled by multiplying by twiddle factors \( Y = (I_{2^{n-i}} \otimes T_{2^{i-1}}^{2^n})(X_{i-1}) \), followed by the butterfly computation \( X_i = (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(Y) \).

**Matrix Transposition**

The transposition of a \( p \times q \) matrix \( M^{pq} \) can be expressed using a stride permutation \( L_p^q \) as \( (M^{pq})^T = L_q^p(M^{pq}) \), where \( M^{pq} \) is the row-major linear representation of \( M^{pq} \). Various matrix transposition algorithms can be expressed using tensor product formulas involving stride permutations [11]. For example, the block matrix transposition algorithm for transposing a \( p \times q \) matrix can be described by the following formula:

\[
L_q^p = (I_{q_2} \otimes L_p^{q_2} \otimes I_{p_2})(L_{q_2}^{q_2} \otimes I_{p_2 q_1}) (I_{p_2 q_1} \otimes L_p^{q_1 q_2}) (I_{p_2 q_2} \otimes L_p^{q_1 q_2} \otimes I_{q_1}),
\]
where \( p = p_2 p_1 \) and \( q = q_2 q_1 \). The first (rightmost) factor converts the row-major representation of the input matrix to a row-major representation of the input matrix viewed as a \( p_2 \times q_2 \) block matrix consisting of \( p_1 \times q_1 \) size blocks. The second and third factor express transposition of each block and transposition of the block matrix, respectively. The fourth factor is the inverse of the first and it reverts the block row-major representation to row-major representation of the output. The correctness of this representation can be seen by applying the factors to the input basis \( \beta_s \equiv e_{i_2}^{p_2} \odot e_{j_1}^{q_1} \odot e_{i_1}^{p_1} \) to get the following sequence of bases,

\[
\beta_s \rightarrow e_{i_2}^{p_2} \odot e_{j_1}^{q_1} \odot e_{i_1}^{p_1} \rightarrow e_{i_1}^{p_1} \odot e_{j_1}^{q_1} \odot e_{i_2}^{p_2} \rightarrow e_{j_1}^{q_1} \odot e_{i_1}^{p_1} \odot e_{i_2}^{p_2} \rightarrow e_{j_1}^{q_1} \odot e_{i_2}^{p_2} \odot e_{i_1}^{p_1} = \beta_t,
\]

and noting that \( \beta_t = L_{q}^{p}(\beta_s) \). Note that we have used the identity

\[
(A^{m,n} \otimes B^{p,q})(e_i^{n} \otimes e_j^{q}) = A^{m,n}(e_i^{n}) \otimes B^{p,q}(e_j^{q}).
\]

The basis \( \beta_t \) is called the output basis.

### 3 Parallel I/O Model with Block-Cyclic Data Distributions

We use a two-level model which is similar to Vitter and Shriver’s two-level memory model [24]. However, in our model the data on disks (called out-of-core data) can be distributed in different (logical) block sizes. The model consists of a processor with an internal random access memory and a set of disks. The storage capacity of each disk is assumed to be infinite. On each disk, the data is organized as physical block with fixed size. Four parameters: \( N \) (the size of the input), \( M \) (the size of the internal memory), \( B_d \) (the size of each physical block), and \( D \) (the number of disks), are used in this model. We assume that \( M < N, 1 \leq B_d \leq \frac{M}{T}, \) and \( 1 \leq D \leq \frac{M}{B_d} \).

In this model, disk I/O occurs in physical tracks (defined below) of size \( B_d D \). The physical blocks which have the same relative positions on each disk constitute a physical track. The physical tracks are numbered contiguously with the outermost track having the lowest address and the innermost track having the highest address. The \( i \)th physical track is denoted by \( T_i \). Fig. 1 shows an example data layout with \( B_d = 4, D = 4, \) and \( N = 64 \). Each parallel I/O operation can simultaneously access \( D \) physical blocks, one block from each disk. Therefore parallelism in data access is at two levels: elements in one physical block are transferred.
concurrently and $D$ physical blocks can be transferred in one I/O operation. In this paper, we use the *striped disk* access model in which physical blocks in one I/O operation come from the same track, as opposed to the *independent I/O* model in which block can come from different tracks. We use the parallel primitives, \texttt{parallel\_read}(i) and \texttt{parallel\_write}(i), to denote the read and write to the physical track $T_i$, respectively. We define the measure of I/O performance as the number of parallel I/Os required.

### 3.1 Block-Cyclic Data Distributions

Block-cyclic distributions have been used for distributing arrays among processors on a multiprocessor system. A block-cyclic distribution partitions an array into equal sized block of consecutive elements and then maps them onto the processors in a cyclic manner. If we regard the disks in the above model as processors, then the data organization described above (e.g. in Fig. 1) is exactly a block-cyclic distribution (denoted as \texttt{cyclic}(B_d)) with the block size $B_d$.

Moreover, we can assume that data can be distributed with an arbitrary block size. \textsuperscript{2} Fig. 2 shows the data organization for the same parameters as in Fig. 1, but with a \texttt{cyclic}(8) distribution. Notice that the size of the physical track and the size of the physical block are not changed. However, they contain different records. We will call $B$ records in a block formed by a \texttt{cyclic}(B) distribution as a logical block. Similarly, the logical blocks which have the same relative positions on each disk consist of a logical track. The $i$th logical track is denoted as $LT_i$. Note that each parallel I/O operation still accesses a physical track not a logical track. Hence, several parallel I/O operations are needed to access a logical track. For example, to load the logical track $LT_1$ in Fig. 2, two \texttt{parallel\_read} operations \texttt{parallel\_read}(2) and \texttt{parallel\_read}(3), which respectively load the physical tracks $T_2$ and $T_3$, are needed. We next use a simple example to show the advantages of using logical distributions on developing I/O-efficient programs for block recursive algorithms.

### Why Logical Data Distributions?

Assume that we want to implement $F_8 \otimes I_8$ on our target model under the parameters given in Fig. 1. Further, we assume that the size of the main memory is the half of the size of the inputs. Because we are mainly interested in data access patterns, we ignore the real computations conducted by $F_8$. The only thing we need to remember is that $F_8$ needs eight elements with a stride of eight because of the existence of the identity matrix $I_8$.

\textsuperscript{2} Cormen has called this data organization on disks as a banded data layout [3] and studied the performance for a class of permutations and several other basic primitives of NESL language[1].

---

**Figure 2:** The data organization for $N = 64$ inputs with $B_d = 4$, $D = 4$, and $B = 8$. Each column is a disk. The first left shadowed box denotes an example logical block. There are two logical tracks $LT_0$ and $LT_1$ each of them consists of two physical tracks.
We first consider implementing $F_8 \otimes I_8$ on the physical block distribution. From the above discussion, we know that the first $F_8$ needs to be applied to eight elements: 0, 8, 16, 24, 28, 32, 40, 48, and 56. From Fig. 1, we can see that these elements required by the $F_8$ computation are stored on four physical tracks. However, our main memory can hold only two physical tracks, so that we can not simply load all of the four physical tracks into the main memory and accomplish the computation in one pass of I/O. To get around this memory limitation, we can use two different approaches.

First, we load the first physical track and keep the first half of the records in each physical block in that loaded physical track and throw other half of the records. We do this for every other physical track. Then we do the computation for half of the records in the main memory. After finishing computation for half of the records, we write the results out. Then we repeat the above procedure. However, we now keep other half of the records in the main memory for each loaded track. By doing computation in this way, it is obvious that we need two passes to load out-of-core data.

Another method is to use a logical block distribution. Let the size of a logical block be eight as shown in Fig. 2. In this case, the eight records required by one $F_8$ are stored on two physical tracks, either $T_1$ and $T_3$, or $T_2$ and $T_4$. Therefore, if we can first load and perform computation on $T_1$ and $T_3$, followed by loading and performing computation on $T_2$ and $T_4$, then the entire computation can be performed in a single pass. Hence logical distribution can be used to reduce the number of passes needed to perform the entire computation. However, there are several issues which need to be addressed, such as how to determine the block size of the logical distribution and how to determine the data access patterns. We will discuss these issues in the following sections. For simplicity, we make the following assumptions. The input and the output data are stored in separate set of disks. All parameters are power of two $^3$. The block size $B$ of the distribution is a multiple of $B_d$.

### 3.2 Semantics of Data Distributions and Access Patterns

A block-cyclic distribution can be algebraically represented by a tensor basis by identifying the bases which correspond to processor index [10]. This approach can be adapted to represent data distributions onto disks in our parallel I/O model by substituting disks for processors. However, due to existence of physical blocks and physical tracks, the method of using tensor bases to define a block-cyclic distribution for multiprocessors needs to be generalized. This we achieve by further factoring the tensor basis to get bases for physical block index and physical track index. We call this factored tensor basis an (out-of-core) data distribution basis, which is defined as follows:

**Definition 3.1** Let $B = B_B B_d$. If a vector of length $N$, where $N = GBD$ and $G$ is an integer, is distributed according to the cyclic($B$) distribution on $D$ disks, then its data distribution basis is defined as:

$$D = e_G^D \otimes e_b^B \otimes e_b^{B_d}.$$  \hspace{1cm} (5)

We use $D(s)$ to refer to the $s$th factor (from the left), e.g., $D(2) = e_d^D$.

$^3$The results can be easily generalized to all parameters to be power of any integer.
For example, the data distribution basis for Figure 2 is $e_g \otimes e_d^4 \otimes e_{b_b}^2 \otimes e_{b_d}^4$, where the size of each physical block is four; each logical block contains two physical blocks; there are four disks; and the inputs are stored on two logical tracks. The data distribution basis for Figure 1 can be written as $e_d^4 \otimes e_{b_d}^4$, where $B_d = 1$.

A selected portion of the distribution basis in Formula (5) can be used to obtain the indexing function needed to denote a particular data unit such as a logical track or a physical track. Let,

$$\text{logical-track}(D) = e_g^G$$  \hspace{1cm} (6)
$$\text{physical-track}(D) = e_g^G \otimes e_{b_b}^D$$  \hspace{1cm} (7)

Then the indexing function for accessing the physical tracks can be obtained by linearizing $\text{physical-track}(D)$. Similarly, we can have tensor bases which denote the records inside a logical track and a physical track, respectively. These tensor bases are called the logical track-element basis ($e_d^D \otimes e_{b_b}^B \otimes e_{b_d}^B$) and the physical track-element basis ($e_d^D \otimes e_{b_d}^B$), respectively. An interesting point to note is that, the logical track-element basis can be obtained by deleting the bases corresponding to the logical track index from the data distribution basis $D$. Similarly, the physical-track element basis can be obtained by deleting the bases corresponding to the physical track index from the data distribution basis $D$. Formally,

$$\text{logical-track-element}(D) = D - \text{logical-track}(D),$$  
and

$$\text{physical-track element basis}(D) = D - \text{physical-track}(D)$$

where the basis difference operator, denoted as $-$, is defined as.

**Definition 3.2** Let $S$ and $G$ be two tensor bases. Their difference is denoted as $S - G$ and is a tensor basis which is constructed by deleting all of the vector bases in $G$ from $S$.

### 3.3 Tensor Bases for Data Access

For fixed input and output data distribution bases, different orders of instantiating the indices in the indexing function of the data distribution bases (as defined in Formula (5)) correspond to different access patterns for out-of-core data. For example, if we instantiate the indices in the order from right to left (which is what we have used to interpret a tensor basis so far), i.e. $g$ is the slowest and $b_d$ is the fastest changing indices, then we actually access data first in the first logical block of the first disk and then access the first logical block in the second disk. After finishing the access to the first logical track sequentially, the second logical track is accessed, and so on. This data access pattern can be better understood by examining the following code, which uses the indices in each vector basis as loop index variable.

```
DO g = 0, G - 1
  DO d = 0, D
    DO b_b = 0, B_b - 1
      DO b_d = 0, B_d - 1
        read(gB_bDB_d + dB_bB_d + b_bB_d + b_d)
      ENDDO
    ENDDO
  ENDDO
ENDO ENDDO ENDDO ENDDO
```

If we instantiate the index $b_b$ in $e_{b_b}^D$ after the index $d$ in $e_d^D$ in Formula (5), then it results in an access pattern where first the data along a physical track is accessed and then the successive physical tracks are
accessed. This change in the instantiation order of the indices can be regarded as a permutation \(^4\) of the data distribution basis. We call a permutation of a data distribution basis as a loop basis. For the above example, the loop basis can be denoted as,

\[ L = e_g^G \otimes e_{b_h}^R \otimes e_{d}^D \otimes e_{b_d}^D \]  

(8)

Together, a data distribution bases and a loop bases specify a data access pattern. To synthesize a program with this data access pattern, every index in a loop basis may correspond to a loop in the generated loop nest. Moreover, the order of the loops in the loop nest is determined by the order of the vector bases in the loop basis. A program which can access out-of-core data specified by the loop basis denoted by Formula (8) is shown in below.

DO \( g = 0, G - 1 \)
DO \( b_h = 0, B_h - 1 \)
DO \( d = 0, D \)
DO \( b_d = 0, B_d - 1 \)
read(\( gB_hDB_d + dB_hB_d + b_hB_d + b_d \))
ENDDO ENDDO ENDDO ENDDO

Note that in the above program, the indexing function for accessing each record is obtained by linearizing the data distribution basis. The order of loops is specified by the loop basis. In terms of programs, a loop basis can be understood as a notation specifying how to re-order the loop nests and further how to split a loop nest \([25]\).

4 Synthesizing I/O-Efficient Programs

In this section, we first give an overview of our program synthesis framework. We then describe the structure of the generated program and how the program can be obtained from an augmented tensor basis. In the following section we describe how to compute the augmented tensor basis to obtain the desired program structure.

4.1 Overview of Program Synthesis

The three major steps in synthesizing efficient parallel I/O programs for a block recursive algorithm are shown in Fig. 3. The first step transforms the input tensor product formula into an efficient form. It uses

\(^4\)Let \( S \) be a tensor basis and \( S = \bigotimes_{i=1}^{q} e_{i_1}^{s_i} \). Let \( \alpha \) be a permutation on \([1 \ldots q]\), then a permutation of \( S \) is a tensor basis defined as follows, \( \alpha(S) = \bigotimes_{i=1}^{q} e_{\alpha(i)}^{s_i} \),
the target machine parameter and properties of tensor products to obtain the efficient form using either a greedy approach or an approach based on dynamic programming. It also determines the appropriate input and output data distributions for implementing the transformed formula. The second and the third steps is applied to each computational step, which is represented by a tensor product. In the final program, an outermost loop structure is used to construct the program for overall tensor product formula. More specifically, the second step decomposes the computation of each tensor product into sub-computations by analyzing data access patterns and exploiting locality and concurrency. The results of these analyses are represented as an augmented tensor basis. The augmented tensor basis consists of the following four components: data distribution bases, loop bases, sub-computations and memory-loads. These four components are then used by the third step of the code generation algorithm to generate parallel I/O programs.

Our presentation of the derivation of efficient implementations for the block recursive algorithms is in the reverse order of Fig. 3. We first present a procedure for code generation by using the information contained in the augmented tensor basis. Then we determine efficient implementations for a stride permutation and a simple tensor product with a given data distribution on a given model by determining the corresponding augmented tensor bases. Further, we develop a simple algorithm to determine the data distribution which can result in an efficient implementation. Furthermore, we use a dynamic (or a multi-step dynamic) programming algorithm to determine an efficient implementation for the block recursive algorithms. The dynamic programming algorithm will use the properties of tensor products and the performance of each tensor product. The method of estimating the performance for each tensor product will be presented in Section 5.2 and Section 5.3 with the analysis of the second step (determining augmented tensor bases).

4.2 Structure of the Generated Parallel I/O Code

To minimize the number of I/O operations for a synthesized program for a tensor product, we need to exploit locality by reusing the loaded data. This requires decomposing the computation and reorganizing data and data access patterns to maximize data reuse. In the synthesized program, the same sub-computation is performed several times over different data sets. Hence, the loop structure of the synthesized program is constructed as follows. An outer loop nest enclosing three inner loop nests: read loop nest, computation loop nest, and write loop nest. The read loop nest loads out-of-core data without overflowing main memory. The computation loop nest performs sub-computation on a memory-load. And the write loop nest writes the output to the disk. The data sets are accessed one track at a time using parallel primitives, parallel_read and parallel_write.

To reflect the structure of the outer and inner loops described above, we need to separate input loop bases λ into three parts: a) the part specifying memory-loads (λn), b) the part specifying the physical tracks in a memory load (λm), and c) the part specifying the records within a track (λµ). Under our striped I/O model, each I/O operation reads and writes in terms of physical track each time. Hence in the synthesized program, the loops which correspond to λµ may not appear explicitly. Formally, we can write the input loop basis as follows:

\[ \lambda = \lambda_n \otimes \lambda_m \otimes \lambda_\mu, \]  

(9)

11
1. Generate loops for indices in $\lambda_m$
2. Generate loops for indices in $\lambda_m$
3. *Parallel_read* the physical track whose index is determined using $\text{physical-track}(\beta)$
4. Keep records for current memory-load
5. End the loops corresponding to $\lambda_m$
6. Perform operations to a memory-load
7. Generate loops for indices in $\theta_m$
8. *Parallel_write* a physical track whose index is determined using $\text{physical-track}(\delta)$
9. End the loops corresponding to $\theta_m$
10. End loops corresponding to $\lambda_n$

Figure 4: Procedure of code generation for a tensor product.

where, we call $\lambda_n$ a *memory basis*, since each instantiation of the indices in $\lambda_n$ corresponds to a memory-load. Similarly, we can separate the output loop basis as follows,

$$\theta = \theta_0 \otimes \theta_m \otimes \theta_\mu.$$  \hfill (10)

Moreover, our method of determining loop bases will guarantee that $\theta_n$ is a permutation of $\lambda_n$. Furthermore, in order to have a common outer loop nest $\theta_n = \lambda_n$.

To minimize the parallel I/O operations, it is desirable that the synthesized program makes a single pass over the input data. That is to say each memory-load should have the following *perfect memory-load* property: the input data elements of the memory-load can be organized to form a set of tracks consistent with input data distribution and the output data elements of the memory load can be organized to form a set of tracks consistent with output data distribution. If we can construct perfect memory-loads, then we can synthesize a program which accesses out-of-core data only once (called a *one-pass program*). However, for some computations, it may not be possible to construct perfect memory-loads. For these computations, the synthesized program keeps only part of the records from a loaded physical track in the main memory and discards other records. Therefore, in a *multiple-pass program* the same physical track is loaded several times.

In terms of input and output loop bases, perfect memory-loads can be constructed if $\lambda_\mu$ and $\theta_\mu$ consist of the physical-track-element bases from the input and output data distribution bases, respectively. Hence, initially, we assume that the initial loop bases $\lambda$ and $\theta$ have the properties that $\lambda_\mu$ and $\theta_\mu$ consist of the physical-track-element bases from the input and the output data distribution bases, respectively. If it turns out that a single pass program cannot be synthesized for the computation, then $\lambda_\mu$ (or $\theta_\mu$) is further factorized into two parts, $\lambda_{\mu_1}$ and $\lambda_{\mu_2}$. Further, $\lambda_{\mu_2}$ is moved out of $\lambda_\mu$ and put into $\lambda_n$. This moved tensor basis $\lambda_{\mu_2}$ is used to determine which portions of a physical block should be kept for the current memory-load. The size of this moved vector basis is equal to the number of times the same physical tracks are loaded.

### 4.3 Parallel I/O Code Generation

In this subsection, we first define the augmented tensor basis and then describe the generic code generation routine which uses the augmented tensor basis to generate parallel I/O code.

An augmented tensor basis for a single-processor multi-disk system includes data distribution bases, loop
bases, memory-loads and operations on each memory-load. Moreover, for a tensor product computation, the input and output data may be organized and accessed differently. We therefore need to use input data distribution basis $\beta$, output data distribution basis $\delta$, input loop basis $\lambda$, and output loop basis $\theta$ to denote them respectively.

**Definition 4.3** An augmented tensor basis constitutes the following four components,

1. **Data distribution basis.** Let data be distributed by cyclic($B$) on $D$ disks. Let $B = B_nB_d$ and the number of data elements be $N$, where $N = GBD$. Then the (input or output) data distribution basis has the form:

$$
D = e^G \otimes e^B \otimes e^R \otimes e^R_d .
$$

2. **Loop Basis.** An (input or output) loop basis has the following generic form,

$$
\mathcal{L} = \mathcal{L}_n \otimes \mathcal{L}_m \otimes \mathcal{L}_\mu_1
$$

where,

- $\mathcal{L}_\mu_1$ is a subset of $\mathcal{L}_\mu$, where $\mathcal{L}_\mu = D(2) \otimes D(4)$ and $\mathcal{L}_\mu_1 = \mathcal{L}_\mu - \mathcal{L}_{\mu_2}$;

- $\mathcal{L}_m$ consists of the last portions of $D - \mathcal{L}_\mu_1$ such that $|\mathcal{L}_m| = \frac{M}{\mathcal{L}_\mu_1}$;

- $\mathcal{L}_n = D - \mathcal{L}_m - \mathcal{L}_\mu_1$.

3. **Memory-load.** The records in each memory-load are denoted by $\mathcal{L}_m \otimes \mathcal{L}_\mu_1$. More specifically, each memory-load is obtained by an instantiation of indices in $\mathcal{L}_n$, looping over indices in $\mathcal{L}_m$, and using $\mathcal{L}_{\mu_2}$ to identify which portions in each loaded physical track should be kept for the current memory-load.

4. **Sub-computation.** The decomposed computation which will be applied to each memory-load.

Note that the input and the output data distribution bases can be different. Moreover, the input data distribution basis can be obtained by factoring the input basis. The output data distribution basis can be obtained by applying the corresponding tensor product or stride permutation to the input data distribution basis.

Using this augmented tensor basis and assuming that $\theta_n = \lambda_n$, a generic program can then be obtained as described in Fig. 4. Further, Fig. 5 shows an example synthesized program for $I_4 \otimes F_2 \otimes I_4$. We assume that $M = 16$, $D = 2$, $B_d = 2$, $B = 2$, $F_2$ is a $2 \times 2$ matrix, and data are distributed in a cyclic($2$) manner. It uses $e^8 \otimes e^2_d \otimes e^2_d$ as both the input and the output distribution bases. The input and the output loop bases are also the same as $e^2_{g_2} \otimes e^4_{g_1} \otimes e^2_{g_2} \otimes e^2_{g_1}$, where $e^2_{g_2} \otimes e^4_{g_1}$ is a factorization of $e^8_g$. The sub-computation is denoted by $I_2 \otimes F_2 \otimes I_4$. The memory basis is $e^2_{g_2}$. The details of how to determine this information are discussed in Section 5.3.

---

5 As described in Sec. 4.2, the size of $\mathcal{L}_{\mu_2}$ (i.e. $\lambda_{\mu_2}$ or $\lambda_{\mu_2}$) depends upon the tensor product being implemented. The procedure for determining $\mathcal{L}_{\mu_2}$ is described in the following sections.
DO \( g_2 = 0, 1 \)
DO \( g_1 = 0, 3 \)

// Parallel read from a track
\( X(4g_1 : 4g_1 + 3) \leftarrow parallel\_read(4g_2 + g_1) \)
ENDDO

// Perform operations for a memory-load
\( Code(X(1 : 16) \leftarrow A \times X(1 : 16)) \)
// Write the result back
DO \( g_1 = 0, 3 \)

// Parallel write to a track
parallel\_write(4g_2 + g_1) \leftarrow X(4g_1 : (4g_1 + 3))
ENDDO ENDDO

Figure 5: Code for \( I_4 \otimes F_2 \otimes I_4 \), where \( X \) is an array of size \( M \) and \( A = I_2 \otimes F_2 \otimes I_4 \).

5 Synthesizing Programs for Stride Permutations

In this section, we discuss how to determine an efficient augmented tensor basis for stride permutations using a cyclic\( (B) \) distribution. Our goal is to decompose computations into a sequence of sub-computations performed on perfect memory-loads. In the case that perfect memory-loads cannot be constructed, we minimize the number of times the data is loaded for each memory-load. In doing so, we ensure that each physical track of the output is written out only once. We first develop an approach to determining the input and output loop bases for the given distribution cyclic\( (B) \). Based on these loop bases and data distribution bases, we determine memory-loads and operations on the memory-loads. Following this a program can be synthesized by using the procedure presented in Section 4.3. The cost of the program can also be determined from the loop bases. We summarize our results in the following theorem and then present a constructive proof, which constructs the augmented tensor basis.

**Theorem 5.1** Let \( Y = L^PQ X \), where \( PQ = N \) and \( X \) and \( Y \) are input and output vectors with length \( N \), respectively. Let \( X \) and \( Y \) be distributed according to cyclic\( (B) \) and the data distribution bases be denoted as \( \beta \) and \( \delta \), respectively. Further let \( \lambda_\mu = \beta(2) \otimes \beta(4) \) and \( \theta_\mu = \delta(2) \otimes \delta(4) \). Then a program can be synthesized with \( \frac{N}{\mu D}(1 + \max\{1, \left\lfloor \frac{\lambda_\mu - \theta_\mu}{M} \right\rfloor \}) \) \(^6\) parallel I/O operations for the stride permutation \( Y = L^PQ X \).

Proof: We present an algorithm as shown in Fig. 6 for determining the input and the output loop bases. The algorithm is further explained in Step 1 as shown below. In Step 2 and Step 3, we show how to construct memory-loads and operations for a memory-load. In Step 4, we show that I/O costs can be obtained from this information.

1. **Determine input and output loop bases.** We begin with the following construction for the input and the output loop bases,

\[
\lambda = (\lambda - (\theta_\mu - \lambda_\mu)) \otimes (\theta_\mu - \lambda_\mu) \otimes \lambda_\mu, \tag{13}
\]

\[
\theta = (\theta - (\lambda_\mu - \theta_\mu)) \otimes (\lambda_\mu - \theta_\mu) \otimes \theta_\mu. \tag{14}
\]

\(^6\) The notation \(| S |\) denotes the size of the tensor basis \( S \), which is equal to the multiplication of the dimensions of each vector basis in \( S \).
// Initialization
\[ \lambda = \beta(1) \otimes \beta(3) \otimes \beta(2) \otimes \beta(4) \]
\[ \theta = \delta(1) \otimes \delta(3) \otimes \delta(2) \otimes \delta(4) \]
\[ \lambda_\mu = \beta(2) \otimes \beta(4), \theta_\mu = \delta(2) \otimes \delta(4) \]
\[ \lambda_m = \theta_m - \lambda_\mu, \theta_m = \lambda_m - \theta_\mu \]
\[ \lambda_{\mu_2} = \lambda_\mu - \theta_\mu \]
// One-pass or multiple-pass implementation
if \( |(\theta_\mu - \lambda_\mu) \otimes \lambda_\mu| \leq M \) then
\[ \lambda_n = \lambda - \lambda_m - \lambda_\mu \]
else
\[ \text{Factor}(\lambda_\mu - \theta_\mu) \text{ such that } \theta_m \text{ consists of the last factors of the factored tensor basis and } |\theta_m| = \frac{M}{D}. \]
\[ \lambda_{\mu_2} = (\lambda_\mu - \theta_\mu) - \theta_m, \lambda_{\mu_1} = \lambda_\mu - \lambda_{\mu_2} \]
\[ \lambda_n = \lambda - \lambda_m - \lambda_{\mu_1} \]
// The final input and output loop bases
\[ \theta_n = \lambda_n \]
\[ \lambda = \lambda_n \otimes \lambda_m \otimes \lambda_{\mu_1} \]
\[ \theta = \theta_n \otimes \theta_m \otimes \theta_\mu. \]

Figure 6: Algorithm for determining input and output loop bases for stride permutations.

where we use the convention that \( \lambda \) appearing on the right hand side refers to the original representation, which is equal to \( \beta(1) \otimes \beta(3) \otimes \beta(2) \otimes \beta(4) \), and \( \lambda \) appearing on the left hand side refers to an update. So does \( \theta \). Further, we assume that \( \lambda_\mu = \beta(2) \otimes \beta(4), \theta_\mu = \delta(2) \otimes \delta(4) \). It is easy to verify that \((\theta_\mu - \lambda_\mu) \otimes \lambda_\mu\) is a permutation of \((\lambda_\mu - \theta_\mu) \otimes \theta_\mu\). Therefore, they denote the same records. Thus, if the number of records denoted by \( |(\theta_\mu - \lambda_\mu) \otimes \lambda_\mu| \) is less than the size of the main memory, then we can simply take \( \lambda_m = \theta_\mu - \lambda_\mu \) and \( \theta_m = \lambda_\mu - \theta_\mu \). However, the number of the records denoted by \( |(\theta_\mu - \lambda_\mu) \otimes \lambda_\mu| \) may exceed the size of the main memory. In that case, we want to construct memory-loads which can be obtained by reading the input data several times while writing the output data only once. In terms of tensor bases, as we discussed in Section 4.3, this reloading can be achieved by looping over part of the indices in \( \lambda_\mu \). In other words, we need to factor \( \lambda_\mu \) as \( \lambda_{\mu_2} \) and \( \lambda_{\mu_1} \) such that the instantiation of the indices in \( \lambda_{\mu_2} \) selects which sub-blocks should be kept for a loaded physical track and the instantiation of the indices in \( \lambda_{\mu_1} \) denotes records inside each sub-block. Further, \(|\lambda_{\mu_2}|\) is equal to the number of times we will reload each physical track. This reloading is achieved by taking \( \lambda_m = \theta_\mu - \lambda_\mu \) and moving \( \lambda_{\mu_2} \) before \( \lambda_m \). In summary, the input and output loop bases in Formulas (13) and (14) are modified as follows:

- **Factor** \((\lambda_\mu - \theta_\mu)\) such that \( \theta_m \) consists of the last factors of the factored tensor basis and the size of \( \theta_m \) is equal to \( \frac{M}{D} \).
- **For input loop basis**, let \( \lambda_{\mu_2} = (\lambda_\mu - \theta_\mu) - \theta_m, \lambda_{\mu_1} = \lambda_\mu - \lambda_{\mu_2} \).

Thus, the input and output loop bases can be written as,

\[ \lambda = \lambda_n \otimes \lambda_m \otimes \lambda_{\mu_1}, \]  \hspace{1cm} (15)

\[ \theta = \theta_n \otimes \theta_m \otimes \theta_\mu. \]  \hspace{1cm} (16)
where \( \lambda_n = \lambda - \lambda_m - \lambda_{\mu_1} \) and \( \theta_n = \theta - \theta_m - \theta_{\mu} \). We further verify the following facts.

First, \( \lambda_m \otimes \lambda_{\mu_1} \) and \( \theta_m \otimes \theta_{\mu} \) contain the same vector bases, although in a different order \([\cdot]\). Second, from the previous results, we have that \(| \lambda_m \otimes \lambda_{\mu_1} | = | \theta_m \otimes \theta_{\mu} | = M \). Therefore the records denoted by them can fit into a memory-load. Third, since \(| \lambda_m | > | \theta_m | (= \frac{M}{BD}) \), loading \( | \lambda_m | \) physical tracks will overflow the main memory unless some records are discarded from the loaded tracks. The details for determining which records to be discarded will be discussed in the next step. Fourth, \( \lambda_n \) and \( \theta_n \) contain the same vector bases. We therefore can set \( \theta_n = \lambda_n \), which will only change the order of writing results onto physical tracks.

2. **Determine memory-load.** When \(| (\theta_m - \lambda_{\mu_1}) \otimes \lambda_{\mu_2} | \leq M \), \( \lambda_m = \theta_m = \lambda_\mu - \lambda_{\mu_2} \) and \( \theta_m = \lambda_m = \lambda_\mu - \lambda_{\mu_2} \). Therefore, the records denoted by \( \lambda_m \otimes \lambda_{\mu_1} \) or \( \theta_m \otimes \theta_{\mu_2} \) can be used to form a perfect memory-load. However, when this condition is not satisfied, we need to use Formulas (15) and (16) as the input and output loop bases, respectively. Because \(| \lambda_m \otimes \lambda_{\mu_1} | = | \theta_m \otimes \theta_{\mu_2} | = M \), the size of each memory-load can be set to be equal to the size of the main memory. However, as we mentioned before, we need to discard some records from each loaded track to form the memory-load. This can be done by linearizing \( \lambda_{\mu_2} \). Each instantiation of the indices in \( \lambda_{\mu_2} \) will give a set of sub-blocks in a physical track which should be kept.

3. **Determine operations for a memory-load.** As we mentioned above, for each memory load, the tensor vectors in the input and output loop bases which denote the records inside a memory-load are the same, but in a different order. In other words, one is a permutation of the other. Because the input and output loop bases are permutations of the input and output data distribution bases, we actually permute a memory-load of data each time. Therefore, each in-memory operation is nothing more than a permutation for a subset of data distribution bases denoted by \( \lambda_m \otimes \lambda_{\mu_1} \) and \( \theta_m \otimes \theta_{\mu_2} \). Note that when \( \lambda_{\mu_2} = \phi_\mu \), \( \lambda_{\mu_1} = \lambda_\mu \).

4. **I/O cost of synthesized programs.** It is easy to see that if \(| (\theta_m - \lambda_{\mu_1}) \otimes \lambda_{\mu_2} | \leq M \), a one-pass program can be synthesized, i.e., the number of parallel I/Os is \( \frac{2N}{BD} \). When the above condition does not hold, we keep \( | \lambda_{\mu_1} | \) records for each loaded physical track and load the same physical track \( | \lambda_{\mu_2} | \) times. Moreover, since \(| \theta_m | = \frac{M}{BD} \), it can be easily determined that \(| \lambda_{\mu_2} | = [\frac{\lambda_m - \theta_m}{BD}] \). Because we write out each record only once, the number of parallel I/O operations is \((1 + \frac{|\lambda_m - \theta_m|BD}{M}) \frac{N}{BD} \). Combining these two cases yields the performance results presented in the theorem. Further, a program with this performance can be synthesized by using the procedure listed in Fig. 4.

\[ \square \]

We now use an example to illustrate the methods of determining augmented tensor bases and synthesizing parallel I/O programs for stride permutations. Assume that we have a stride permutation \( L_{46} \), which can be interpreted as an \( 8 \times 4 \) matrix transposition. The parameters of the model are defined as follows: \( D = 2 \), \( B_d = 2 \), \( B_b = 2 \), and \( M = 8 \). Then the input and output data distribution bases can be written as follows,

\[ \beta = e_{\phi}^x \otimes e_{\phi}^y \otimes e_{\phi}^2 \otimes e_{\phi}^2, \]

\[ \delta = e_{\phi}^y \otimes e_{\phi}^x \otimes e_{\phi}^2 \otimes e_{\phi}^2. \]
DO $g_1 = 0, 1$
DO $b_h = 0, 1$
DO $g_2 = 0, 1$
   // Parallel read from a track
   X(4g_2 + g_1 + b_h) ← parallel_read(4g_2 + 2g_1 + b_h)
ENDDO
   // Perform operations for a memory load
   Code(X(1 : 16) ← L_2^4(X(1 : 16)))
   // Write the result back
DO $b_d = 0, 1$
   // Parallel write to a track
   parallel_write(4b_d + 2b_d + g_1) ← X(4b_d + (4b_d + 3))
ENDDO
ENDDO
ENDDO

Figure 7: Parallel I/O Program for $L_4^{36}$

Moreover, the output data distribution basis can also be obtained by applying the stride permutation $L_4^{36}$ to the input data distribution basis. In other words, it can be written as,

\[ \delta = e_2^2 \otimes e_3^2 \otimes e_4^1 \otimes e_5^2. \]  \hspace{1cm} (19)

Then, following the procedure of the proof of Theorem 5.1, we can first determine the input and output loop bases as follows. We first factor $e_3^2$ as $e_2^2 \otimes e_3^2$. Then, by the algorithm presented in Fig. 6, we have,

\[ \lambda_\mu = e_2^2 \otimes e_3^2, \theta_\mu = e_2^2 \otimes e_5^2, \]  \hspace{1cm} (20)

\[ \lambda_m = e_2^2 \otimes e_3^2, \theta_m = e_3^2 \otimes e_3^2, \]  \hspace{1cm} (21)

\[ \lambda_n = e_2^2 \otimes e_3^2, \theta_n = e_3^2 \otimes e_5^2. \]  \hspace{1cm} (22)

Further, the records denoted by $\lambda_m \otimes \lambda_\mu$ or $\theta_m \otimes \theta_\mu$ will be used to form perfect memory-loads. The in-core computation can be determined by finding out the permutation which permutes $\lambda_m \otimes \lambda_\mu$ to $\theta_m \otimes \theta_\mu$. This can be easily determined as $L_2^2$. Since $| (\theta_\mu - \lambda_\mu) \otimes \lambda_\mu | \leq M$, a one-pass program, as shown in Fig. 7, can be synthesized by using the information determined above and the code generation algorithm presented in the previous subsection.

The procedure of computing $L_4^{36}$ using the synthesized program is illustrated in Fig. 8, and Fig. 9. Fig. 8 shows the input vector when explained as a matrix and its initial data distribution on two disks. It also shows the first two intermediate sub-transposition steps. Fig. 9 illustrates the successive two intermediate steps and the final outputs. Each of the intermediate sub-transposition steps reads a block of matrix, transposes the block in the internal memory and then writes the block onto disks. For clarity, we assume that the outputs are written on a different set of disks.

6 Synthesizing Programs for Tensor Products

In this subsection, we first present an algorithm to determine efficient loop bases for a tensor product under a given data distribution cyclic(B). Based on these loop bases and data distribution bases, we can determine
Figure 8: Example matrix transposition. (a) Inputs when viewed as an $8 \times 4$ two-dimensional array. (b) Input data distribution on two disks. (c) Load physical tracks $T_0, T_2$, in-core permutation, and write to physical tracks $T_0, T_2$. (d) Load physical tracks $T_1, T_3$, in-core permutation, and write to physical tracks $T_4, T_6$.

Figure 9: Example matrix transposition. (a) Load physical tracks $T_2, T_6$, in-core permutation, and write to physical tracks $T_1, T_3$. (b) Load physical tracks $T_5, T_7$, in-core permutation, and write to physical tracks $T_5, T_7$. (c) Outputs.
memory-loads and operations to each memory-load. In other words, the augmented tensor basis can be obtained. Therefore, a program can be generated by using the procedure discussed in Section 5.1. We also show that the cost of the program synthesized can be obtained from the algorithm.

Since the computation of the tensor product $I_R \otimes A_V \otimes I_C$ does not change the order of the inputs (or it can be computed in-place), we will use the same input and output data distribution bases for the input and output data and also the same input and output loop bases for programs synthesized in this subsection. Therefore, we will only consider input, input distribution and input loop bases. We summarize our results as a theorem and then present a constructive proof which constructs the augmented tensor basis. Before we present the theorem, we first introduce the concept of desired records and discuss several properties of the possible locations in which the desired records may reside on disks.

For the tensor product $I_R \otimes A_V \otimes I_C$, the major computational matrix $A_V$ is applied to $V$ input records and these $V$ records have a stride $C$ in the input vector. We call each of these $V$ records for the first $A_V$ computation a desired record. More specifically, $V$ desired records can be denoted as $\{X[iC]|0 \leq i \leq V-1\}$. Note that all of the other $A_V$ computations will have a similar data access pattern. For example, the second $A_V$ computation is applied to the $V$ inputs beginning from the second record with the same stride $C$.

We now discuss several properties of the possible locations in which the desired records may reside on disks.

- The consecutive desired records will be first stored in a logical block, and then the successive desired records will be stored to other logical blocks on other disks. Thus, for example, when $C > B_d$ and $V C < B$, the number of physical tracks which holds the desired records is $\frac{V}{C/B_d}$ rather than $\frac{V}{C/(B_d B)}$.
- If the desired records are stored on several disks, then each of these disks will contain the same number of desired records and the desired records in each of these disks are stored in the same relative locations.
- If the desired records are stored on several logical tracks, then all of the logical tracks which contain the desired records will have the same number of desired records and the desired records in each logical track are stored in the same relative locations.

The correctness of these properties follows the definition of data distribution, the regular data access pattern of each computational matrix in the input tensor product, and the assumptions that all of the parameters in the machine model and the input tensor product are powers of two. For example, the correctness of the first property can be explained as follows. Since $V C < B$, all of the desired records are stored in the first logical block. The distance of the physical blocks which contain the desired records is $\frac{C}{B_d}$. Therefore, the number of physical tracks which hold the desired records is $\frac{V}{C/B_d}$. These properties will be used in the proof of the following theorem.

**Theorem 6.2** Let the input data be distributed according to cyclic($B$). Let $N_t$ denote the number of physical tracks where the records for an $A_V$ computation are stored. Then for the tensor product $I_R \otimes A_V \otimes I_C$, where $R V C = N$ and $V \leq M$, if $N_t \leq \frac{M}{N_t}$, a program can be synthesized with $\frac{2N}{N_t B}$ parallel I/O operations; otherwise a program can be synthesized with $\frac{3N}{N_t} N_t$ parallel I/O operations.
The above theorem can also be stated in terms of tensor bases as follows. Let \( \lambda \) be the input data distribution basis. Let \( \lambda_\mu = \beta(2) \otimes \beta(4) \). Further assume that \( \lambda_{\mu_1} \) denotes a subset of \( \lambda_\mu \) and \( \lambda_\mu - \lambda_{\mu_1} (= \lambda_{\mu_2}) \) is moved into the memory basis. Then for the tensor product \( I_R \otimes A_V \otimes I_C \), where \( RVC = N \) and \( V \leq M \), if \( \lambda_{\mu_1} = \lambda_\mu \), a program can be synthesized with \( \frac{2N}{T_D} \) parallel I/O operations; otherwise a program can be synthesized with \( | \lambda_{\mu_2} | \frac{3N}{T_D} \). In the following proof of the theorem, we will show how to construct \( \lambda_{\mu_1} \) and \( \lambda_{\mu_2} \). We will also prove that \( | \lambda_{\mu_2} | = \frac{2M}{M_1} N_t \).

Proof:

1. **Determine input loop basis.** If the desired records for an \( A_V \) computation are stored in \( N_t \) physical tracks and \( N_t \leq \frac{M}{T_D} \), then we can simply load the \( N_t \) physical tracks each time and therefore a one-pass program can be generated. However, when \( N_t > \frac{M}{T_D} \), we can not keep all of the records in \( N_t \) physical tracks in the main memory. We take the following simple approach: we construct \( M/V \) sets of desired records by loading each physical and retaining in the main memory only those records which fall in these sets. Each physical track needs to be reloaded to perform computation on the remaining records. In terms of tensor bases, we need to do nothing more than factor and permute the input data distribution basis to reflect this data access pattern.

More specifically, we begin with \( \lambda_\mu = \beta(2) \otimes \beta(4) \), and \( \lambda_n \otimes \lambda_m = \lambda - \lambda_\mu \), where \( \lambda \) has the same initial value as defined in Section 5.2. For a one-pass program, we factor and permute \( \lambda_n \otimes \lambda_m \) to change the order of accessing physical tracks. However, for a multi-pass program, we need to factor and permute all of the \( \lambda_\mu \)s, since we need to keep part of the records loaded in the main memory and discard other records. As we discussed before, the part of the records to be kept or discarded can be denoted by a subset of the vector bases in the physical-track-element basis. In order to factor and permute a tensor basis to a desired form, we need to examine the relative values of the parameters in the targeted I/O model, the tensor product and the size \( B \) of the data distribution. We summarize the above ideas as an algorithm in Fig. 10, which is further explained as follows.

* **Initialization.** This step initializes the values of \( \lambda_n \otimes \lambda_m \), \( \lambda_\mu \), and several temporary variables.

For example, \( R_b \) denotes the maximum number of the desired records for an \( A_V \) computation in a physical block. \( R_t \) is the number of the desired records in a physical track. \( R_d \) is the number of disks where the desired records for an \( A_V \) are stored. \( S \) is the distance of two consecutive physical tracks which contain the desired records. Since the stride of two desired records is \( C \), \( R_b \) can be determined as \( \lceil \frac{B_d}{C} \rceil \). The correctness of \( R_t \) and \( N_t \) can be similarly verified. Compute will invoke a procedure to compute the values such as \( R_d \) and \( S \). Fig. 11(a) and Fig. 11(b) shows the details on how to determine those values.

The correctness of the algorithm in Fig. 11(a) for computing \( R_d \) can be proved as follows. When \( C \leq B_d \), the successive disks may contain the same number of the desired records if the desired records can not be stored in one logical block. The number of these successive disks is dependent on the value of \( V \). Further, since there are \( R_b B_b \) desired records per logical block, and \( R_b B_b = B \) (since \( R_b = \frac{B}{C} \) in this case), the number of disks which contain the desired records is equal to the smaller of \( \frac{V}{T_D} \) and \( D \). This results in the first case of the algorithm. Similarly, when \( B_d < C \leq B \),
\[ \lambda_n \otimes \lambda_m = e_g^G \otimes e_{b_2}, \lambda_p = e_d^D \otimes e_{b_3}, R_b = \left[ \frac{B_d}{C} \right] \]

**Compute** \((R_d)\)

\[ R_l = R_b R_d, N_l = \left[ \frac{Y}{M} \right] \]

**Compute** \((S)\)

\[ // One-pass program \]

If \( S \leq B_b \) then

\[ B_{b_1} = S, \text{ Factor } e_{b_1} \text{ as } e_{b_2} \otimes e_{b_3} \]

\[ \lambda_n \otimes \lambda_m = e_g^G \otimes e_{b_1} \otimes e_{b_2}, \lambda_p \]

else

\[ G_1 = \frac{S}{B_b}, G_2 = \frac{C}{B_b}, \text{ Factor } e_g^G \text{ as } e_{g_2}^G \otimes e_{g_1}^G \]

\[ \lambda_n \otimes \lambda_m = e_{g_2}^G \otimes e_{b_1} \otimes e_{b_2} \]

If \( C \leq B_d \) then \( Z = C \) else \( Z = \frac{B}{M} B_d \)

\[ // Multi-pass program \]

If \( N_l > \frac{M}{R_d} \) then

\[ // Further determine \( \lambda_{\mu_2} \) and \( \lambda_{\mu_1} \) \]

\[ X = \min \{ B_d, C, \frac{M}{R_d N_l} \} \]

If \( X = \min \{ B_d, C \} \) then \( Y = \frac{M}{R_d B_d N_l} \) else \( Y = 1 \)

Let \( e_d^D = e_{d_1}^D \otimes e_{b_2}^D \otimes e_{d_1}, e_{b_3}^D = e_{b_3}^D \otimes e_{b_2}^D \otimes e_{b_1}^X \)

\[ \lambda_{\mu_2} = e_{d_2}^D \otimes e_{b_2}^D, \lambda_{\mu_1} = \lambda_p - \lambda_{\mu_2} \]

\[ // Compute the value of \( Z \) \]

If \( X = C \) or \( X = \frac{M}{R_d N_l} \) then \( Z = X \)

If \( X = B_d \) then \( Z = Y B_d \)

Figure 10: Algorithm for determining input loop bases and the value of \( Z \) for a tensor product.

\[
\begin{array}{c|c}
\text{Compute}(R_d) & \text{Compute}(S) \\
\hline
\text{If } C \leq B_d \text{ then } R_d = \min \{ \frac{S}{M}, D \} & \text{If } C \leq B_d \text{ then } S = 1 \\
\text{If } B_d < C \leq B \text{ then } R_d = \min \{ \frac{C}{M}, D \} & \text{If } B_d < C \leq B \text{ then } S = \frac{B}{M} C \\
\text{If } B < C \leq BD \text{ then } R_d = \frac{BD}{C} & \text{If } B < C \leq BD \text{ then } S = 1 \\
\text{If } C > BD \text{ then } R_d = 1 & \text{If } C > BD \text{ then } S = \frac{C}{M} \]
\end{array}
\]

Figure 11: (a) Algorithm for computing \( R_d \) and (b) Algorithm for computing \( S \).

the successive disks may contain the desired records. Since in this case, each logical block contains \( \frac{B}{D} \) desired records, the number of disks which contain the desired records is again equal to the smaller of \( \frac{S}{M} \) and \( D \). For the third case, any two disks which contain two consecutive desired records have a stride \( \frac{C}{M} \). Therefore, \( R_d = \frac{B}{M} \). The last case is trivial. Similarly, we can prove the correctness of the algorithm in Fig. 11(b).

**One-pass program.** This step determines how to access physical tracks. The idea is straightforward. It determines the decompositions and permutations for \( \lambda_n \otimes \lambda_m \) based on the stride between two consecutive physical tracks which contain the desired records. The result from this step may also be needed for the next step to determine the final loop basis for synthesizing multi-pass programs.

**Multi-pass program.** If the number of physical tracks which hold the records for an \( A_V \) computation is larger than the number of physical tracks which the main memory can hold, then
a multi-pass program needs to be synthesized. More specifically, we need to determine which portions of the records in a physical track should be kept for each pass of computation. The basic idea of keeping records for the current memory-load can be described as follows.

First, for each desired record, we want to take \( X - 1 \) successive records and keep these \( X - 1 \) records with the corresponding desired record as the current memory-load. One approach of determining \( X \) is to take \( X \) as large as possible. However, \( X \) needs to satisfy the following three conditions. First, \( X \) must be less than the gap between any two consecutive desired records in a physical block. Second, \( X \) must be less than the size of a physical block. Third, all of the desired records with their \( X - 1 \) successive records should be able to fit into the main memory, which means that \((XR_t)N_t \leq M\), or \( XV \leq M \). These three conditions can be expressed as \( X = \min\{C, B_d \frac{M}{R_tN_t}\} \).

Fig. 12 shows an example of how to construct memory-loads by taking portions of the records from a physical block, where we assume that there are four desired records in a physical block, and \( C = 2X \). The example can be interpreted as follows. The physical block is first broken into eight sub-blocks. Then we take the records in the odd-numbered sub-blocks to construct one memory-load and take the records in the even-numbered sub-blocks to construct another memory-load. In terms of tensor bases, we first decompose \( e_{b_h}^{B_h} \) as \( e_{b_{d_2}}^4 \otimes e_{b_{d_2}}^2 \otimes e_{b_{d_1}}^X \). Then, we permute the resulting tensor basis as \( e_{b_{d_2}}^2 \otimes e_{b_{d_2}}^4 \otimes e_{b_{d_1}}^X \).

Second, we apply a similar idea for disks. For each disk which contains the desired record, we take \( Y - 1 \) successive disks and we keep the records at the same relative locations with the original disk in each successive disk for the current memory-load. We want to take the largest possible value of \( Y \) given that the number of the records kept must fit into the main memory. We consider the following two cases. First, \( X = \min\{B_d, C\} \). In this case, either all of the records between any two desired records or all of the records in a physical block are chosen to be kept for the current memory-load. However, if all of the records between any two desired records are chosen, all of the records in a physical block will be covered. Thus, it is identical to the case that all of the records in a physical block are chosen to be kept. Further, \( R_d \) disks contain desired records. Therefore, \( R_dB_d \) records are chosen from each physical track. In order to not overflow.
the main memory, we need that \( R_d Y B_d N_1 \leq M \). Second, \( X = \frac{M}{R_c N_1} \). In this case, we do not choose all of the records between two desired records. However, since we have already chosen the largest possible value for \( X \), the main memory has been filled up in this case. Therefore, we can not add any more records from successive disks from this approach. In other words, \( Y = 1 \).

An example, which is similar to the example shown in Fig. 12, can be constructed for disks. More specifically, if we view the records in a physical block as disks, \( X \) as \( Y \), \( R_b \) as \( R_d \), then we have an example for disks. Further, in terms of tensor bases, we can interpret this idea as follows. We first decompose \( e_d^D \) as \( e_d^D \otimes e_d^R \otimes e_d^Y \). Then, we permute it as \( e_d^D \otimes e_d^R \otimes e_d^Y \). The resulting tensor basis allows us to access odd-numbered subset of disks first and then even-numbered subset of disks.

We now consider an example which contains both disks and records in physical blocks. More specifically, we consider the example in which data can be represented by combining factored \( e_d^D \) and \( e_b^B \). Assume that we want to access the records first in the odd-numbered disk sub-blocks and then in the even-numbered disk sub-blocks. Further, for each physical block we want to access the records first in the odd-numbered sub-blocks and then in the even-numbered sub-blocks. To achieve this data access pattern, we move \( e_d^D \otimes e_b^B \) from their current locations in \( e_d^D \otimes e_b^B \) to the beginning of \( e_d^D \otimes e_b^B \). In the algorithm presented in Fig. 10, we have denoted \( e_d^D \otimes e_b^B \) as \( \lambda_{\mu_2} \). Therefore, to construct each memory-load, we can simply move \( \lambda_{\mu_2} \) into \( \lambda_n \otimes \lambda_m \) and put them anywhere in \( \lambda_n \).

For the following analysis, we assume that we have found the subsets of \( \lambda_m \), namely \( \lambda_{\mu_1} \) and \( \lambda_{\mu_2} \), by the above algorithm. \( \lambda_{\mu_2} \) is moved into the memory basis and will generate loop nests for data access. The other portions of the algorithm, which are used for computing the value of \( Z \), will be discussed in Step 3.

2. **Determine memory-load.** For a one-pass program, we can simply factor \( \lambda - \lambda_{\mu_1} \) as \( \lambda_n \otimes \lambda_m \) and take \( | \lambda_m | = \frac{M}{R_b D} \). For a multiple-pass program, we factor \( \lambda - \lambda_{\mu_1} \) to be \( \lambda_n \otimes \lambda_m \) such that \( | \lambda_m | = \frac{M}{| \lambda_{\mu_1} |} \) and all of the vector bases in \( \lambda_{\mu_2} \) appear in \( \lambda_n \). Moreover, for the multiple-pass program, as discussed in Section 5.2, we use \( \lambda_{\mu_2} \) to determine which records should be kept for the current memory-load.

3. **Determine operations for a memory-load.** The original tensor product can be regarded as \( R \) parallel applications of \( A_V \) to the inputs with a stride \( C \). When data are distributed among disks and loaded in units of physical tracks, the net effect is to possibly reduce the stride of the records which each \( A_V \) will access in main memory. The operations on a memory-load have the general form of \( I_{\mu_2} \otimes A_V \otimes I_{\lambda_n} \). However, the value of \( Z \) will depend on the relative values of the parameters in the target machine model and the input tensor product. The algorithm presented in Fig. 10 can be used to determine this value. The correctness of the value of \( Z \) obtained from the algorithm can be proven as follows. For one-pass programs, when \( C \leq B_d \), we do not change the stride for sub-computations.

\footnote{More specifically, the initial \( \lambda_n \otimes \lambda_m \) should be modified to \( \lambda_n' \otimes \lambda_m' \), where \( \lambda_n' \) contains the last factors of \( \lambda_n \otimes \lambda_m \) and \( | \lambda_n' | = \frac{M}{| \lambda_{\mu_1} |} \), and \( \lambda_n' \) contains \( \lambda_n \otimes \lambda_m \cdot \lambda_n' \) and \( \lambda_{\mu_2} \).}
Therefore, $Z = C$. Otherwise, the stride will be reduced to be equal to the distance of two consecutive desired records in a physical track, which is equal to $\frac{2}{M}B_d$. For multi-pass programs, when $X = C$, we choose all of the records between any two desired records for the current memory-load, so the stride of in-core computation does not change. When $X = \frac{M}{B_dN_t}$, we reduce the stride of in-core computation from $C$ to $X$. When $X = B_d$, the next desired record is not in the same physical block. Since we keep $Y$ disks as a subset of disks, we reduce the stride from $C$ to $YB_d$.

4. I/O cost of synthesized programs. For a one-pass program which does not move any vector bases in $\lambda_\mu$, the number of parallel I/Os is simply equal to $\frac{2N}{B_dN_t}$. In other words, the synthesized program is optimal in terms of the number of I/Os. For a multi-pass program, we need to read the inputs $|\lambda_\mu_2|$ times. Therefore the number of parallel I/O operations is $|\lambda_\mu_2| \times \frac{3N}{B_dN_t}$. From the algorithm presented in Fig. 10, we can determine that $|\lambda_\mu_2| = \frac{B_d}{N_t}$. We therefore can attain the performance presented in the Theorem.

The constant 3 can be explained as follows. When we store a physical track, we need to read that physical track into main memory again, since portions of the records in that physical track have been discarded. By reloading this physical track, we can reassemble the physical track with the part of updated records and then write it out in parallel. Otherwise, part of the records to be written out in that physical track may not be correct. Further, “reassembling” the physical track needs to use the tensor basis $\theta_\mu_2$ (notice that $\theta_\mu_2$ is equal to $\lambda_\mu_2$) to put the updated records into the correct locations on the physical track. This is similar to using $\lambda_\mu_2$ to take sub-blocks out from a loaded physical track for the current memory-load.

Now, a program with the performance discussed above can be synthesized by using the procedure listed in Fig. 4. However, to be accurate, when synthesizing a multi-pass program, we need to incorporate the idea of “reassembling” a physical track into the write-out part of the procedure listed in Fig. 4, which, as we discussed above, is nothing more than using the linearization of $\theta_\mu_2$ to put sub-blocks in the current memory-load into the correct locations of the reloaded physical track.

\[ \square \]

Note that the value of $N_t$ can be determined at the initialization step. Therefore, the performance of the synthesized program for a tensor product can be determined without generating the whole augmented tensor basis. This result is used in the first phase of transforming tensor product formulas, where we need the performance value for each tensor product to determine efficient transformations.

6.1 Determining Efficient Data Distributions

In the previous subsections, we presented approaches for synthesizing efficient I/O programs for a given data distribution. We now present an algorithm to determine a data distribution which optimizes the performance of the synthesized program. The idea of the algorithm is as follows. We begin with the physical track distribution $cyclic(B_d)$, i.e., initially $B = B_d$. If a one-pass program can be synthesized under this data distribution, then $B_d$ is the desired block size for the data distribution. Otherwise, we double the
\[
B = B_d \\
\text{Cost} = \text{number of I/Os when using cyclic}(B) \\
\text{while } (\text{Cost} \neq \frac{2N}{D_{l_4}} \text{ and } B \leq \frac{N}{D_{l_4}}) \text{ do} \\
B = 2 \times B \\
C_{\text{new}} = \text{number of I/Os when using cyclic}(B) \\
\text{If } C_{\text{new}} \leq \text{Cost} \text{ then } \text{Cost} = C_{\text{new}} \text{ else break} \\
\text{output distribution size } = B/2, \text{ number of I/Os } = \text{Cost}
\]

Figure 13: Algorithm for computing the efficient size of data distributions.

value of \( B \). If the performance of the synthesized program under this distribution increases, we continue this procedure. Otherwise, the algorithm stops and the current block size is the desired size of data distributions. We formalize this idea in Fig. 13.

### 6.2 Transforming Tensor Product Formulas

In this subsection, we discuss techniques of program synthesis for tensor product formulas. There are several strategies for developing I/O-efficient programs, such as exploiting locality and exploiting parallelism in accessing the data. Similar ideas have been discussed in [16], where they use factor grouping to exploit locality and data rearrangement to reduce the cost of I/O operations. We have also presented a greedy method which uses factor grouping to improve the performance of block recursive algorithms for Vitter and Shriver’s striped two-level memory model with a fixed block size of data distribution [11].

Factor grouping combines contiguous tensor products in a tensor product formula and therefore reduces the number of passes to access secondary storage. Consider the core Cooley-Tukey FFT computation, which does not contain the initial bit-reversal operation and the twiddle factor computation. For \( i=2 \) and \( i=3 \), we have the tensor products \( I_{2n-2} \otimes F_2 \otimes I_2 \) and \( I_{2n-3} \otimes F_2 \otimes I_2' \), respectively. Assuming that each of these tensor products can be implemented optimally, the number of parallel I/O operations required to implement these two steps individually is \( \frac{4N}{D_{l_4}} \). However, they are contiguous tensor products in Formula (2). Hence, by using the properties of tensor products, such as Properties 1 and 2 listed in Section 3, they can be combined into one tensor product,

\[
(I_{2n-2} \otimes F_2 \otimes I_2)(I_{2n-3} \otimes F_2 \otimes I_2') \\
= (I_{2n-3} \otimes I_2 \otimes F_2 \otimes I_2)(I_{2n-3} \otimes F_2 \otimes I_2 \otimes I_2) \\
= (I_{2n-3} \otimes (I_2 \otimes F_2) \otimes I_2)(I_{2n-3} \otimes (F_2 \otimes I_2) \otimes I_2) \\
= I_{2n-3} \otimes F_2 \otimes F_2 \otimes I_2,
\]

which may also be implementable optimally by using only \( \frac{2N}{D_{l_4}} \) parallel I/O operations.

Data rearrangement uses the properties of tensor products to change data access patterns. For example, the tensor product \( I_R \otimes A_V \otimes I_C \) can be transformed into the equivalent form \( (I_R \otimes L_{V}^C) \ (I_{RC} \otimes A_V) \ (I_R \otimes L_{C}^V) \). In the best case, the number of parallel I/Os required is \( \frac{3N}{D_{l_4}} \) after using this transformation, since at least three passes are needed for the transformed form. Because of the extra passes introduced by this transformation, it is not profitable to use it for our targeted machine model. Further, the first and
the last terms in the transformed formula may not be implementable optimally. Therefore, we have not incorporated this transformation into our current optimization procedures.

Minimizing I/O Cost by Dynamic Programming Since factor grouping (as shown above) and the size of the data distribution (as will be shown in the next section) have a large influence on the performance of synthesized programs, we take the following approach for determining an optimal manner in which a tensor product formula can be implemented. We use the algorithm for determining the optimal data distribution presented in Fig. 13 as a main routine. However, for each cyclic\(B\) data distribution, we use a dynamic programming algorithm to determine the optimal factor grouping. Hence, we also call this method a multi-step dynamic programming method.

Let \(C[i, j]\) be the optimal cost (the minimum number of I/O passes required to access the out-of-core data) for computing \((j - i)\) tensor factors from the \(i\)th factor to the \(j\)th factor in a tensor product formula. Then \(C[i, j]\) can be computed as follows:

\[
C[i, j] = \begin{cases} 
C_0 & \text{if } i = j \\
\min_{i \leq k < j} \{C[i, k], C[k + 1, j]\} & \text{if } i < j
\end{cases}
\]

In the above formula, \(C_0\) denotes the cost for computing a tensor product. The method of determining the cost of a tensor product has been discussed in Section 5.3. The values of \(C_0\) can be computed using the results in Theorem 6.2 and the algorithm presented in Fig. 11(a) to compute \(N_i\). A special case of \(k = j\) needs to be further explained. When \(k = j\), we assume that \(C[j + 1, j] = 0\) and we use \(C[i, k]\) to represent the cost of grouping all the tensor product factors from \(i\) to \(j\) together. Because the grouped tensor product is a simple tensor product, the value of \(C[i, k]\) in this case can also be determined by using the results in Theorem 6.2 and the algorithm presented in Fig. 11(a) to compute \(N_i\). However, in this case, if \(k - i > m\), or the size of grouped operations is larger than the size of the main memory, we do not want to group all of the \(k - i\) factors together. We assign a large value such as \(\infty\) to \(C[k, j]\) to prevent it from being selected.

7 Performance Results of Synthesized Programs

7.1 Matrix Transposition

Given the flexibility of choosing different data distributions, we can synthesize programs with better performance than those obtained using fixed size data distributions for stride permutations. We present a set of experimental results for the number of I/O operations required by the cyclic\(B_d\) distribution and cyclic\(B\) distribution, where the size \(B\) of the distribution varies. These results are summarized in Table 1 and Table 2. From the tables, we can see that the number of passes is not a monotonically increasing or decreasing function. However, it normally decreases and then increases as \(B\) is increased. Therefore it is likely that the algorithm in Fig. 13 will find an efficient size of data distributions.

7.2 Tensor Products

The number of I/O passes required by the synthesized programs are summarized in Table 3, Table 4, and Table 5 by going through various cases of \(N_i\). In those tables, \(M_i = \frac{M}{\pi_{2B}}\) is the maximum number of
Table 1: Number of I/O passes for stride permutation $L_{PQ}^D$. $D = 4$, $B_d = 4$, $M = 64$, and $N = PQ = 2048$.

<table>
<thead>
<tr>
<th>$B$</th>
<th>$B_d$</th>
<th>$2B_d$</th>
<th>$4B_d$</th>
<th>$8B_d$</th>
<th>$16B_d$</th>
<th>$32B_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = 2^4, Q = 2^8$</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$P = 2^4, Q = 2^7$</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$P = 2^5, Q = 2^9$</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$P = 2^6, Q = 2^{10}$</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Number of I/O passes for stride permutation $L_{PQ}^D$. $D = 16$, $B_d = 512$, $M = 2^{22}$, and $N = PQ = 2^{30}$.

physical tracks in a memory-load. We can verify that the results presented here are more comprehensive than the results presented in [11]. In most cases, using the approach presented in Section 5.3, we can actually synthesize programs with better performance. For example, when $VC > M$, $M < VDB_d$ and $M > VB_d$, from [11], a program with $\frac{VB_dD}{M}$ passes will be synthesized. However, for these conditions, we have that $C > B_d$, and $VC > M$. If we further assume that $C < B_dD$, then from the results in Table 3 and Table 4, we can synthesize a program with $\frac{VC}{M}$ passes, which is less than $\frac{VB_dD}{M}$.

Table 3: Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$.

We now show that by using an appropriate cyclic($B$) data distribution, a better performance program can be synthesized for most of the cases. Several typical examples are shown in Table 6. We notice that when we increase $B$, we can reduce the number of passes of data access for most of the cases and the decrease in the number of passes can be as large as eight times. The values in the table also suggest that we can use the algorithm presented in Fig 13 to find an efficient size of data distributions for a given tensor product. We also notice that for some cases, such as $C \leq B_d$, we can not improve the performance. The reason is that the stride required by $A_V$ is less than the size of the physical block, and we can not reduce it further by redistribution.

7.3 Tensor Product Formulas

We show the effectiveness of the multi-step dynamic programming method by comparing the programs synthesized by it with the programs synthesized by the greedy method and the dynamic programming method (applied to a data distribution of fixed size), respectively. The example we use is the core Cooley-Tukey FFT computation. The results for several typical sizes of inputs are shown in Table 7. We find that
Table 4: Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$.

<table>
<thead>
<tr>
<th>$B_d \leq C &lt; B$</th>
<th>$B &lt; VC \leq BD$</th>
<th>$VC &gt; BD$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V \leq M_t$</td>
<td>$V &gt; M_t$</td>
<td>$\frac{M_t}{n_t} \leq M_t$</td>
</tr>
<tr>
<td>1</td>
<td>$\frac{M_t}{n_t} \leq M_t$</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5: Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$.

using dynamic programming for a fixed size cyclic($B_d$) distribution normally can not improve performance over the greedy method. However, by using the multi-step dynamic programming method, we can reduce the number of passes for the synthesized programs by at least 1 if N is very large. Because the input size is large, the performance gain by eliminating even one pass to access out-of-core data is significant.

8 Conclusions

We have presented a novel framework for synthesizing out-of-core programs for block recursive algorithms using the algebraic properties of tensor products. We use the striped Vitter and Shriver’s two level memory model as our target machine model. However, instead of using the simpler physical track distribution normally used by this model, we use various block-cyclic distributions supported by the High Performance Fortran to organize data on disks. Moreover, we use tensor bases as a tool to capture the semantics of data distributions and data access patterns. We show that by using the algebraic properties of tensor products, we can decompose computations and arrange data access patterns to generate out-of-core programs automatically.

We demonstrate the importance of choosing the appropriate data distribution for the efficient out-of-core implementations through a set of experiments. The experimental results also shows that our simple algorithm for choosing the efficient data distribution is very effective. From the observations about the importance of data distributions and factor grouping for tensor products, we propose a dynamic programming approach to determine the efficient data distribution and the factor grouping. For an example FFT computation, this dynamic programming approach can reduce the number of I/O passes by at least one comparing with using a simpler greedy algorithm.

References


Table 6: Number of I/O passes for the tensor product $I_K \otimes A_V \otimes I_C$ with various data distributions. $D = 16$, $B_d = 512$, $M = 2^{22}$, and $N = RVC$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$2^{20}$</th>
<th>$2^{20}$</th>
<th>$2^{20}$</th>
<th>$2^{100}$</th>
<th>$2^{150}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy ($B = B_d$)</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>D.P. ($B = B_d$)</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>M.D.P.</td>
<td>4 ($B = 2^{10}$)</td>
<td>6 ($B = 2^{12}$)</td>
<td>8 ($B = 2^{15}$)</td>
<td>9 ($B = 2^{18}$)</td>
<td>15 ($B = 2^{14}$)</td>
</tr>
</tbody>
</table>

Table 7: Number of I/O passes for the synthesized programs using Greedy, Dynamic programming (D.P), and Multiple-step dynamic programming(M.D.P) methods ($D = 16$, $B_d = 512$, and $M = 2^{22}$).


