J Supercomput (2009) 47: 171–197
DOI 10.1007/s11227-008-0186-0

Matrix-based streamization approach for improving locality and parallelism on FT64 stream processor

Xuejun Yang · Jing Du · Xiaobo Yan · Yu Deng

Published online: 16 March 2008

Abstract FT64 is the first 64-bit stream processor designed for scientific computing. It is critical to exploit optimizing streamization approaches for scientific applications on FT64 due to the inefficiency of direct streamization approach. In this paper, we propose a novel matrix-based streamization approach for improving locality and parallelism of scientific applications on FT64. First, a Data&Computation Matrix is built to abstract the relationship between loops and arrays of the original programs, and it is helpful for formulating the streamization problem. Second, three key techniques for optimizing streamization approach are proposed based on the transformations of the matrix, i.e., coarse-grained program transformations, fine-grained program transformations, and stream organization optimizations. Finally, we apply our approach to ten typical scientific application kernels on FT64. The experimental results show that the matrix-based streamization approach achieves an average speedup of 2.76 over the direct streamization approach, and performs equally to or better than the corresponding Fortran programs on Itanium 2 except CG. It is certain that the matrix-based streamization approach is a promising and practical solution to efficiently exploit the tremendous potential of FT64.

Keywords FT64 · D&C Matrix · Streamization · Program transformation · Stream organization

1 Introduction

The stream processors are designed to address the processor-memory gap through streaming technology [1–4]. They have shown tremendous performance advantages

X. Yang · J. Du (✉) · X. Yan · Y. Deng
PDL, School of Computer, National University of Defense Technology, Changsha, Hunan, 410073, China
e-mail: jdstarry@yahoo.com.cn

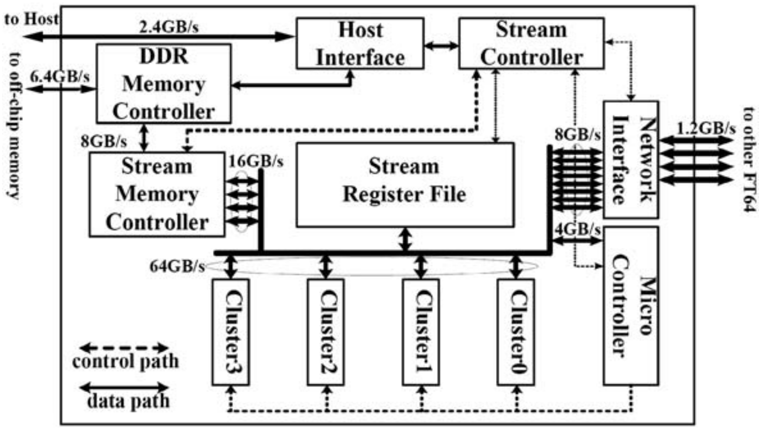


Fig. 1 FT64 stream architecture

in the domains regarding signal processing, multimedia and graphics [5–7]. Yet it has not been sufficiently validated whether stream processor is efficient for scientific computing.

FT64 and FT64-oriented compiler optimizations are presented to address this problem. FT64 is the first 64-bit stream processor for scientific computing [8], which has a peak performance of 16GFLOPS. Its instruction set architecture (ISA) is optimized for scientific computing. FT64 consists of four ALU clusters and a three-level memory hierarchy, which includes LRF, SRF, and off-chip memory used to keep so many ALUs saturated during stream processing. Figure 1 diagrams the FT64 stream architecture. Like other stream processors, FT64 supports stream programming model [9, 10]. It expresses a program as a set of computation kernels which produce and consume data streams, which can be classified into two types, namely basic streams and derived streams. Any modification to the derived stream results in additional memory access overhead. This explicit stream programming model incurs heavy burden on inheriting legacy scientific programs [11]. Therefore, it is crucial to explore efficient method of mapping scientific programs to stream programs on FT64, namely streamization technique.

The prior researches show that loops and arrays are the fundamental structures of most scientific applications. Thus, the development of stream programs on FT64 focuses on streamizing loops of original programs. A direct streamization process for a given program is as follows: inner perfect loop nests are transformed to kernels, array variables are mapped to basic streams, and array references are looked upon as derived streams. Unfortunately, the direct streamization approach can not exploit sufficient locality and parallelism of the generated stream programs on FT64, and, therefore, it is inefficient and unpractical. To address the problem, this paper proposes a matrix-based streamization approach for improving locality and parallelism of scientific programs on FT64 stream processor. The proposed streamization approach has been implemented in SCompiler, which is a compiler used to map Fortran programs to FT64. The contributions of the research lie in four aspects:

- The Data&Computation Matrix (D&C Matrix). It is proposed to abstract the relationship between loops and arrays of the original program, and help formulate the streamization problem. Then based on the matrix, three key techniques for optimizing streamization are proposed as follows.
- Coarse-grained program transformations. The technique performs global program transformations among loops in terms of the column transformations of the original program's D&C Matrix, so as to improve the locality in LRF and SRF, computational intensiveness, and parallelism of the generated stream program.
- Fine-grained program transformations. The technique reorders computations and data within loops by shortening the computation distances and data distances in the original program's D&C Matrix, so as to optimize the locality in kernels, computational intensiveness, and derived streams of the generated stream program.
- Stream organization optimizations. The technique focuses on selecting optimum basic streams and reusing derived streams based on analyzing the array references of D&C Matrix, which is optimized by the above two techniques. So, the generated stream programs can achieve low memory overhead and high locality in SRF.

We perform an evaluation of the proposed matrix-based streamization on FT64, compared with the direct streamization approach on FT64 and the original programs on Itanium 2. The experimental results of ten typical scientific application kernels show that the optimizing streamization approach achieves an average speedup of 2.76 over the direct streamization approach, and performs equally to or better than the original programs on Itanium 2 except CG. It is certain that the matrix-based streamization approach is a promising and practical solution to efficiently exploit the tremendous potential of FT64.

The remainder of this paper is organized as follows: Sect. 2 presents the overview of our approach; Sect. 3 presents the key techniques of matrix-based streamization process. The experiments are shown in Sect. 4. Section 5 presents related work. In Sect. 6, we conclude the paper and discuss some future works.

2 Overview of optimizing streamization

2.1 FT64 processing system

FT64 is an access/execute decoupled processor [8]. Each ALU cluster shown in Fig. 2 is composed of four floating-point multiply-accumulate units (FMACs), a divide/square root unit (DSQ) and their local register files (LRFs), a 256-word scratchpad memory (SP) used for local arrays, etc. The memory hierarchy of FT64 includes

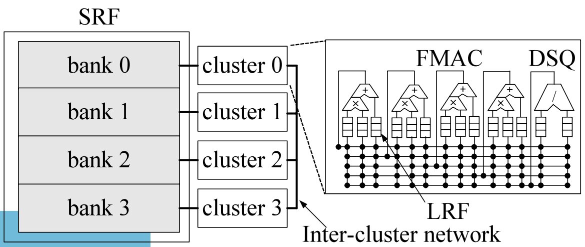


Fig. 2 The microarchitecture of ALU clusters

LRF, stream register file (SRF), and off-chip memory. The LRFs directly feed those arithmetic units inside the clusters with their operands. Its total capacity is 19 KB and total bandwidth is 544 GB/s. The SRF is a software-controlled memory hierarchy which is used to buffer the input/output and intermediate streams during execution. It is in the size of 256 KB and divided into four banks which correspond to the four clusters, respectively. The off-chip memory is controlled by a stream memory controller (SMC) and a DDR memory controller (DDRMC), which cooperate to load and store derived streams. The SMC operates at 500 MHz with the bandwidth of 16 GB/s to the SRF, and 8 GB/s to the DDRMC. The DDRMC works at 200 MHz with the bandwidth of 6.4 GB/s.

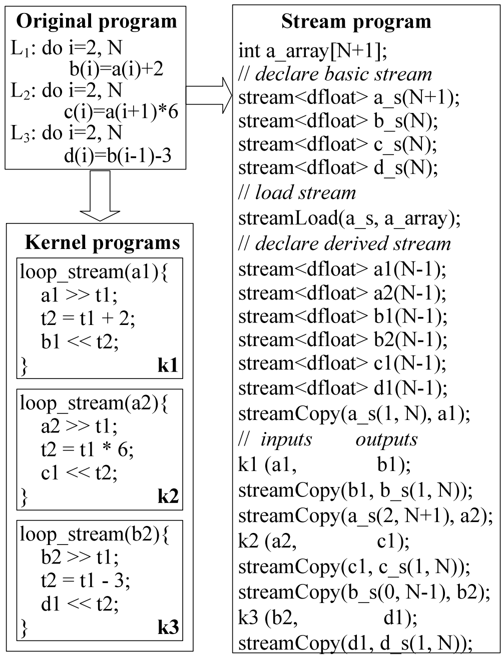
FT64 supports stream programming model, such as StreamC/KernelC [12–14]. In the model, stream applications consist of stream-level programs and kernel-level programs. A stream-level program specifies the order in which kernels execute and organize data into sequential streams that are passed from one kernel to the next. A kernel-level program is structured as a loop that processes elements from each input stream and generates outputs for each output stream. Every data stream is a sequence of data records with the same type, which can be classified into two kinds: basic streams and derived streams. Basic stream defines a new sequence of data records while derived stream refers to all or part of an existing basic stream. Any modification to the derived stream results in memory overhead. However, there are a few stream versions of scientific programs currently. And the simple and direct streamization of scientific programs is inefficient. Therefore, it is necessary to explore an optimizing streamization approach to achieve significant performance improvements on FT64.

2.2 Basic idea

In this section, we will take the program in Fig. 3 and Fig. 4 as an example to illustrate the basic idea of the optimizing streamization.

First, Fig. 3 shows an example of the direct streamization process. The top left of the figure gives an example of the original FORTRAN95 program, which consists of three loop nests (L_1, L_2, L_3) and four arrays (a, b, c, d). The other parts of the figure give the stream-level program and kernel-level program generated by directly mapping the original program to FT64. This direct streamization transforms the three loops to three kernels ($k1, k2, k3$), respectively, maps all arrays to basic streams (a_s, b_s, c_s, d_s) whose length are according to the total length of the corresponding arrays, and generates derived streams from all array references within the loop based on the basic streams. For example, as defined in Fig. 3, stream a_s is a basic stream with the length of $N+1$, which is exactly transformed from array a. The input stream $a1$ and $a2$ of kernel $k1$ and $k2$ are both defined as derived streams of basic stream a, which correspond to array references $a(i)$ and $a(i+1)$, respectively.

Obviously, the new generated stream program exposes many problems. First, expressing each loop in the original program to an individual kernel leads to overfull amount of kernels and small kernel granularity, and thus the computations in a kernel are too few to exploit multi-level parallelism effectively. Second, without centralizing affinitive computations and data, the program loses the locality in LRF and SRF and achieves low computational intensiveness. Thus, the arithmetic units cannot be



Original program

```
L1: do i=2, N
        b(i)=a(i)+2
L2: do i=2, N
        c(i)=a(i+1)*6
L3: do i=2, N
        d(i)=b(i-1)-3
```

Kernel programs

```
loop_stream(a1){
   a1 >> t1;
   t2 = t1 + 2;
   b1 << t2;
}                    k1

loop_stream(a2){
   a2 >> t1;
   t2 = t1 * 6;
   c1 << t2;
}                    k2

loop_stream(b2){
   b2 >> t1;
   t2 = t1 - 3;
   d1 << t2;
}                    k3
```

Stream program

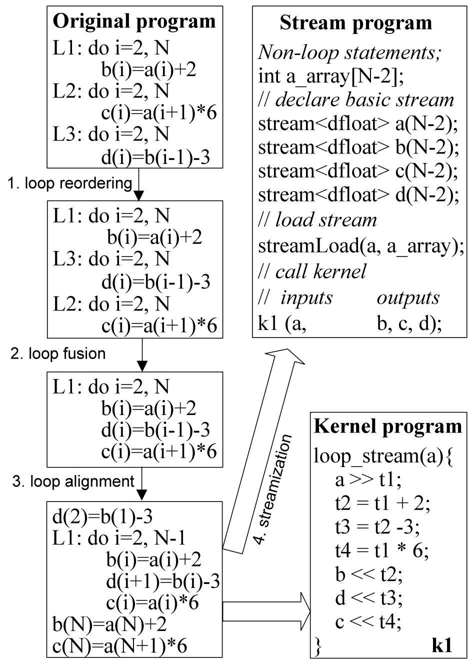
```
int a_array[N+1];
// declare basic stream
stream<dfloat> a_s(N+1);
stream<dfloat> b_s(N);
stream<dfloat> c_s(N);
stream<dfloat> d_s(N);
// load stream
streamLoad(a_s, a_array);
// declare derived stream
stream<dfloat> a1(N-1);
stream<dfloat> a2(N-1);
stream<dfloat> b1(N-1);
stream<dfloat> b2(N-1);
stream<dfloat> c1(N-1);
stream<dfloat> d1(N-1);
streamCopy(a_s(1, N), a1);
//  inputs       outputs
k1 (a1,          b1);
streamCopy(b1, b_s(1, N));
streamCopy(a_s(2, N+1), a2);
k2 (a2,          c1);
streamCopy(c1, c_s(1, N));
streamCopy(b_s(0, N-1), b2);
k3 (b2,          d1);
streamCopy(d1, d_s(1, N));
```

Fig. 3 An example of direct streamization

kept saturated during stream processing. Third, generating basic streams and derived streams without optimizing practical array reference patterns causes abundant derived streams, which impose great pressure on memory accesses [15].

To address the problems mentioned above, the optimizing streamization process of the example program is proposed, which is shown in Fig. 4. First, loop reordering and loop fusion are used here to reduce loops and tighten the reuse distance of array b. Second, loop alignment is used to unify the different references of the array a and b, so that the array references are reduced by 33% (2/6). Finally, since the original program only accesses the subsets of all arrays, the basic streams are organized based on these subsets in stead of the whole arrays. Note that the nonloop statements needn't be mapped to the stream forms here. Therefore, compared with the result in Fig. 3, the generated stream program in Fig. 4 achieves many improvements as follows. First, the number of kernels is reduced to 1/3, i.e., only one kernel is produced here. Second, suppose $N = 2$, the total size of basic streams is $(4 \times N - 8) \times 8B = 252B$, which is 96% of that of the basic streams from direct streamization. Third, the derived streams are reduced from 6 to 0, namely the derived streams no longer exist in the optimized stream program.

In brief, the basic idea of the optimizing streamization is to firstly insert program transformation process to produce stream architecture oriented intermediate codes. Then apply optimizing strategy for stream organization in streamization process. The direct and optimizing streamization processes are shown in Fig. 5. The targets of optimizing streamization are as follows:



Original program
```
L1: do i=2, N
        b(i)=a(i)+2
L2: do i=2, N
        c(i)=a(i+1)*6
L3: do i=2, N
        d(i)=b(i-1)-3
```

```
L1: do i=2, N
        b(i)=a(i)+2
L3: do i=2, N
        d(i)=b(i-1)-3
L2: do i=2, N
        c(i)=a(i+1)*6
```

```
L1: do i=2, N
        b(i)=a(i)+2
        d(i)=b(i-1)-3
        c(i)=a(i+1)*6
```

```
d(2)=b(1)-3
L1: do i=2, N-1
        b(i)=a(i)+2
        d(i+1)=b(i)-3
        c(i)=a(i)*6
b(N)=a(N)+2
c(N)=a(N+1)*6
```

Stream program
```
Non-loop statements;
int a_array[N-2];
// declare basic stream
stream<dfloat> a(N-2);
stream<dfloat> b(N-2);
stream<dfloat> c(N-2);
stream<dfloat> d(N-2);
// load stream
streamLoad(a, a_array);
// call kernel
// inputs      outputs
k1 (a,         b, c, d);
```

Kernel program
```
loop_stream(a){
    a >> t1;
    t2 = t1 + 2;
    t3 = t2 -3;
    t4 = t1 * 6;
    b << t2;
    d << t3;
    c << t4;
}                    k1
```

Fig. 4 Optimizing streamization for the example

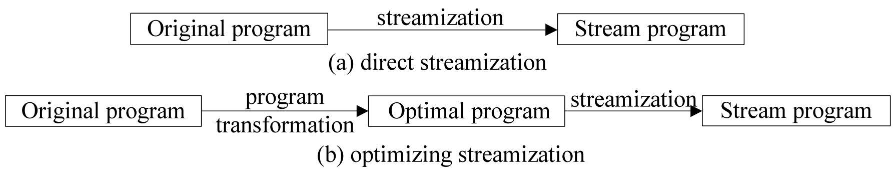


Fig. 5 The direct and optimizing streamization processes

- By performing program transformations among loops, data reuse within and between loops is enhanced and loop granularity is enlarged, so as to improve the locality in LRF and SRF and parallelism of the generated stream programs.
- By performing program transformations within loops, affinitive computations and data are centralized and various references on the same array are optimized, so as to achieve high computational intensiveness, high LRF locality, and low derived stream overheads of the generated stream programs.
- By optimizing stream organizations in streamization process, optimum basic streams and derived streams are formed, so as to improve the memory bandwidth utilization of the generated stream programs.

2.3 D&C Matrix

The proposed optimizing streamization approach relates to computation and data reordering within loops, transformations among loops, and data reference pattern for

$$\begin{array}{c|ccccc}
 & L_1 & L_2 & \cdots & L_j & \cdots \\
\hline
D_1 & m_{11} & m_{12} & & & \\
D_2 & m_{21} & \ddots & & \vdots & \\
\vdots & & & \ddots & & \\
D_i & & \cdots & & m_{ij} & \\
\vdots & & & & &
\end{array}$$

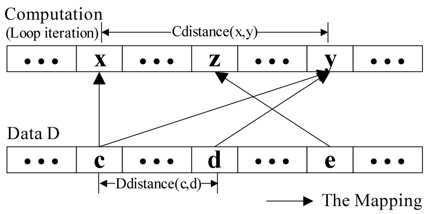


Fig. 6 The D&C Matrix and the mapping in the matrix

Fig. 7 D&C Matrix of the example program

$$\begin{array}{c|ccc}
 & L_1 & L_2 & L_3 \\
\hline
a & m_{11} & m_{12} & \\
b & m_{21} & & m_{23} \\
c & & m_{32} & \\
d & & & m_{43}
\end{array}$$

stream organization. Based on the loops, computations and data information of a given program, our approach builds a matrix called Data&Computation Matrix (D&C Matrix) to formulate the relationship between these informations, as shown in Fig. 6. The matrix shows the reference pattern between each iteration of all loops and each array. Each raw of the D&C Matrix represents an array and each column of the matrix describes the reference pattern of a loop. Suppose D_i represents a sequential layout of the array in the ith row and L_j denotes the loop in the jth column, the item in the ith row and the jth column position in the D&C Matrix corresponds to a mapping denoted as $m_{ij} : D_i \rightarrow I$, where I is an iteration vector which presents a computation sequence according to the data layout. In other words, $m_{ij}(d)$ for $d \in D_i$ is the iteration numbers in L_j that access d. For the original program given in Fig. 4, the corresponding D&C Matrix is organized as shown in Fig. 7. Note that when each array in the nest is referenced many times, the mapping m_{ij} maps multiple data to multiple iterations. The right part of Fig. 6 gives an example of this mapping such that $m_{ij}(c) = \{x, y\}$, $m_{ij}(d) = y$ and $m_{ij}(e) = z$ for $c, d, e \in D_i$ and $\{x, y\}, \{y\}, \{z\} \in I$. To afford facilities for clarifying our approach, we express the reverse mapping of m_{ij} as $m_{ij}^{-1}(y) = \{c, d\}$. In order to explain our technique, it is necessary to introduce the following definitions.

Definition 1 Suppose that two iterations x and y of a loop access the same data, the computation distance Cdistance(x, y) is defined as the number of iterations between x and y such that Cdistance$(x, y) = y - x$.

Definition 2 Suppose that data c and d are accessed successively, the data distance Ddistance(c, d) is defined as the interval between the two data layouts such that Ddistance$(c, d) = d - c$.

Each item in the D&C Matrix is a mapping that presents some significant information of the stream reference pattern, including the temporal locality, the spatial locality, the reference order, and the stream organization. For instance, data D in the right part of Fig. 6 presents the successive layout like stream layout so that we can loop upon D as a basic stream. The Cdistance(x, y) expresses the temporal locality of record c and Ddistance$(c, \underline{d})$ denotes the spatial locality of stream D. Furthermore, we treat loop iteration spaces unrolling as the stream organization pattern, that is, the data sequence accessed by all the ordinal iterations can be organized as a stream. To clarify distinctly, we formulate the approach of stream organization as follows, where $ORG(i, j)$ is the stream organization of the ith array accessed by the jth loop in the D&C Matrix, the symbol "$\sum^{+}$" denotes the connection of different data sequences, $\max(x)$ is the maximum iteration of the loop body

$$ORG(i, j) = \sum_{x=0}^{\max(x)}{}^{+} m_{ij}^{-1}(x|x \in I) \tag{1}$$

The D&C Matrix, once built, contains all the memory access information between all loops and all arrays. Based on the transformation of this matrix, optimizing streamization can be implemented easily and efficiently.

3 Matrix-based optimizing streamization

The work of this paper focuses on the streamization process in SCompiler, which is a compiler to map Fortran programs to low-level codes executed on FT64. Figure 8 gives the framework of SCompiler. And the right part of this figure shows the steps of the matrix-based streamization process (grey part in the framework). First, after the front-end parsing, the D&C Matrix like Fig. 6 is produced. Then taking the matrix representation as input, some architecture-oriented program transformations are used to generate the optimal Fortran intermediate code. Finally, optimizing streamization is performed to map the intermediate code to stream program. We will discuss the implementation of matrix-based streamization process in detail, including coarse-grained program transformations, fine-grained program transformations, and stream organization optimizations.

3.1 Coarse-grained program transformations

Coarse-grained program transformations regard a loop as an atomic group, and perform global loop transformations among loops [16–18], such as loop reordering, loop fusion, arrays unifying, strip-mining, etc. This approach aims at enabling array reuse between loops, improving locality within loops, and enlarging the loop granularity

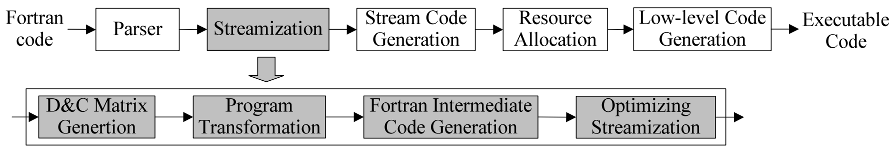


Fig. 8 The framework of SCompiler

based on the column transformation of the original program's D&C Matrix. Therefore, the generated stream program can achieve high locality in LRF and SRF, high computational intensiveness, and fine kernel granularity to exploit ample parallelism.

3.1.1 Loop reordering

The producer-consumer locality in SRF is exposed by forwarding the streams produced by one kernel to the subsequent kernels [15]. In order to make the neighboring kernels be provided with reused streams, the relative order of the corresponding loops in the D&C Matrix needs to be reordered for high computational intensiveness and fine locality in SRF. For instance, shown in Fig. 4, loop L_3 is shifted up by loop reordering, in order to tighten the production and consumption of array b. Loop reordering must satisfy safety and profitability considerations. The safety refers to reordering loops can not violate ordering constraints implied by dependences. The profitability lies on the data reuse between loops after reordering loops. A minimal requirement for profitability is that loop reordering should not destroy the original data reuse in SRF. To guarantee safety and profitability, some ordering constraints need to be proposed.

First, we should reorganize data reuse between loops, which is the basis of loop reordering. Since there is a data reuse from loop L_i to loop L_j when dependences exist between the two loops, we can analyze the D&C Matrix of the original program to obtain the reuse between loops. Concretely, for loop L_i and L_j, if $m_{xi} \cap m_{xj} \neq \phi$, we say that L_i and L_j reuse array D_x. Thus, the determination of data reuse between L_i and L_j (denoted as $L_i \delta L_j$) can be concluded as the following formula. That is, $L_i \delta L_j$ when both of them access a certain array (D_x)

$$\exists x (m_{xi} \cap m_{xj} \neq \phi) \rightarrow L_i \delta L_j \tag{2}$$

As shown in Fig. 7, since $m_{11} \cap m_{12} \neq \phi$ and $m_{21} \cap m_{23} \neq \phi$, we can conclude $L_1 \delta L_2, L_1 \delta L_3$ according to formula (2). It is convenient to characterize loop reuse by the distance between loops. Thus, we define the reuse distance between loops.

Definition 3 Suppose that there is a reuse between loop L_i and L_j, the reuse distance $d(i, j)$ is defined as the total length of all arrays in the two loops, namely $d(i, j) = \sum \mathrm{len}(D_x) | m_{xi} \cup m_{xj} \neq \phi$, where $\mathrm{len}(D_x)$ denotes the length of array D_x.

Second, ordering constraints are proposed based on the definition of exchangeable loops.

Definition 4 The neighboring loops L_i and L_j are defined as exchangeable loops if there is no data reuse between the two loops. The exchangeable loops are denoted as $L_i \leftrightarrow L_j$.

As shown in Fig. 7, $L_2 \leftrightarrow L_3$. Obviously, reordering the exchangeable loops can not influence the reuse order in the original program. This is the primary safety determination of loop reordering. But it is not sufficient to guarantee profitability. The requirement for profitability is that reordering should not increase the original reuse

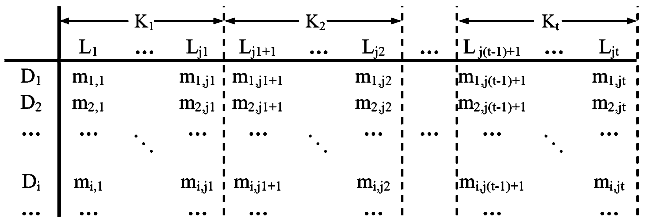

K1
K2
Kt
L1 … Lj1
Lj1+1 … Lj2
…
Lj(t-1)+1 … Ljt
D1 m1,1 m1,j1 m1,j1+1 m1,j2 m1,j(t-1)+1 m1,jt
D2 m2,1 m2,j1 m2,j1+1 m2,j2 m2,j(t-1)+1 m2,jt
Di mi,1 mi,j1 mi,j1+1 mi,j2 mi,j(t-1)+1 mi,jt

Fig. 9 Kernel partition based on the new matrix

distance, or the loop reordering is not profitable. Thus, the definition of potentially adjacent loops is proposed.

Definition 5 Suppose loop L_i, L_j, L_k are executed in serial. If $L_i \delta L_k$ and $L_j \leftrightarrow L_k$, L_j and L_k are defined as potentially adjacent loops. Meanwhile, the reuse between L_j and L_k are defined as potentially adjacent reuse, which is denoted as $L_i \overset{\leftrightarrow}{\delta} L_k$.

Loop reordering is used to transform the data reuses in the original program to potentially adjacent reuses, so as to improve the computational intensiveness and data reuse between loops. That is, we perform potentially adjacent reuse guided loop reordering on the original program. As for the above example, $L_1 \overset{\leftrightarrow}{\delta} L_3$. Thus, we need to reorder the loops in the program.

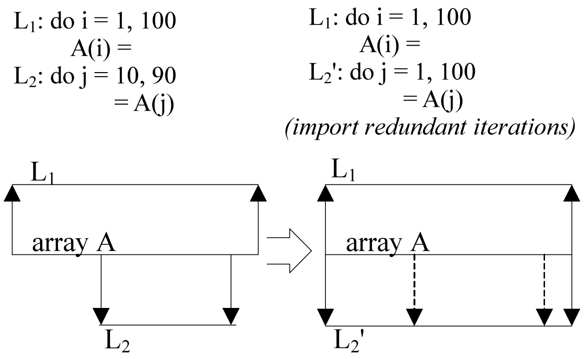
3.1.2 Loop fusion

To enhance computational intensiveness, we merge different regions about the same array into a large loop based on the D&C Matrix, and thereby temporal locality in the generated kernel can be increased and the memory delays can be overlapped with computations easily. On the other hand, we need to combine loops into larger loops as many as possible to enlarge the kernel granularity, and thus the parallelism in kernels can be increased via fully utilizing multi-ALUs on FT64. To implement this optimization, we perform matrix distribution and matrix fusion on the loops. Arbitrary loop L_i and L_j are fused when $L_i \delta L_j$.

Then a new D&C Matrix with fine loop granularity is yielded, and we can partition all loops into t kernels denoted K_t, respectively, according to the new matrix, which is diagrammed in Fig. 9. For example shown in Fig. 4, loop fusion gives a common iteration space where the consumption of a value $b[i]$ can be made nearer from its production, so as to improve computational intensiveness and loop granularity. To increase the opportunities for this optimization, loop distribution is sometimes applied to extract perfect loop nests from an imperfect nesting.

3.1.3 Array transformations between loops

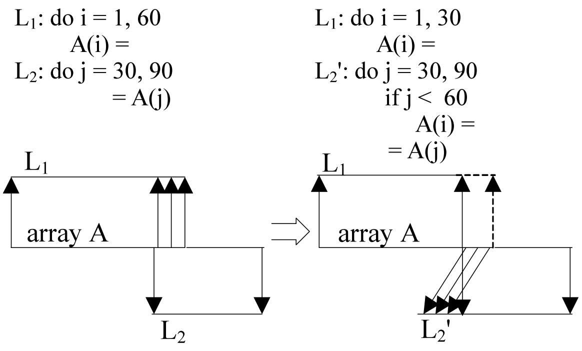
First, to achieve the stream reuse between successive kernels, we need to alter the arrays' region to unify the arrays in successive loops. The idea given in Fig. 10 emphasizes on adding or reducing some additional data of some arrays with the variety of the corresponding computations.



```
L1: do i = 1, 100
        A(i) =
L2: do j = 10, 90
            = A(j)
```

```
L1: do i = 1, 100
        A(i) =
L2': do j = 1, 100
            = A(j)
(import redundant iterations)
```

Fig. 10 Unifying arrays between loops



```
L1: do i = 1, 60
        A(i) =
L2: do j = 30, 90
            = A(j)
```

```
L1: do i = 1, 30
        A(i) =
L2': do j = 30, 90
        if j <  60
            A(i) =
        = A(j)
```

Fig. 11 Instruction scheduling

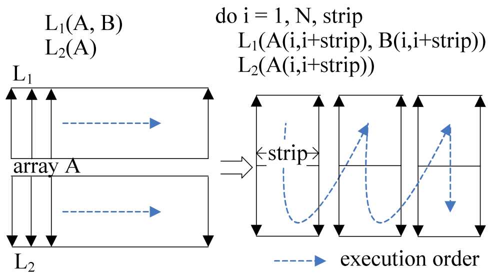
Then we can transfer some loop instructions in the previous loop to the next loop, when this loop exhibit data dependency with next loop, which is shown in Fig. 11. This idea can reduce the production of intermediate results to guarantee SRF capacity enough and enhance SRF reuse. The essence of this transformation is to distribute the loops in different kernels and then to fuse partial loops to a kernel based on data-centric analysis.

3.1.4 Strip-mining

Most scientific programs often face the same memory access bottleneck for they can't fit all their streams in the SRF, which harms the SRF locality. To address the problem, it is necessary to partition long streams into segments known as strips, such that all of the intermediate state for the computation on a single strip fits in the SRF [10], thus avoiding the memory transfers. In order to implement this optimization, we perform strip-mining on the original programs to enhance part reuse between loops [18]. An example is shown in Fig. 12, suppose array A is larger than SRF, strip-mining is used here to reuse strips of array A so that the generated stream program can achieve high locality in SRF.

To improve the reuse degree in SRF, we need to first determine the loop set for strip-mining. This loop set must benefit from strip-mining operation. So it is may be

Fig. 12 Strip-mining



the maximal set that consists of potentially adjacent loops. We define the loop set for strip-mining as follows.

Definition 6 Arbitrary potentially adjacent loop set can be used as a loop set for strip-mining. And if the intersection of two loop sets for strip-mining is not empty, the coalition of the two sets is also a loop set for strip-mining.

After data-centric loop reordering, the loop sets for strip-mining are relatively intensive. To determine the optimum loop set for strip-mining, reuse distance is used as the metric because it reflects the possibility of reuse in SRF. The maximal reuse distance in the optimum loop set for strip-mining must be smaller than SRF. Thus, the optimum loop set is the maximal one in which all the reuse can be optimized by strip-mining.

Selecting the optimal strip size is also a crucial optimization for strip-mining technique. Smaller strip sizes are also advantageous, because they can make full use of all the memory access components, reduce the startup and finishup time of the corresponding stream programs, and overlap the computation time with memory time efficiently.

3.2 Fine-grained program transformations

Fine-grained program transformations refer to reordering computations and data inside loops by shortening the computation distances and data distances in the D&C Matrix. This process aims at improving locality within loops and optimizing data references. Therefore, the generated stream programs can achieve high locality in kernels, high computational intensiveness, and low memory overhead of derived streams. Note that since all the clusters work as SIMD fashion, the locality in kernels focuses on enhancing the affinity for records in LRF and making a computation sequence access the same record, which is unlike the traditional cache locality that aims at centralizing all data referenced by the same computation.

3.2.1 Enhancing spatial locality inside loops

The spatial locality inside loops presents the spatial locality in the generated kernels, namely the affinity for records in LRF. Since LRF can't make random access through index support, the spatial locality inside loops must be successive and limited to the

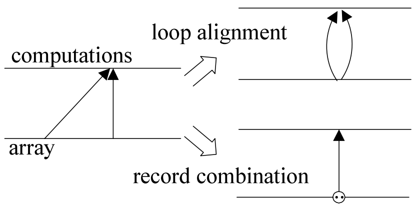


Fig. 13 Enhancing LRF spatial locality

capacity of LRF and the overhead caused by SPs. If an iteration operates on different array elements with large data distance, the spatial locality is lost. In order to reference adjacent records simultaneously, we need to shorten the data distance in the D&C Matrix through reordering the data accessed by the same computation as the following formula, where a is an arbitrary element in the ith row array D_i

$$\forall i \forall j \left(\forall a \in D_i \left(m_{ij}(a) \cap m_{ij}(a \pm 1) \neq \phi\right)\right) \tag{3}$$

1. Loop alignment

The approach of loop alignment [18] is to align different data to the same computation by adding extra iterations and adjusting the indices of one of the statement. So the data distance can be reduced to achieve fine spatial locality in the loop. For the example in Fig. 13, each iteration of the loop produces values $b(i)$, $c(i)$, $d(i)$ and uses values $b(i-1)$, $a(i)$ and $a(i+1)$. Value a and b are produced at each iteration and must be kept until their last use by another statement of the loop. Here loop alignment is used to optimize the use of the arrays a and b. We can see that the values of the array a and b are consumed as soon as they are produced, and the spatial locality of the generated stream program is enhanced. Figure 13 shows data distance is reduced by performing loop alignment.

2. Combining elements as a record

All items of a record are placed on a cluster to perform the same computations. So the spatial locality can be improved by combining elements referenced by the same computation as a big record as shown in Fig. 13. At the same time, we must claim attention to save the array boundary of the big record because the record may be as large as the capacity of LRF [19]. This idea can avoid assigning dependent data within an iteration to different clusters and make full use of LRF.

3.2.2 Improving temporal locality inside loops

Improving temporal locality inside loops aims at achieving high temporal locality in the corresponding kernels of generated stream programs. The temporal locality is enhanced if each array element is accessed many times successively inside loops. We consider reducing the computation distance in the D&C Matrix as the following formula by computation reordering to improve the temporal locality, where x is an arbitrary iteration in the jth column loop L_j

$$\forall i \forall j \left(\forall x \in L_j \left(m_{ij}^{-1}(x) \cap m_{ij}^{-1}(x \pm 1) \neq \phi\right)\right) \tag{4}$$

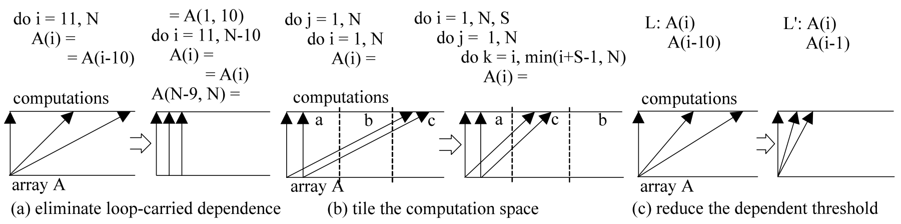


Fig. 14 Enhancing temporal locality inside loops

1. Eliminating the loop-carried dependences

Data dependence tells us that two references point to the same LRF location. Thus, the computation distance can be shortened by eliminating the loop-carried dependence shown in Fig. 14a through array expansion, code replication, etc. [17], and making dependence just exists within inner loops.

2. Tiling the computation space

If the loop-carried dependence between loops of a long stream can't be converted into the loop-independent dependence, we consider tiling the computation space [18] given in Fig. 14b for reducing the computation distance. Above all, we need partition the computation space to several parts. Then change the order of these parts to shorten the computation distance between the parts. So, the size and the order of these computation parts play an important role in the LRF temporal locality.

3. Reducing the dependent threshold in the inner loop

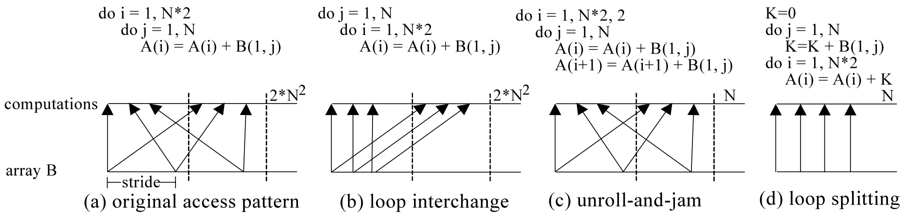
In loops, the variables produced by the previous iteration are assigned to SPs for the usage of the latter iteration. So, the allocation and usage of SPs are important for enhancing temporal locality in kernels. We formulize the number of SPs kept before iteration y as follows, where $NUM(a)$ denotes the number of different values of variable a

$$\sum NUM(a) \mid \forall i(\forall z > y)\big(\exists a \in m_{ij}^{-1}(z)\big(a < \max\big(m_{ij}^{-1}(y)\big)\big)\big) \tag{5}$$

To improve temporal locality in kernels, the fewest SPs are required to hold the values between the source and sink of the dependence to compact the affinitive computations, that is, minimize $\sum NUM(a)$. To reduce SP overhead, we must reduce the dependent threshold of inner loop shown in Fig. 14c, which denotes how many SPs would be allocated.

3.2.3 *Optimizing array references*

The usage of derived streams makes stream organization flexibly, but it also brings too much extra overhead of stream reordering and reloading. So, we must reduce the amount, length and stride of derived streams to lessen the pressure of off-chip memory. Since array references are transformed to derived streams, some necessary loop transformations [18] are introduced to optimize array references to achieve optimal derived streams.



```
do i = 1, N*2
   do j = 1, N
      A(i) = A(i) + B(1, j)
```

```
do j = 1, N
   do i = 1, N*2
      A(i) = A(i) + B(1, j)
```

```
do i = 1, N*2, 2
   do j = 1, N
      A(i) = A(i) + B(1, j)
      A(i+1) = A(i+1) + B(1, j)
```

```
K=0
do j = 1, N
   K=K + B(1, j)
do i = 1, N*2
   A(i) = A(i) + K
```

Fig. 15 Optimizing array references

1. Loop interchange

We can apply loop interchange to shorten the reference stride according to the reference pattern of the basic stream. For example, as shown in Fig. 15a, suppose array B in the program is stored in column-major order. Obviously, the iterations of innermost loop perform computations on the rows of array B, thus produce large reference stride. In order to shorten the stride of generated derived stream of B, we need to perform loop interchange so that the innermost loop is striding over the contiguous dimension, as shown in Fig. 15b.

2. Unroll-and-jam

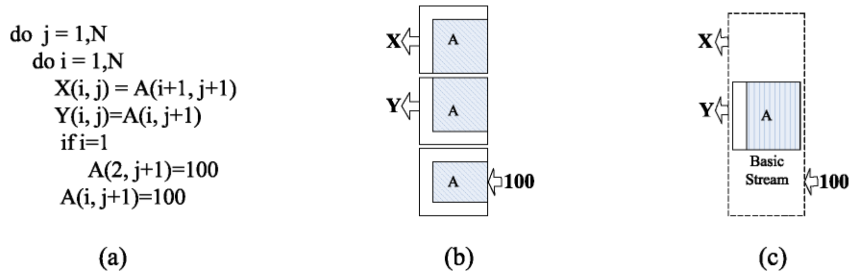
Performing unroll-and-jam can reduce the length of derived streams by improving computations per record. The essence of this transformation is to unroll the outer loop to multiple iterations and then to fuse the copies of the inner loop. As an example, consider the loop in Fig. 15c. By performing this transformation, the new version of the loop performs only one load of $B(1, j)$ for each two references. Therefore, the derived streams of B are shortened half length from 2^*N^2 to N^2 so as to reduce the loading overhead of the derived stream.

3. Data-centric loop splitting

To achieve higher performance than unroll-and-jam, we propose a new transformation to eliminate derived streams, namely data-centric loop splitting. We distill the computations that reuse data with large temporal span as self-governed loop. As previous example, the multiple loop can be split into two loops with computations on B and A, respectively, due to the discontinuous temporal reuse of $B(1, j)$. Thus, the kernel can use the basic streams of B and A without derived overhead as shown in Fig. 15d.

3.3 Stream organization optimization

After performing the program transformations above, an optimal intermediate code is produced. Then the optimal intermediate code needs to be mapped to a stream program through organizing kernels and streams effectively. Fine kernels can be generated by the above program transformations. However, efficient stream organization can not be achieved by one-to-one mapping between array and streams. In other words, every array variable can not be directly transformed to a basic stream, and every array reference can not be independently mapped to a derived stream. Therefore, as shown in Fig. 4, some necessary optimizations of stream organization need



```
do j = 1,N
   do i = 1,N
      X(i, j) = A(i+1, j+1)
      Y(i, j)=A(i, j+1)
       if i=1
          A(2, j+1)=100
       A(i, j+1)=100
```

Fig. 16 An example of selecting basic stream

to be introduced to the streamization process. We should firstly analyze the array reference pattern based on the mapping function of $m_{ij}^{-1}(x)$. Then according to the analysis, we transform effective array regions of selected arrays as basic streams, and produce derived streams after reusing different references on the same array, so as to reduce the overhead of derived streams and improve the locality in SRF.

3.3.1 Basic stream selection

This optimization focuses on choosing appropriate arrays as basic streams. In order to reduce the MEMORY access overhead and avoid maintaining storage consistency, we need to select successive basic streams as operation objects of kernels [15]. For example, if the basic stream D is organized as Fig. 6, it needs to derive a derived stream as (c, e, c, d) which presents large derived stride; while if the basic stream D is organized as (c, e, d), the derived stream is organized as relatively small stride to reduce the overhead of off-chip DRAM reordering.

Though an array may be referenced many times in different reference pattern, its corresponding basic stream will be generated as one copy. Figure 16a gives an example program and Fig. 16b presents the corresponding data layout of array A which is achieved from the D&C Matrix. After performing basic stream selecting on the program, three data reference pattern will generate a same basic stream shown in Fig. 16c, which avoids unnecessary stream loading and storage overhead.

By analyzing the reference pattern and access region of all arrays based on the D&C Matrix, the basic streams are organized according to the least common array region of high access frequency. We formulate the basic stream layout of each array as follows, where $BAS(i)$ denotes the basic stream layout of the ith row array in the D&C Matrix, f represents the time-consuming factor involving the invoking frequency and the running time, which shows the importance of each loop for deciding the basic stream layout

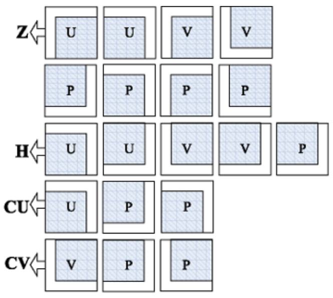
$$BAS(i) = \forall j\left(\cap\left(ORG(i, j) \cdot / f\right)\right) \tag{6}$$

3.3.2 Derived stream generation

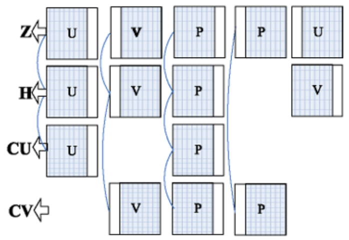
After selecting the required basic streams, we need to explore how to distribute these arrays on clusters, that is, derived stream generation. Some array reference optimizations are introduced in Sect. 3.2.3, which reorder the arrays with supplementing and

```
do j = 1, N
   do i = 1, M
      Z(i,j)=F_z (V(i+1,j+1),V(i,j+1),U(i+1,j+1),U(i+1,j),
               P(i,j),P(i+1,j),P(i+1,j+1),P(i,j+1))
      H(i,j) =F_H (P(i,j),U(i+1,j),U(i,j),V(i,j+1),V(i,j))
      CU(i+1,j) =F_cu(P(i+1,j),P(i,j),U(i+1,j))
      CV(i,j+1) =F_cv(P(i,j+1),P(i,j),V(i,j+1))
```

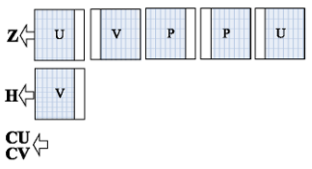
(a)



(b)



(c)



(d)

Fig. 17 Stream enlarging and stream reusing

updating the variant boundary by using dependence analysis and program transformation techniques. Based on the optimized array references, we need to reuse the arrays and reduce the derived streams as many as possible. First, unifying different derived streams by stream enlarging can combine the streams referenced by the same computations as a long stream. Then if there is diversiform reference pattern and access region of a stream in a kernel, we need to reuse the common block. The process is denoted as stream reusing.

We explicate stream enlarging and stream reusing in detail according to an example modified from a scientific application Swim in SPEC 2000. Figure 17a shows the example that is a perfect loop nest and performs multiform reference pattern on data A. Figure 17b presents the data layout of A in the original loop. After stream enlarging, the access overhead of A is reduced by 26% (5/19), as shown in Fig. 17c. With go on performing stream reusing on data linked by the same curve in Fig. 17c, the access overhead is reduced by 57% (8/14) again and the number of input parameters is reduced from 19 to 6, as shown in Fig. 17d.

4 Experimental results and analysis

To validate the effectiveness of our optimizing streamization approach, we perform tests on 10 typical scientific application kernels as specified in Table 1. Nlage-5 is a nonlinear algebra solver of two-dimensional nonlinear diffusion of hydrodynamics, and Transp is the time-consuming subroutines in Capao that is an optics application. All the programs are Fortran versions, and they are compiled by three kinds of

Table 1 Specifications of 10 benchmarks

Name	Swim	EP	MG	CG	DFFT	Laplace	Jacobi	GEMM	NLAG-5	Transp
Source	Spec2000	NPB	NPB	NPB	–	NCSA	–	BLAS	–	
#Arrays	14	1	3	2	1	1	4	2	2	5
Prob. size (doubles)	513×513	131072	$64 \times 64 \times 64$	500×500	4096	256×256	128×128	256×256	256×256	512×512

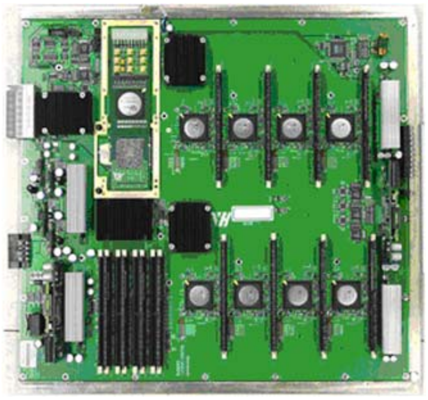


Fig. 18 The FT64 development board

Table 2 Comparison of different implementation for the scientific programs

Tests	Swim	EP	MG	CG	DFFT	Laplace	Jacobi	GEMM	NLAG-5	Transp
Opti vs. Orig	0.98	2.55	1.35	0.10	8.01	2.41	1.04	1.97	0.97	2.32
Opti vs. Dire	2.62	1.18	3.23	1.21	1.79	2.75	4.65	5.06	2.25	2.83

compilers, respectively, including Intel's compiler *ifort* (version 9.0) with the optimization option *-O3*, *SCompiler* with *the direct streamization*, and *SCompiler* with *the optimizing streamization*. The first compiling results (denoted as *Orig*) are executed on a single-core Itanium 2 server. Itanium 2 runs at 1.6 GHz and the sizes of the caches are 16 KB for the L1 cache, 256 KB for the L2 cache, and 6 MB for the L3 cache. There is also a 4 GB off-chip memory with the bandwidth of 6.4 GB/s. The latter two results (denoted as *Dire* and *Opti*) are executed on one FT64 stream processor, which is lain on the FT64 development board shown in Fig. 18. The parameters of the FT64 processor are shown in Sect. 2.1.

The execution time is obtained by inserting the clock-fetch assembly instructions. If the data size of the program is small, we eliminate the extra overheads (such as system calls) by means of executing it multiple times and calculating the average time consumption. As I/O overheads are hidden in our experiments, the CPU time is nearly equal to the wall-clock time.

Table 2 illustrates the performance results of the optimizing streamization versions (*Opti)* compared with the direct streamization versions (*Dire*) and the original Fortran versions (*Orig*). It can be observed that compared with Itanium 2 system, our optimizing streamization approach achieves high speedup of 5 programs (EP, DFFT, Laplace, GEMM, and Transp), and provides comparable speedup of other 4 applications (Swim, MG, Jacobi, and NLAG-5). This is because our approach can efficiently hide latency to achieve good performance of FT64, while Itanium 2 is highly sensitive to memory latency. As for CG's lowest speedup, it is because that its short arrays and irregular memory reference pattern can not be optimized efficiently by loop transformations. Table 2 also shows that the optimizing streamization approach achieves an average speedup of 2.76 over the direct streamization approach. It is certain that our approach can efficiently exploit the tremendous potential of FT64 for scientific applications.

To evaluate our approach more amply, three key techniques proposed in Sect. 3 are evaluated respectively in the rest of this section.

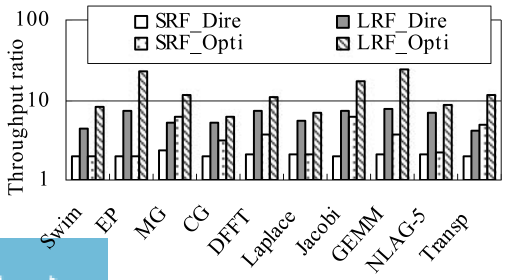
4.1 Evaluate coarse-grained program transformations

The purpose of coarse-grained program transformations is to achieve high locality in LRF and SRF, high computational intensiveness, and fine kernel granularity of the generated stream programs.

SRF (or LRF)-to-memory throughput ratio is the ratio of the data throughput in the SRF (or LRF) to that in the off-chip memory. It is used to measure the locality in SRF and LRF. Figure 19 shows the SRF (LRF)-to-memory throughput ratios with and without the coarse-grained program transformations during the streamization process. It can be observed that all the optimizing versions can achieve higher LRF-to-memory throughput ratios than that of the direct streamization versions. This is because the locality within loops of these programs can be efficiently enhanced by using the transformations among loops including loop reordering, loop fusion, etc., and thus the abundant memory access of the generated stream programs are focused on LRF. Besides, the optimizing versions of MG, GEMM, Jacobi, DFFT, CG, and Transp achieve higher SRF-to-memory throughput ratios than that of the direct streamization versions, which means that they well exploit SRF locality. This is because the strip-mining of the loops in MG, GEMM and Jacobi can achieve the data reuse, and the loop scheduling can be used to exploit the data reuse between loops of DFFT, CG and Transp. But the SRF-to-memory throughput ratios of optimizing versions of Swim, EP, Laplace, and NLAG-5 are comparable with that of the direct streamization versions, which show these programs' SRF only transfers the data from memory to LRF. Since the reused data have been moved together, there is no data reuse between kernels of these generated stream programs.

Figure 20 shows the effect on kernel size by applying our optimization compared with the direct streamization programs, as well as the computations per word (C/W) and the number of kernels (Kernels). We can observe that our optimization improves the kernel granularity of all the programs. But the kernel granularity of Swim achieves a little varying because it has huge data amount and irregular reference pattern, so that the loops are difficult to be distributed or fused. The other applications can enlarge the code amount of kernels obviously to centralize all the computations (array

Fig. 19 SRF- and LRF-to-memory throughput ratios



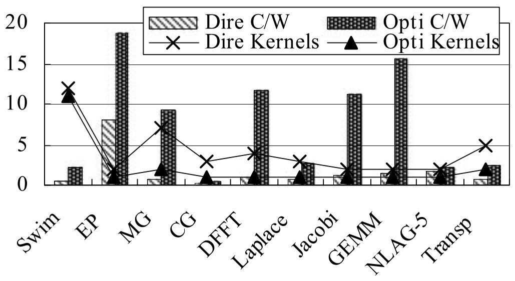


Fig. 20 The variety of kernel size

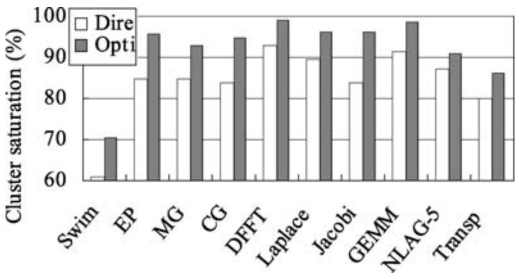


Fig. 21 Cluster saturation

expanding, loop fusion, and loop scheduling are used here), and thus their computational intensiveness is enhanced sharply except Transp. Transp that involves two imperfectly loop nests applies loop distribution and loop fusion by array expanding effectively, however, all arrays in Transp are referenced rarely leading a little variety of computational intensiveness.

4.2 Evaluate fine-grained program transformations

Fine-grained program transformations are used to improve the locality in LRF, increase the computational intensiveness, and optimize the derived streams of generated stream programs.

For improving the efficiency of the program, clusters are expected to be as busy as possible. Figure 21 shows the cluster saturation for these applications with and without fine-grained program transformations, which shows the saturation of the functional units during stream processing influenced by the locality enhancement. Cluster saturation depends on cluster's stall time. When plenty of memory accesses hit the LRF and the computations in clusters are abundant, the kernels need not wait for the supply of required streams so that stall will hardly occur. Obviously, all the programs achieve large increment of the utilization of overall ALUs by applying our streamization approach compared with the direct streamization versions. Because the locality in LRF and the computational intensiveness of these programs are enhanced by reordering computations and data within loops, such as reducing the dependent threshold, eliminating the loop-carried dependence, loop tiling, etc. As for the lowest cluster saturation of Swim, it is because its streams are in large size, and thus there are double buffers in SRF and the memory overhead can't be overlapped with computations. The problem cannot be solved by fine-grained program transformations.

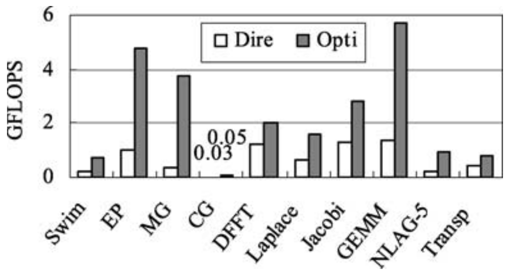


Fig. 22 Computation rate

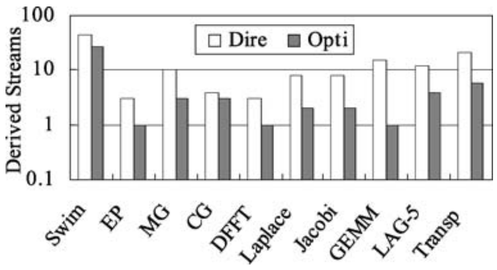


Fig. 23 The reduction of derived streams

Figure 22 presents the computation rate of the stream applications measured in GFLOPS. FT64's peak performance can achieve 16GFLOPS. The results show that the sustained performance of all the optimizing stream programs except CG reaches 4% to 36% of the peak performance, but the original stream programs only reach 1% to 9% of the peak performance. This is because the computational intensiveness can be increased and the memory access overhead can be reduced by introducing the fine-grained program transformations, such as reducing the computation and data distance in loops and array references optimizations. However, Swim and Transp still achieve a little performance improvement because the fine-grained program transformations don't relate to the basic stream optimizations, so that there still exist overfull derived streams in these programs.

4.3 Evaluate stream organization optimization

The stream organization optimization focuses on selecting optimum basic streams and enabling stream reusing, so as to reduce the overhead of derived streams and improve data reuse.

Figure 23 shows the reduction of derived streams with and without stream organization optimization in streamization process. It is obvious that the derived streams of all the optimizing versions are reduced, which validates the effectiveness of our optimization for stream organization. For the large reduction of derived streams of Jacobi, GEMM, and Transp, it is because their basic streams are effectively selected according to the practical array reference pattern. For the large reduction of derived streams of MG, Laplace and NLAGE-5, it is because stream reusing is used to reuse the common region of diversiform reference pattern. On the other hand, EP and DFFT have a few data in the direct streamization versions, so the derived streams are lessened a

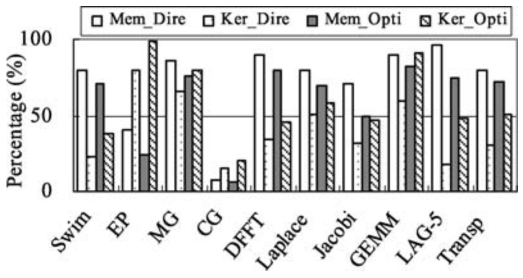


Fig. 24 Percentages of memory access and kernel execution

little too. In Swim and CG, the choice of basic stream has little effect on stream organization owing to the complex data reference pattern, and thus the number of derived streams is also reduced a little.

The overlap between computation and memory access is another important factor that impacts a stream processor's performance. Figure 24 demonstrates the distribution of memory access time and kernel execution time when programs with (denoted as Mem_Opti and Ker_Opti) and without (denoted as Mem_Dire and Ker_Dire) stream organization optimizations running on FT64, i.e., what percent the two parts take up as to the total program execution time. Obviously, all the programs except CG with the stream organization optimizations achieve less memory access proportion and more kernel execution proportion compared with that of the direct streamization versions. Especially, the memory access time and the kernel execution time of MG, Laplace, Jacobi, and GEMM are comparable. It is certain that selecting optimum basic streams and reusing derived streams can reduce the memory access overhead, and thus lead to well hiding the memory access overhead with computation time. For CG's lowest total proportion of memory access and kernel execution, it is because most of the time is consumed in SRF allocation and memory access preparation due to its irregular memory reference pattern, which cannot be efficiently optimized by the stream organization optimization.

5 Related work

Many media applications for stream processing have been previously studied, such as stereo depth extraction, MPEG-2 encoding, QR decomposition, space-time adaptive processing, polygon rendering, FFT, convolution, DCT, and FIR [5–7]. Though media applications are becoming the dominate consumer of stream processors, there is an important effort to research whether scientific applications are suited for stream processors, so as to fully exploit their powerful processing ability. Examples including efficient fluid flow simulation and iterative solvers for sparse linear systems [20–22] have been demonstrated to run on GPU, which is a graphic stream processor. Many linear algebra routines and scientific applications have been mapped to the Merrimac supercomputer that is also stream architecture [23–26]. Some dense and sparse matrix applications and some mathematic algorithm such as transitive closure have been implemented on Imagine [27]. However, the prior studies focused on how these idiographic applications were expressed as stream programs. There is little research on general automatic generation of stream programs on stream architectures. Furthermore, some existing researches on memory optimization for stream architectures mostly concerned partial hardware melioration [28] and common cache hierarchy optimization [29]. There are few studies on stream program optimizations through programming approach based on managing the memory hierarchy and processing unites in stream processors. Paper [19] developed some programming optimizations for mapping scientific programs to Imagine. Our work is a further effort to explore the systemic automatic streamization approach for scientific programs to improve locality and parallelism on FT64, which is the first 64-bit stream processor for scientific computing.

The optimization in this paper focuses on computation reorder and data layout restructure. The classical computation optimizations [30–32] and data layout optimizations [33–35] have introduced in traditional parallel compiling technique. However, the prior transformations aimed at optimizing latency-oriented cache hierarchy. To implement optimization for bandwidth-oriented stream hierarchy, our work on locality and parallelism enhancement is different from those mentioned. Because it is necessary to design special optimizations to exploit the hardware performance, such as the clusters that run as SIMD pattern and the registers without indexed access.

This paper abstracts the relationship between loops and arrays as a matrix to formulate the streamization problem. In the prior works, some authors also propose a few intermediate structures used to improve locality, such as the layout graph [36, 37], the data transformation matrix [38] and the communication-parallelism graph [39], etc. However, these researches just considered data transformations or iteration transformations within loops without exploring the relation between loops. In comparison, we concentrate on the D&C Matrix for locality and parallelism enhancement combining loop transformations and data transformations, and especially the transformations between columns in the matrix can present the potential of program restructure for computational intensiveness.

Therefore, to our knowledge, our work is the first study on the automatic generation of scientific stream programs on FT64 through combined computation and data transformation to improve locality and parallelism.

6 Conclusion and future work

In this paper, we have presented a novel matrix-based streamization approach for improving locality and parallelism to exploit the powerful processing ability of FT64. Our specific contributions are as follows. We firstly formulate the problem on the Data&Computation Matrix (D&C Matrix) that is proposed to abstract the relationship between loops and arrays. Furthermore, we propose the key techniques for optimizing streamization based on the matrix, including program transformations and stream organization optimizations. Our approach is simple and generates stream programs for ten scientific application kernels in our experiment. The experimental evaluation shows that our optimizing streamization approach can effectively enhance the locality in LRF and SRF, and improve the instruction and data parallelism of generated stream programs on FT64. With the effort in this work, a great deal of scientific applications can be easily mapped to FT64 stream processor.

In the future, our efforts will mainly focus on two aspects. First, we plan to explore more streamization optimizations such as branch and reduction to exploit more architectural features of FT64, so that our optimizing streamization can achieve higher performance and wider applicability. Second, we would like to search more scientific applications suited for FT64 by applying our optimization.

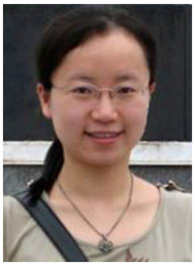
Acknowledgements We would like to thank Lab 610 of School of Computer Science in National University of Defense Technology for their efforts on this work. We also acknowledge the reviewers for their insightful comments. This work was supported by NSFC (60621003).

References

1. Kapasi UJ, Rixner S, Dally WJ et al (2003) Programmable stream processors. IEEE Comput 54–62
2. Khailany B (2003) The VLSI implementation and evaluation of area-and energy-efficient streaming media processors. Ph.D. thesis, Stanford University
3. Taylor M, Kim J, Miller J et al (2002) The RAW microprocessor: a computational fabric for software circuits and general purpose programs. IEEE Micro 22(2):25–35
4. Burger D, Keckler SW, McKinley KS et al (2004) Scaling to the end of silicon with EDGE architectures. Computer 37(7):44–55
5. Gordon MI, Thies W, Amarasinghe S (2006) Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Proceedings of ASPLOS'06, California, USA
6. Andrew AL, Thies W, Amarasinghe S (2003) Linear analysis and optimization of stream programs. In: Proceedings of the SIGPLAN'03 conference on programming language design and implementation, San Diego, CA
7. Owens JD, Rixner S et al (2002) Media processing applications on the imagine stream processor. In: Proceedings of the 2002 international conference on computer design
8. Yang X, Yan X, Xing Z et al (2007) A 64-bit stream processor architecture for scientific applications. In: ISCA'07: Proceedings of the 34th annual international symposium on computer architecture. ACM Press, New York, pp 210–219
9. Amarasinghe S et al (2003) Stream languages and programming models. In: Proceedings of the international conference on parallel architectures and compilation techniques 2003
10. Mattson P (2002) A programming system for the imagine media processor. Ph.D. thesis, Dept of Electrical Engineering, Stanford University
11. Du J, Yang X et al (2007) Architecture-based optimization for mapping scientific applications to imagine. In: ISPA'07: Proceedings of the 2007 international symposium on parallel and distributed processing with applications, Ontario, Canada
12. Das A, Dally WJ, Mattson P (2006) Compiling for stream processing. In: PACT'06: Proceedings of the 15th international conference on parallel architectures and compilation techniques. ACM Press, New York, pp 33–42
13. Johnsson O, Stenemo M, ul-Abdin Z (2005) Programming & implementation of streaming applications. Master's thesis, Computer and Electrical Engineering Halmstad University
14. Ahn JH, Dally WJ et al (2004). Evaluating the imagine stream architecture. In: Proceedings of the annual international symposium on computer architecture 2004
15. Jayasena NS (2005) Memory hierarchy design for stream computing. Ph.D. thesis, Stanford University
16. Wolf ME, Lam M (1991) A loop transformation theory and an algorithm to maximize parallelism. IEEE Trans Parallel Distrib Syst 2(4):452–471
17. Kuck D, Kuhn R et al (1981) Dependence graphs and compiler optimizations. In: Conference record of the eighth annual ACM symposium on the principles of programming languages, Williamsburg, VA, January 1981
18. Wolfe MJ (1996) High performance compilers for parallel computing. Addison-Wesley, Reading
19. Du J, Yang X et al (2006) Scientific computing applications on the imagine stream processor. In: Proceedings of the 11th Asia-pacific computer systems architecture conference, Shanghai, China
20. Fan Z, Qiu F et al (2004) Gpu cluster for high performance computing. In: Proceedings of supercomputing conference 2004

21. Harris MJ, Baxter WV et al (2003) Simulation of cloud dynamics on graphics hardware. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware, Switzerland, pp 92–101
22. Bolz J, Farmer I, Grinspun E, Schr Öder P (2003) Sparse matrix solvers on the Gpu: conjugate gradients and multigrid. ACM Trans Graph 22(3):917–924
23. Dally WJ, Hanrahan P et al (2003) Merrimac: supercomputing with streams. In: Proceedings of supercomputing conference 2003
24. Erez M, Ahn J et al (2004) Analysis and performance results of a molecular modeling application on Merrimac. In: Proceedings of supercomputing conference 2004
25. Erez M (2007) Merrimac—high-performance, highly-efficient scientific computing with streams. Ph.D. thesis, Dept of Electrical Engineering, Stanford University
26. Erez M, Ahn J et al (2007) Executing irregular scientific applications on stream architectures. In: (ICS'07): Proceedings of the 21th ACM international conference on supercomputing
27. Griem G, Oliker L (2003) Transitive closure on the imagine stream processor. In: Proceedings of the 5th workshop on media and streaming processors, San Diego, CA
28. Ahn J, Dally WJ, Erez M (2007) Tradeoff between data-, instruction-, and thread-level parallelism in stream processors. In: (ICS'07): Proceedings of the 21th ACM international conference on supercomputing
29. Sermulins J, Thies W et al (2005) Cache aware optimization of stream programs. In: Proceedings of LCTES'05, Chicago, Illinois, USA
30. Wolf M, Lam M (1991) A data locality optimizing algorithm. In: Proceedings of ACM SIGPLAN'91 conference on programming language design and implementation, Ontario, Canada, pp 30–44
31. McKinley K, Carr S, Tseng CW (1996) Improving data locality with loop transformations. ACM Trans Program Lang Syst
32. Li W (1993) Compiling for NUMA parallel machines. Ph.D. thesis, Cornell University
33. Kandemir M, Choudhary A et al (1999) A linear algebra framework for automatic determination of optimal data layouts. IEEE Trans Parallel Distrib Syst 10(2):115–135
34. Cierniak M, Li W (1995) Unifying Data and control transformations for distributed shared memory machines. In: ACM SIGPLAN IPDPS, pp 205–217
35. Kandemir M, Choudhary A et al (1998) Improving locality using loop and data transformations in an integrated framework. In: Proceedings of international symposium on microarchitecture, pp 285–297
36. Kandemir M, Banerjee P et al (2001) Static and dynamic locality optimizations using integer linear programming. IEEE Trans Parallel Distrib Syst 12(9):922–940
37. Kandemir M et al (1999) A graph based framework to detect optimal memory layouts for improving data locality. In: Proceedings of the 13th international parallel processing symposium, San Juan, Puerto Rico, pp 738–743
38. O'Boyle M, Knijnenburg P (1996) Non-singular data transformations: definition, validity, applications. In: Proceedings of 6th workshop on compilers for parallel computers, pp 287–297
39. Garcia J, Ayguade E et al (1996) Dynamic data distribution with control flow analysis. In: Proceedings of supercomputing conference 1996

Xuejun Yang received his M.Sc. and Ph.D. degree in computer science from the National University of Defense Technology (NUDT), China, in 1986 and 1991, respectively. He is a Full Processor and the head of the School of Computer Science of NUDT. He is also the head of the Creative Compiler research group at NUDT. His research interest lies in high performance computing, parallel computer architecture, high performance compiler and operating system.



Jing Du received her B.Sc. degree in computer science from the National University of Defense Technology (NUDT), China, in 2002. Now she is a Ph.D. student in the School of Computer Science of NUDT. She is a member of the Creative Compiler research group at NUDT. Her research interests include high performance computing, parallel algorithms and porgramming.

Xiaobo Yan received his B.Sc. degree in computer science from the National University of Defense Technology (NUDT), China, in 2001. He is now a Ph.D. student in the School of Computer Science of NUDT. He is a member of the Creative Compiler research group at NUDT. His research interests include high performance computing, parallel computer architecture and compiler design.

Yu Deng received his B.Sc. and M.Sc. degree in computer science from the National University of Defense Technology (NUDT), China, in 2000 and 2003, respectively. He is now a Ph.D. student in the School of Computer Science of NUDT. He is a member of the Creative Compiler research group at NUDT. His research interests include high performance computing, parallel computer architecture and compiler design.