Exploiting Loop-Dependent Stream Reuse for Stream Processors

Xuejun Yang, Ying Zhang, Jingling Xue†, Ian Rogers‡, Gen Li and Guibin Wang
School of Computer, National University of Defence Technology, ChangSha, China
†The University of New South Wales, Sydney, Australia
‡The University of Manchester, Manchester, UK
{xjy, zhangying, genli, gbw}@nudt.edu.cn
jingling@cse.unsw.edu.au ian.rogers@manchester.ac.uk

ABSTRACT

The memory access limits the performance of stream processors. By exploiting the reuse of data held in the Stream Register File (SRF), an on-chip storage, the number of memory accesses can be reduced. In current stream compilers reuse is only attempted for simple stream references, those whose start and end are known. Compiler analysis from outside of stream processors does not directly enable the consideration of other complex stream references. In this paper we propose a transformation to automatically optimize stream programs to exploit the reuse supplied by loop-dependent stream references. The transformation is based on three results: algorithms to recognize the reuse supplied by stream references, a new abstract expression called the Stream Reuse Graph (SRG) to depict the reuse and the optimization of the SRG for the transformation. Both the reuse between whole sequences accessed by stream references and that between partial sequences are exploited in the paper. In particular, the problem of exploiting partial stream reuse does not have its parallel in the traditional data reuse exploitation setting (for scalars and arrays). Finally, we have implemented our techniques using the StreamC/KernelC compiler for Imagine. Experimental results show a resultant speedup of 1.14 to 2.54 times using a range of typical stream processing application kernels.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors, Compilers; D.3.2 [**Programming Languages**]: Language Classifications, Specialized application languages

General Terms

Algorithms, Design, Management, Performance, Experimentation

PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.

Keywords

Stream Programming Model, Stream Professor, Stream Register File, StreamC, Stream Reuse

1. INTRODUCTION

In the past with increasing clock frequencies, and now with an increasing number of cores, processing power is doubling every eighteen months. Memory access times have also been increasing but at a slower rate, leading to what has become to be known as the memory wall [28]. Architectural and software techniques, including reusing data on-chip and prefetching to hide long memory latency, have been applied to relieve the memory wall. The research of advanced processors now focuses on two ways: developing multi-core as well as multi-thread processors of conventional architectural models, and developing stream processors of novel architectural models, such as Imagine [14], Merrimac [9], Cell Processor [13], RAW [26] and FT64 [30]. Much research has been into reusing data on-chip of a conventional architectural model [3, 5, 18, 6, 24, 17, 29]; however, research into reusing data on-chip of a stream processor is a less mature research area.

1.1 Stream Processing

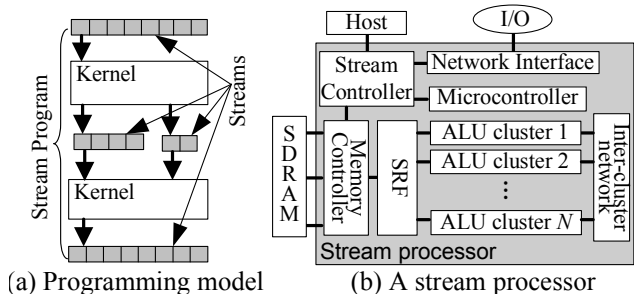


Figure 1: Stream processing.

The stream programming model [25, 15, 14, 30, 4, 9], shown in Fig. 1(a), divides an application into a *stream-level program* and one or more *kernels* that define each processing step. The stream-level program declares the streams and defines the high-level control- and data-flow between kernels. Each kernel is a function that consumes and produces elements from *streams*. A stream is a sequence of data records.

Popular languages implementing the stream programming model include StreamC/KernelC [23, 19] for Imagine and Merrimac processor [9], Brook [4] for GPU and SF95 [30] for FT64 processor. These languages can also be used to develop stream programs for Cell Processors [13]. All these stream architectures have the characteristic of SIMD stream coprocessors with a large local memory for stream buffering. Fig. 1(b) shows a simplified diagram of such a stream processor. A stream-level program is run on the host while kernels are run on the stream processor. A single kernel that operates sequentially on records of streams is executed on clusters of ALUs, in a SIMD fashion. Only data in the *local register files* (LRFs), immediately adjacent to the arithmetic units, can be used by the clusters. Data passed to the LRFs is from the Stream Register File (SRF) that directly access memory. Off-chip memory is used for application inputs, outputs and for intermediate streams that cannot fit in the SRF.

```
const int N = 256;
stream s(N*(N+2)),o(N*N);//declare basic streams s and o.

stream s0,s1,s2,s3,o0;     //declare variable-bound stream references.

for(int i = 0; i<N; i++){
  s0 = s(i     *N,(i+1)*N);
  s1 = s((i+1)*N,(i+2)*N);      //define variable-bound stream
  s2 = s((i+2)*N,(i+3)*N);      //references. These statements
  s3 = s((i+2)*N,(i+3)*N-N/2); //just define some pointers.
  o0 = o(i     *N,(i+1)*N);
  k(s3,s2,s1,s0,o0);        //call kernel.
}
```

Figure 2: Example 1, a stream program for StreamC/KernelC.

StreamC/KernelC [19] is a popular language that can be used to develop stream programs for Imagine [14], Merrimac [9], Cell Processor [13] and FT64 [30]. Additionally, the Brook language [4] is an extension of StreamC/KernelC. Fig. 2 shows a StreamC/KernelC program. Two basic streams, s, o and five stream references s_0, s_1, s_2, s_3, o_0 are defined. Each stream reference accesses a subset of the basic stream corresponding to its name. The stream reference is defined by a *start bound*, *end bound* and an optional *access mode*. The *access mode* defines which records between the start and end are part of the stream. In Fig. 2, all stream reference accesses are sequential. Stream references with constant start and end bounds are called *constant-bound stream references*; all other stream references are called *variable-bound stream references*. Among variable-bound stream references, the most popular type of references whose start and end bounds are functions of the index variables we call *loop-dependent stream references*. For example, the stream reference s_0 is a loop-dependent reference and accesses the $0th$, $1st$, ..., $(N-1)th$ records of the stream s in the first iteration. In this paper, we assume each kernel call only has a single output stream (the last argument).

1.2 Reusing Streams

Memory access still dominates most stream programs' performance [1], especially for scientific stream programs [21]. The only way to reduce off-chip memory bandwidth requirement is to exploit the reuse of streams in the SRF. Only the data defined by stream references is held sequentially in the software-managed SRF. The SRF location and length of the data are stored in a *stream descriptor register* (SDR). Stream operations specify what values they operate on by referring to appropriate SDR. Except stream transfers, all operations require that the operands be taken from the SRF and all results be assigned to the SRF.

For two stream references accessing the same values, the second reference can reuse the values that the first reference generated in to the SRF. We call the scenario *whole reuse*. For two stream references accessing the shared portion of values, the second reference can reuse this part of the values generated by the first reference in to the SRF. We call the scenario *partial reuse*. For partial reuse, stream compilers try to find the supersequence of the two stream references, then allocates a SRF buffer for the supersequence and allocates a SDR for each stream reference that defines which part of the buffer is accessed by the reference. For example, two input stream references $a_0 = a(32, 96)$ and $a_1 = a(64, 128)$ access the shared portion $a(64, 96)$, so partial reuse exists between them. The compiler finds their supersequence, i.e. $a(32, 128)$, allocates a SRF buffer for the supersequence and allocates two SDRs for the two references. One SDR describes that a_0 accesses from the $0th$ to $63th$ records of the supersequence; the second SDR describes that a_1 accesses from the $32th$ to $95th$ records. During execution, only 96, instead of 128, words are loaded.

The stream compilers utilize all reuse supplied by constant-bound stream references [19, 20]. However, when processing a variable-bound stream reference, the stream compiler generates a stream load before each read and a stream save after each write, without utilizing any reuse.

The problem addressed in this paper is to effect the reuse supplied by loop-dependent stream references held in the SRF. Some special consideration is required. Firstly, the SRF is managed by software, which brings the opportunity of partial reuse. Secondly, moving data within the SRF is expensive. The SRF is divided into lanes so that each lane supplies data to one cluster. If stream processors, such as stream processors with indexed SRF [12] and Cell Processor [13] allow communication across SRF lanes, inter-lane bandwidth is much lower than Lane-to-Cluster bandwidth; if not, off-chip memory is used as a relay to avoid inter-lane communication by generating SRF-to-memory and memory-to-SRF data moves. Therefore, SRF-to-SRF data moves should be avoided during the transformation. Finally, only the reuse with respect to the innermost loop is useful, due to the limited SRF capacity and the relatively large work set of kernels.

1.3 Our Solution

We describe an algorithm to perform a source-to-source transformation, called *constant-bound stream replacement*, which replaces loop-dependent stream references with constant-bound stream references to effect the reuse among loop-dependent stream references. Our transformation is based on three results: an algorithm recognizing reuse among stream references, a new model called *stream reuse graph* (*SRG*) and the optimization of the built SRG for the transformation.

Our stream reuse algorithm recognizes both whole and partial reuse. Our algorithm first identifies the reuse of locations and then judges whether the values to be reused will be unchanged before being reused. If the values are unchanged, then reuse is possible.

The second result is a model, *SRG*, which describes the whole reuse among stream references. During the SRG con-

struction, a *Directed Acyclic Graph* (*DAG*) is built first to describe how kernels consume input streams and produce output streams. Loop-independent reuse is also depicted by the DAG. Then, a *reuse graph* (*RG*) is built based on the DAG, by inserting a reuse edge with the value of $d_{i,j}(d_{i,j} \geq 0)$ from the stream node s_i to s_j if the values generated by s_i is used by s_j after $d_{i,j}$ iterations.

Finally, the built SRG is optimized for the transformation from the following two aspects. (1) A stream node in the built SRG may have multiple reuse sources, we prune the SRG so that every stream node has only one source for the following transformation. The stream node chosen as reuse source is called the *generator*. The concept is also used in [6]. (2) The SRG is expanded for partial reuse. The generator is updated to be the stream node that other stream nodes reuse part or whole of data generated by it before certain iterations.

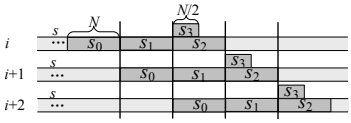


Figure 3: Relationship of the access locations between s_0, s_1, s_2 and s_3.

We use Example 1 from Fig. 2 to describe our solution. Fig. 3 shows the relationship of locations accessed by s_0, s_1, s_2 and s_3. The values generated by s_3 are reused as part of s_2 in the same iteration; the values generated by s_2 in iteration i are reused as s_1 after 1 iteration, reused as s_0 after 2 iterations. However, current stream compilers cannot recognize and utilize the reuse.

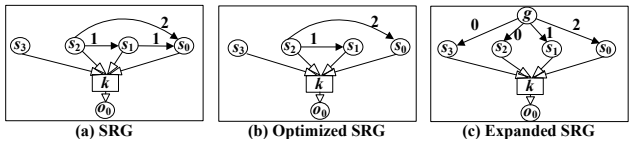


Figure 4: Building the SRG for Example 1 from Fig. 2.

Fig. 4 shows the graphs generated when building the SRG: Fig. 4(a) shows the SRG that describes whole reuse of values; Fig. 4(b) shows the optimized SRG in which each stream node has only one reuse source; Fig. 4(c) shows the expanded SRG with partial reuse of values. In Fig. 4(c), g, accessing the same sequence with s_2, is the generator; s_2, s_1 and s_0 reuse the whole data generated by g before 0, 1 and 2 iteration(s), respectively; s_3 reuses partial data generated by g in the same iteration.

We get code in Fig. 5(a) by introducing four constant-bound streams, s_{00}, s_{01}, s_{02} and o_{00}, initializing s_{02} and s_{01} before the loop, defining the generator g, loading the values referred to by g to s_{00}, replacing the references involved by references to whole or part of the corresponding constant-bound stream references, saving the output o_{00} to the locations defined by o_0 and moving the values of s_{01} and s_{00} to s_{02} and s_{01} at the end of the loop body. Since s_3 accesses from the $0th$ to $(N/2)th$ records of data generated by g, we replace s_3 with the reference $s_{00}(0,N/2)$. So, the compiler will recognize and utilize both whole and partial reuse of

```
Stream s(N*(N+2)), o(N*N);
Stream s0,s1,s2,s3,o0,g;
Stream s00(N), s01(N),s02(N);
Stream o00(N);

//init stream s01 and s02
s02←s(0,    N);
s01←s(N,2*N);

for(int i = 0; i<N; i++){

  g   = s((i+2)*N, (i+3)*N);
  o0 = o(i1    *N,(i1+1)*N);
  s00←g;
   k( s00(0,N/2), s00, s01, s02, o00);
  o0←o00;

  //Move values among streams
  s02←s01;
  s01←s00;
}
```

(a) Code with data copies

```
··· declare streams s, o, ···
··· init stream s01 and s02.

for(int i = 0; i<N-N%3; i++){

  //the 1st unrolled loop
  ··· define g,o0
   s00←g;
   k( s00(0,N/2), s00, s01, s02, o00);
   o0←o00;

  //the 2nd unrolled loop
   ··· define g,o0
   i = i  + 1;
  s02←g;
   k( s02(0,N/2), s02, s00, s01, o00);
   o0←o00;

  //the 3rd unrolled loop
   ··· define g,o0
   i = i  + 1;
  s01←g;
   k( s01(0,N/2), s01, s02, s00, o00);
   o0←o00;
 }

 // epilogue loop
 for(i = N-N%3; i<N;i++){
   ··· original loop body
 }
```

(b)Unrolled code

Figure 5: Different stages of program transformation.

the transformed code. However, it does so at the expense of introducing two expensive SRF-to-SRF data moves. Since these moves implement a permutation of values in the SRF, we can eliminate the need for moves by unrolling to the cycle length of the permutation (equal to 3 for the example) and permuting the stream references in each unrolled loop bodies, and get the code shown in Fig. 5(b).

Fig. 6 graphically depicts the SRF allocation for the code before (Fig. 6(a)) and after (Fig. 6(b)) transformation. In original code, the stream compiler generates a stream load for each input stream reference without utilizing any reuse; in transformed code, the stream compiler generates only one stream load for the stream reference g.

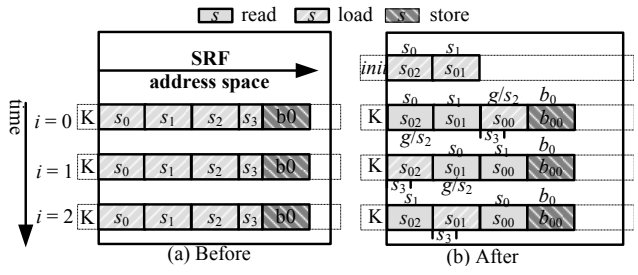


Figure 6: SRF allocation graph for the code.

The main contributions of this paper are as follows.

- We present an algorithm to identify stream reuse;
- We build and optimize, for the first time, the SRG to depict stream reuse in the SRF;
- We present a program transformation, called constant-bound stream replacement, which determines the unrolling factor, unrolls loop body and transforms the stream program, to effect the reuse among loop-dependent stream references;

- We implement our algorithm in the StreamC/KernelC compiler and evaluate the performance benefits of our techniques on a simulator for Imagine.

Section 2 presents work related to this paper. Section 3 describes the algorithm of recognizing both whole and partial stream reuse. In Section 4 we describe the construction of the SRG. Section 5 describes the optimization of the SRG. The constant-bound stream replacement algorithm is presented in Section 6. Our experiments and performance evaluation appear in Section 7 and we conclude the paper in Section 8.

2. RELATED WORK

Many schemes, such as *kernel fusion* [16, 14], *software-pipelining* and *loop unrolling* [22, 10, 20], have been proposed to exploit the reuse of data items in the LRFs. These optimizations can reduce the amount of data transferred between the SRF and LRFs, but have no influence on off-chip memory usage. To exploit the reuse of the data in the SRF, *strip-mining* [10], *software pipelining* and *loop-unrolling* [22, 10, 20] have been proposed. Although these optimizations can improve stream programs' performance, they can only be used for *constant-bound stream references* and only *loop-independent* stream reuse is exploited.

To the best of our knowledge, there are no schemes that look at exploiting the reuse supplied by loop-dependent stream references. Current schemes omit all loop-carried stream reuse.

Many authors have demonstrated exploiting data reuse supplied by indexed-variables for scalar and vector processors. Allen and Kennedy pioneered the use of dependence for optimization of register use on vector machines [2, 3]. Handling of complex loop nests was also due to Carr and Kennedy [5]. Handling of complex loop nests was also due to Carr and Kennedy [6]. Lu and Cooper [18] study the impact of powerful pointer analysis in C programs for register promotion. Kapasi et al. [14] and Lo et al. [17] show how to use static single assignment (SSA) [8] to facilitate register promotion. Lo et al. [17] also shows how PRE can be dualized to handle the removal of redundant store operations.

Strategies for scalar and vector reuse cannot be directly applied to exploiting the reuse supplied by loop-dependent stream references for several reasons. Firstly, the intermediate representations, such as the dependence graph, SSA, and PRE, on which the reuse in scalar and vector processors is exploited, assume that data participating in the computation has the same length. However, stream references may have different lengths. Therefore, existing intermediate forms cannot be used to optimize stream programs directly. Secondly, partial reuse and its treatment are quite new, and have never, to the best of our knowledge, appeared in scalar and vector processing. Finally, SRF-to-SRF data move is expensive and should be avoided during program transformation.

3. RECOGNIZING STREAM REUSE

3.1 Presentation

The iteration space of an n-level loop nest is expressed as $\overline{I} = (i_1, ..., i_n)$, where $i_1, ..., i_n$ from left to right express the index variables from the outermost to the innermost. A stream reference of the stride type is expressed as the tuple $\langle name,\ start,\ end,\ stride \rangle$, where *name* denotes its basic stream, *start* and *end* are start and end bounds respectively, and *stride* is the access stride. An affine start expression is expressed as: $start = \overline{V}_s\overline{I} + c_s$ where $\overline{I}$ is the iteration vector, $\overline{V}_s$ is called the *start access vector* and c_s is the *start constant term*. Correspondingly, an affine end expression can be expressed as: $end = \overline{V}_e\overline{I} + c_e$, where $\overline{V}_e$ is called the *end access vector* and c_e is the *end constant term*.

The reuse relationship is expressed as the tuple $\langle s_0, s_1, d \rangle$, where d is called the *reuse distance*. If $d \geq 0$, s_1 reuses data generated by s_0 d iterations earlier, with s_0 as the *reuse source* and s_1 as the *reuse destination*; otherwise, s_0 reuse data generated by s_1 d iterations earlier, with s_1 as the reuse source and s_0 as the reuse destination.

In addition, we assume that the innermost loop has been normalized, with i_n increased from zero to the constant $N-1$ in steps of 1. We also assume that the work space of a kernel is less than the SRF capacity. When a kernel does not meet this requirement, loop strip-mining [19, 20] can be applied. For simplicity, we assume here that all loops are perfectly nested.

3.2 Algorithms to Recognize Stream Reuse

As with the analysis of [27], we look to exploit location reuse among stream references that have the same start and end access vectors, and differ only in the constant terms, examples of which are shown in Example 1 in Fig. 2, namely the s_0, s_1, s_2 and s_3 references. Such references are called *uniformly generated stream references* [11].

Two streams, $s_0 = \langle s, \overline{V}_{s0}\overline{I} + c_{s0}, \overline{V}_{e0}\overline{I} + c_{e0}, stride_0 \rangle$ and $s_1 = \langle s, \overline{V}_{s1}\overline{I} + c_{s1}, \overline{V}_{e1}\overline{I} + c_{e1}, stride_1 \rangle$, are called *uniformly generated stream references* if:

(1) $\overline{V}_{s0} = \overline{V}_{e0} = \overline{V}_{s1} = \overline{V}_{e1}$ and

(2) $stride_0 = stride_1$. As the vectors are equivalent, we use $\overline{V}$ to represent them. We also define c_n to be the n*th* item of $\overline{V}$. That is, c_n represents the coefficient of the innermost index variable, i_n, in the start and end expressions.

Since little exploitable reuse exists between non-uniformly generated stream references, we partition stream references in a loop nest into equivalence classes of references that operate on the same basic stream and have the same $\overline{V}$. We call these equivalence classes *uniformly generated sets* [27]. In Example 1 from Fig. 2 the stream references s_0, s_1, s_2 and s_3 belong to a single uniformly generated set with an $\overline{V}$ of [1].

Next we analyze the scenarios in which loop-carried reuse exists. We first test whether the reuse of locations exists and then test whether the values of the reuse source maintain unchanged before being reused.

Loop-carried location reuse is possible in the following scenarios. A loop-invariant reference (i.e. $c_n = 0$) refers to the same locations in each iteration; a stream reference gets the values in the SRF generated by another reference in previous iterations. The *Reuse* algorithm tests when whole or partial location reuse exists between two stream references. To unify the process of loop-invariant and loop-variant stream references, let $C = 1$ if $c_n = 0$; otherwise $C = c_n$.

Algorithm *Reuse* For two stream references s_0 and s_1 belonging to the same uniformly generated set, location reuse relationship $\langle s_0, s_1, d \rangle$ is satisfied if:

(1) there are shared locations accessed by s_0 in iteration i_n and by s_1 in iteration $i_n + d$, i.e.

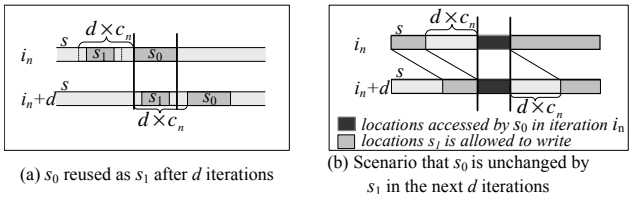
(a) $c_{s0} \bmod C \leq c_{s1} \bmod C < c_{e0} \bmod C$ or,

(b) $c_{s0} \bmod C < c_{e1} \bmod C \leq c_{e0} \bmod C$

(2) and, if inter-lane communication is forbidden, each shared location must lie on the same lane [19], i.e.

$(c_{s0} - c_{s1}) \bmod NC \times stride = 0$.

If $c_n = 0$, i.e. s_0 and s_1 are loop-invariant stream references, $C = 1$; shared locations are accessed by s_0 and s_1 in the same iteration and $d = 0$. Otherwise, $C = c_n$; shared locations are accessed in different iterations and $d = (c_{s0}/c_n - c_{s1}/c_n)$. Condition (2) is not necessary for stream processors with an indexed SRF [12] and the Cell Processor because these processors allow inter-lane communication. If conditions (1a) and (1b) are both satisfied, whole location reuse exists between s_0 and s_1; otherwise, partial location reuse exists. If $d \geq 0$, then s_0 is the reuse source and s_1 is the reuse destination; otherwise, s_1 is the reuse source and s_0 is the reuse destination.



(a) s_0 reused as s_1 after d iterations

(b) Scenario that s_0 is unchanged by s_1 in the next d iterations

Figure 7: Example for the algorithms *Reuse* and *IsClean*.

Fig. 7(a) shows the stream references s_0 and s_1 that meet the conditions of the *Reuse* algorithm with $d > 0$ and $c_n > 0$. As shown, part of the locations accessed by s_0 in iteration i_n are accessed by s_1 in iterations $i_n + d$, hence the reuse.

Since the reuse with respect to the SRF is the reuse of values, intervening writes introduce output dependencies, potentially inhibiting reuse. If the locations to be reused are not written to before reuse occurs then reuse is possible. The *IsClean* algorithm tests whether the values generated by the reuse source s_0 are not changed by the output reference s_1 in the next d iterations.

Algorithm *IsClean* For two stream references s_0 and s_1 that belong to the same uniformly generated set, the values used by s_0 are not changed by the output stream reference s_1 in the next $d(d > 0)$ iterations, if:

$c_{e0} < c_{s1} + \min(0, c_n \times d)$ or $c_{s0} > c_{e1} + \max(0, c_n \times d)$.

Fig. 7(b) draws the locations accessed by s_0 in iteration i_n and the locations that s_1 is allowed to write by the *IsClean* algorithm with $c_n > 0$. As shown, the values used by s_0 in iteration i_n are not changed by s_1 in the next d iterations.

4. STREAM REUSE GRAPH

The SRG captures value reuse among stream references and is applicable across differing stream architectures. It consists of following components: *stream nodes*, *kernel nodes*, *reference edges* and *reuse edges*. The *stream node*, drawn as a circle, denotes stream references with distinct values. The *kernel node*, drawn as a square block, denotes kernel calls. The directed *reference edge*, expressed as a hollow arrow, depicts that the kernel consumes the input stream (pointing from a stream to a kernel node) or produces the output streams (the arrow comes from the kernel node). The directed *reuse edge*, drawn as a solid arrow with its value d, shows values of the source node are reused at the destination after d iterations. Stream nodes, kernel nodes and reference edges form a DAG that captures the relationship among kernel and stream nodes. Stream nodes and reuse edges form a RG that captures the reuse relationship among stream nodes. The standard SRG depicts only whole stream reuse. Section 5 shall extend the SRG to also depict partial reuse.

The SRG is built for straight-line loop l in the following two steps. An extension of this work is to consider branches within the innermost loop. The SRG is still applicable as branches can be restructured to be around the innermost loops.

Step 1. This step builds the DAG describing the relationship between stream references and kernels, for l; it builds kernel nodes and stream nodes, and then inserts reference edges between them. During the construction, a stream node is built only for the stream reference with distinct values. That is, if two stream references use the same values, only one stream node is built for the first reference, and a reference edge is built from the node to the kernel that refers to the second reference. So the DAG also describes loop-independent reuse.

First, a kernel node k_i is built for the corresponding i*th* kernel call. Next stream references within k_i are processed. A stream reference with distinct values is either loaded from off-chip memory, if accessed for the first time, or produced by k_i. A stream node is built for such a stream reference. For the other references, we find the node generating data needed by the reference.

In the algorithm implementation, an output stream node is created for each output reference. However, while dealing with an input reference, this step searches all built stream nodes for stream nodes accessing the same locations with the reference. The latest built node among them has the values needed by the input reference. If such a node is not found, an input stream node is built for the input reference. Finally, reference edges are inserted from the built or found stream nodes to the kernel node k_i.

The DAG for code in Fig. 8(a), for example, is built as follows. The kernel node k_1 is built first; the stream nodes t_0, t_1 and t_2 are built, due to their distinct values. Next the kernel node k_2 is built. However, as the values needed by t_0 of k_2 have been loaded into the SRF by the built node t_0 of k_1, a reference edge is inserted from the built node t_0 to k_2. Similar treatment occurs for the input reference t_2 of k_2. The built DAG is shown in Fig. 8(b).

The DAG describes the definition and use of the stream references in l. As a result, loop-independent reuse is shown in the DAG in the form of two or more kernel nodes being connected to the same stream node.

Step 2. This step builds the RG based on the built DAG, it must identify all incidents of loop-carried whole value reuse among stream nodes, and add the corresponding reuse edges.

First, this step, using the *Reuse* algorithm, identifies the opportunities for whole location reuse supplied by two stream nodes s_i and s_j. If the reuse is identified, $\langle s_i, s_j, d_{i,j} \rangle$ is returned. Next, this step detects whether the values of the reuse source are unchanged by every output stream node s_o before the reuse. If s_o is an argument of a lexically earlier kernel than the first kernel call referring to the reuse destination, this step, using the *IsClean* algorithm, tests whether the values remain unchanged in next $|d_{i,j}|$ iterations. If not the algorithm tests whether the values remain unchanged in the next $|d_{i,j}| - 1$ iterations (If $|d_{i,j}| = 0$, the test is not nec-

essary). If the reuse source is unchanged before the reuse, the value reuse $\langle s_0, s_1, d_{i,j}\rangle$ is satisfied. A reuse edge is inserted from the source to the destination with $|d_{i,j}|$ as its value.

The graph in Fig. 8(c) is built by this step. The values of t_2 are changed by the output stream t_1 of k_2 after one iteration. Therefore, t_2 cannot be reused as t_0 after two iterations. However, a reuse edge is inserted from t_2 to t_1 of k_1 because the change to the values occurs after the reuse..

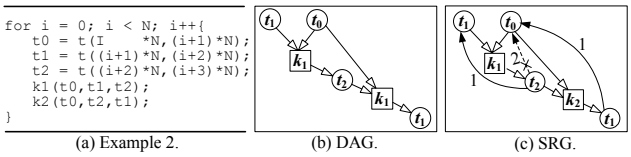


Figure 8: Example 2 showing the construction of the SRG.

5. OPTIMIZING THE SRG

Although the built SRG describes all whole reuse in loop l, it cannot be used for the transformation for the following reasons. (1) One stream node may have multiple data sources, such as the stream node s_0 in the SRG in Fig. 4(a) that is built for Example 1 from Fig. 2. But each stream node may reuse only one data source for the convenience of the transformation. (2) The SRG does not describe the partial reuse. Next we optimize the built SRG to overcome the deficiencies.

5.1 Selecting Data Source

It can be proven that the following properties are satisfied in the built SRG:

If $\langle s_i, s_j, d_{i,j}\rangle$ and $\langle s_j, s_k, d_{j,k}\rangle$, then $\langle s_i, s_k, d_{i,j} + d_{j,k}\rangle$;

If $\langle s_i, s_k, d_{i,k}\rangle$ and $\langle s_j, s_k, d_{j,k}\rangle$, then $\langle s_i, s_j, d_{i,k} - d_{j,k}\rangle$.

The proof is not presented here due to insufficient space. These properties demonstrate that the RG is divided into q unilateral directed connected acyclic *sub-RGs*. Therefore, there is one and only one stream node that has no reuse edges pointing to it in each *sub-RG*$_i$, and there is a reuse edge issued from the stream node to every other stream node. That is, all other stream nodes can reuse values generated by this stream node a number of iterations ago. This stream node is called *generator*. We choose the generator g_i as the data source for stream nodes $s_{i,j}$ in *sub-RG*$_i$. So only g_i needs memory transfers while $s_{i,j}$ reuses the values generated by g_i $d_{i,j}$ iterations earlier.

First, this step picks up *sub-RG*$_i$, then finds the generator g_i for *sub-RG*$_i$ and finally cuts all reuse edges that are not issued from g_i in *sub-RG*$_i$. This step is repeated until all *sub-RGs* are found.

The SRG in Fig. 4(a), for example, has three sub-RGs: o_0 composes *sub-RG*$_0$ and g_0 is o_0; s_3 composes *sub-RG*$_1$ and g_1 is s_3; s_0, s_1 and s_2 compose *sub-RG*$_2$ and g_2 is s_2. In *sub-RG*$_2$, the reuse edge from s_1 to s_0 is cut. The optimized SRG is shown in Fig. 4(b).

5.2 Expanding the Optimized SRG for Partial Reuse

We expand the optimized SRG to make it also depict partial reuse. From the algorithm *Reuse* we reach the conclusion that for two generators g_{i1} and g_{i2}, if the partial location reuse $\langle g_{i1}, g_{i2}, d_{i1,i2}\rangle (d_{i1,i2} \geq 0)$ is satisfied, the partial location reuse $\langle g_{i1}, s_{i2,k}, d_{i1,i2} + d_{i2,k}\rangle$ is satisfied for the stream node $s_{i2,k}$ such that $\langle g_{i2}, s_{i2,k}, d_{i2,k}\rangle$ holds.

Next we give the condition that *sub-RG*$_{i1}$ and *sub-RG*$_{i2}$ can be merged into one new *sub-RG*, with a new generator, g_{new}. The stream nodes in the merged *sub-RG*, i.e. stream nodes in *sub-RG*$_{i1}$ and *sub-RG*$_{i2}$ reuse part or whole data generated by g_{new} certain iterations earlier. We first find g_{new} for the merged *sub-RG* and then give the condition that partial value reuse exists between g_{new} and stream nodes in the merged *sub-RG*.

As defined before, the stream node s_k in the merged *sub-RG* with reuse distance d_k uses data generated by g_{new} d_k iterations earlier. Therefore, the sequence accessed by g_{new} in iteration i_n is the supersequence of all sequences accessed by s_k in iteration $i_n + d_k$. Using the results of the algorithm *Reuse*, we get the following results: $g_{new}.start = g_{i1}.start$ - $g_{i1}.c_s \mathrm{mod} C$ +min($g_{i1}.c_s \mathrm{mod} C$, $g_{i2}.c_s \mathrm{mod} C$); $g_{new}.end = g_{i1}.end$ - $g_{i1}.c_e \mathrm{mod} C$ + max($g_{i1}.c_e \mathrm{mod} C$, $g_{i2}.c_e \mathrm{mod} C$). Let D_{i1} be the maximum reuse distance in *sub-RG*$_{i1}$; D_{i2} be the maximum reuse distance in *sub-RG*$_{i2}$. From Fig. 9, we can intuitively see the relationship between g_{new} and g_{i1}, g_{i2}.

Now, we give the condition that the partial value reuse is satisfied between g_{new} and stream nodes in the merged *sub-RG*. If g_{new} remains unchanged in next $d_{i1,i2}$ iterations, the partial value reuse is satisfied. Fig. 9 describes which parts of g_{new} are reused by stream nodes in *sub-RG*$_{i1}$ and *sub-RG*$_{i2}$ at what iteration. Values generated by g_{i1} are unchanged from iteration i_n to $i_n + D_{i1}$, due to the value reuse in *sub-RG*$_{i1}$, so this part of g_{new} is not changed from iteration i_n to $i_n + D_{i1}$, so partial value reuse exists between g_{new} and the stream nodes in *sub-RG*$_{i1}$. Similarly, values generated by g_{i2} are unchanged from iteration $i_n + d_{i1,i2}$ to $i_n + d_{i1,i2} + D_{i2}$, and g_{new} is unchanged in next $d_{i1,i2}$ iterations. So this part of g_{new} is not changed from iteration i_n to $i_n + d_{i1,i2} + D_{i2}$, so partial value reuse exists between g_{new} and the stream nodes in *sub-RG*$_{i2}$.

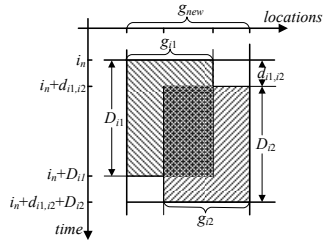


Figure 9: Scenario in which parts of g_{new} are reused as stream nodes in *sub-RG*$_{i1}$ and *sub-RG*$_{i2}$ at what iteration.

The above two results can be expanded straightforwardly to merge multiple *sub-RGs* and find the new generator for the merged *sub-RG*. The following process is performed to expand the SRG for partial reuse. (1) Identify partial value reuse between two generators, whose *sub-RGs* can be merged, compute the new generator g_{new} and place the generators in a set called the *merged-set*. If partial reuse exists between two generators in the same iteration, the stream node of a lexically earlier kernel call is the reuse source. (2) Find another generator that has partial value reuse between g_{new}, and update g_{new}. (3) Repeat (1) and (2) until there is no

stream node having partial value reuse with g_{new}. (4) Build a new stream node for g_{new} and insert a reuse edge, with d_i as its value, from g_{new} to the generator g_i ($g_i \in$*merged-set*) such that $\langle g_{new}, g_i, d_i \rangle$ is satisfied. (4) For each stream node $s_{i,k}$ such that $\langle g_i, s_{i,k}, d_{i,k} \rangle$, update the reuse source of $s_{i,k}$ to be g_{new} and its reuse distance to be $d_i + d_{i,k}$. Thus, the *sub-RGs* with generators in the *merged-set* are merged into one new *sub-RG*. At last, all stream nodes are divided into p connected *sub-RGs*. All stream nodes reuse partial or whole values generated by their generators certain iterations earlier.

Fig. 4(c), for example, is expanded from the optimized SRG in Fig. 4(b) for the code from Fig. 2. Partial value reuse exists between the generators s_3 and s_2, so *sub-RG*$_1$ and *sub-RG*$_2$ are merged into one new *sub-RG* (we call the merged *sub-RG* *sub-RG*$_1$). The new generator is the stream reference $s((i+2)\times N, (i+3)\times N)$ and a stream node g is built for it. The reuse sources of s_3, s_2, s_1 and s_0 are updated to be g.

6. CONSTANT-BOUND STREAM REPLACEMENT

In this section, we transform a stream-level program to make use of the reuse supplied by loop-dependent stream references, based on the built SRG. The procedure consists of four steps, as follows.

Determining the Number of Constant-Bound Stream References. Now, we calculate the number of constant-bound streams, N_i, needed to hold the data generated by the generator, g_i. To eliminate the load of the reuse destinations in *sub-RG*$_i$, D_i+1 constant-bound streams are needed [5], thus providing one constant-bound stream reference for values computed on the D_i previous iterations plus one for the values computed on the current iteration.

Determining the Loop Unrolling Factors. To eliminate all SRF-to-SRF moves that implement data permutation among N_i streams in *sub-RG*$_i$, the loop should be unrolled N_i times. To eliminate all data permutation for all sub-RGs, the unrolling factors, NU, equals $LCM(N_0,...,N_{p-1})$.

Reference Replacement. This procedure first creates N_i constant-bound streams, g_i^0,... ,$g_i^{N_i-1}$ for each generator, g_i. Next, the innermost loop body is unrolled NU times and a statement $i_n = i_n + 1$ is inserted just before each unrolled loop body except the first one. Finally, the procedure replaces the references involved by references to corresponding constant-bound stream references and inserts stream loads or saves at right place.

For each reference $s_{i,j}$ such that $\langle g_i, s_{i,j}, d_{i,j} \rangle$ is satisfied, in order to specify which part of data generated by g_i is reused by $s_{i,j}$ after $d_{i,j}$ iterations, the start and end offsets of locations accessed by $s_{i,j}$ on iteration $i_n + d_{i,j}$ to the start of g_i on iteration i_n are computed. According to the above computation of $d_{i,j}$ and g_i, the start offset of $s_{i,j}$, $startOffset_{i,j} = s_{i,j}.c_s \bmod C - g_i.c_s \bmod C$; the end offset, $endOffset_{i,j} = s_{i,j}.c_e \bmod C - g_i.c_s \bmod C$.

In the first unrolled body, the references to g_i are replaced with g_i^0; the references to $s_{i,j}$ are replaced with the stream reference $g_i^{d_{i,j}}(startOffset_{i,j}, endOffset_{i,j})$. In the uth unrolled body, the references to $s_{i,j}$ are replaced with the stream reference $g_i^{(d_{i,j}-u) \bmod N_i}(startOffset_{i,j}, endOffset_{i,j})$.

For the references to the stream nodes with reuse distance equal to 0, if they are produced by their first kernel call, the produced constant-bound streams are saved back just after its first kernel call. If they are loaded by their first kernel calls, a load of g_i is inserted just before the first kernel call referring to part or whole of g_i. If g_i is loop-invariant (i.e. $c_n = 0$), all loads or saves are merged into a single load or save and moved just before or after the innermost loop body.

Initialization. This procedure initializes the constant-bound streams, processes the innermost loop index variable and produces the epilogue loop. We do not detail the last two parts of the procedure as they are straightforward.

For *sub-RG*$_i$, D_i constant-bound streams should be initialized with the right values. For the stream node $s_{i,j}$ such that $\langle g_i, s_{i,j}, d_{i,j} \rangle$, data used by $s_{i,j}$ in the first iteration should be assigned to $g_i^{d_{i,j}}$. If multiple stream nodes have the same reuse distance, their supersequence is assigned. If there is no reuse distance equal to $d_{i,j}+1$, data used by $s_{i,j}$ in the second iteration is assigned to $g_i^{d_{i,j}+1}$, and so on.

By now, the reuse supplied by loop-dependent stream references has been exploited and corresponding stream loads have been eliminated. Fig. 5(b) shows the final code of the example in Fig. 2. There are two *sub-RGs*, *sub-RG*$_0$ = $\{o_0\}$ and *sub-RG*$_1$=$\{s_0, s_1, s_2, s_3\}$; $g_0 = o_0$ and $g_1 = s((i+2)\times N, (i+3)\times N)$. In *sub-RG*$_1$, s_3 reuses part of the generator g in the same iteration. Its start and offsets equal 0 and $N/2$, respectively. Other stream references reuse whole of g. The loop should be unrolled three times to eliminate SRF-to-SRF data moves among three constant-bound streams in *sub-RG*$_1$.

7. EXPERIMENT

We have implemented a source-to-source translator indicated by dark shading in the stream compiler developed by Stanford University [10]. Fig. 10 shows our experimental framework. The translator takes a StreamC program as input, builds the SRG for each loop body, unrolls each loop body and then replaces loop-dependent stream references by constant-bound stream references. Table 1 summarizes the 10 applications we used. Test1, and Test2 represent the examples in Fig. 2 and Fig. 8, respectively. QMRCGSTAB [7] is a real application and used to solve large sparse linear systems with asymmetrical coefficient matrices with the Krylov subspace iteration method. MVM calculates the multiplication of two band matrices. The item max(*computation density*), used to evaluate the relationship between computations and memory transfers, is quantified with the number of the computations per word transfer, assuming that all reuse in the application is exploited. The code optimized with and without our method is performed on a cycle-accurate simulator of Imagine, with a 128KB SRF.

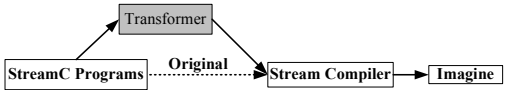


Figure 10: Framework of experimental design.

7.1 Whole Reuse

We first demonstrate the importance of the exploitation of whole reuse. Seven scientific kernels that are simple and frequently used in real applications are used to evaluate the exploitation. There is only one loop in each of the stream-

	Laplace	Swim	MG	GEMM	FFT	Jacobi	QMRCGSTAB	MVM	Test1	Test2
Source	NCSA	Spec2000	NPB	BLAS	HPCC	-	-	-	-	-
Problem Size	$1K \times 1K$	512×512	$128 \times 128 \times 128$	512×512	$4K$	256×256	800×800	832×832	$1K \times 1K$	$1K \times 1K$
max(comp. density)	2.5	3.67	3.67	341	44	4.98	1.85	1.5	1	2

Table 1: Application programs.

Bench-mark	# Stream refer-ences	# sub-RGs	Unroll-ing factor	Execution time (cycles)					Loads (words)			% Comp. Density	
				Before	Unrolled	After	Speedup		Before	After	% Re-duction	Before	After
Laplace	4	2	3	5100800	4920800	4251473	1.20	1.16	3147312	1051472	66.6	50.0	99.8
Swim	6	5	2	2438930	2574754	2015020	1.21	1.18	1579320	793680	49.7	65.2	93.1
MG	13	4	6	2856388	2656388	1682982	1.70	1.58	2261664	934816	58.7	45.3	91.2
GEMM	4	4	1	66548854	-	58377634	1.14	-	27820032	19693568	29.2	0.022	0.029
MVM	5	3	3	2403683	2190350	1778603	1.35	1.23	2107016	1577864	25.1	66.4	85.5
Test1	5	2	4	5539172	4838179	3262166	1.70	1.48	3675520	1576320	57.1	44.4	80.0
Test2	6	2	2	5569908	5379828	3320023	1.68	1.62	4194520	2264	99.9	16.7	49.9

Table 2: Effect of whole reuse exploitation on execution time, memory load and computation density.

level programs. Table 2 shows the effectiveness of the whole reuse exploitation. The column *#Stream references* illustrates the number of loop-dependent stream references of each loop body. The column *#sub-RGs* describes the number of *sub-RGs* in each SRG. A stream load/save is needed only for each generator of each *sub-RG* on each iteration or just before/after the innermost loop body. These two columns somewhat reflect the complexity of building the SRG. The column *Unrolling factor* shows the number of times that each loop is unrolled to avoid SRF-to-SRF data moves.

The column *Execution time* demonstrates the significant effect of our automatic method on the execution time. The column *Unrolled* shows the speed of the original loop after unrolling it an amount equivalent to that done by our method. This column shows the impact of unrolling on the performance. The speedup column has two fields. The first field shows the speedup of the transformed code over original code; our method gains from 1.14 to 2.54 times speedup. The second field shows the speedup of the transformed code over the unrolled code; our method gains from 1.16 to 2.22 times speedup.

As the main contribution of our method is the elimination of the memory loads, the column *Loads* illustrates this ability. Although some stream saves are eliminated by our method in some applications, we shall only focus on the reduction of stream loads. The column *% Reduction* equals $(Before\text{-}After)/Before \times 100\%$ showing the percentage of the reduced loads from the loads in original code.

GEMM has loop-invariant reuse. Except the initial load, one out of the three input streams do not need a load in the innermost loop. Test2 has a similar improvement. All loads within an iteration are eliminated. As our method aims to only reduce stream loads, there are still redundant stream saves that can be optimized in Test2. Laplace, a memory bound application, is another interesting case because its stream loads are reduced to nearly one third. For other applications, the performance improvement is proportional to the reduction of memory transfers.

The column *% Comp. Density* evaluates the exploitation degree of the stream reuse and is quantified by comparing the achieved computation density to the maximum computation density in Table 1. If its value equals 100%, all reuse is exploited. Our automatic method effects all stream reuse in Laplace, Swim, MG and MVM, although they do not get the maximum computation density. The load of kernel code introduces extra memory transfers and thus reduces the computation density in Laplace and Test1; some redundant data is introduced in Swim, NG and MVM when mapping them on the stream processor in order to utilize the architecture features. Partial reuse exits in Test1, which will be analyzed in the next subsection. For Test2, the stream reuse is completely exploited, though there are still redundant stream saves which are not considered by our method. As GEMM is mapped on the stream processor by *loop-tiling*, some reuse does not exist in the innermost loop. However, our method exploits all its reuse with respect to the innermost loop.

7.2 Partial Reuse

We now demonstrate the effectiveness of the exploitation of partial reuse. Three scientific kernels, Jacobi, FFT and Test1, benefit from it. Fig. 11 draws the SRGs of Jacobi and FFT. The outputs of kernels J_0 and J_1 are concatenated together to be an input of the next iteration, which indicates the generator g_0 is the union of x_0 and x_1. For FFT, the first half and second half of its output act as two inputs in the next iteration. The generator g_1 accesses the same sequence with the stream reference o; the stream references i_0 and i_1 access the first and second half parts of data used by g_1 in the last iteration.

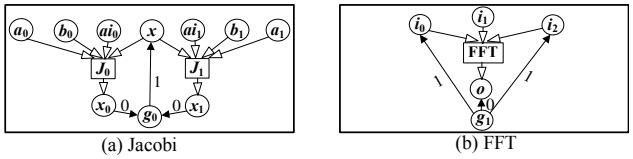


Figure 11: SRGs of Jacobi(a) and FFT(b).

Our method eliminates two loads in each iteration of FFT, one load in each iteration of Jacobi and three loads in each iteration of Test1. The effectiveness of partial reuse is shown in Table 3. FFT is an interesting case. Although FFT is a computation-intensive application, it benefits greatly from our method. Without our method, its stream program is somewhat memory-bound. As FFT is mapped on the stream processor optimized by strip-mining [10], some reuse does not exist in the innermost loop. Therefore we do not

Bench-mark	# Stream refer-ences	# sub-RGs	Unroll-ing factor	Execution time (cycles)					Loads (words)			% Comp. Density	
				Before	Unrolled	After	Speedup		Before	After	% Re-duction	Before	After
Jacobi	10	7	2	384949	372939	303681	1.27	1.23	434308	331754	23.6	19.7	19.9
FFT	4	2	1	153375	-	73935	2.07	-	62232	17832	71.3	19.8	68.7
Test1	5	2	4	5539172	4838179	2174777	2.54	2.22	3675520	1050880	71.4	44.4	99.8

Table 3: Effect of both whole and partial reuse exploitation on execution time, memory load and computation density.

gain the maximum computation density. Although Jacobi has a high inherent computation density, most reuse does not exist in the innermost loop. Therefore, its reuse cannot completely be captured. Compared with the result in Table 2, the exploration of partial reuse in Test1 reduces by one third loads; all reuse in Test1 is exploited.

7.3 Complete Application

We also performed the test on a complete application, QMRCGSTAB [7]. It consists of 4 2-level loops. Two of them, called *QMR iteration*, perform the main calculations. We only perform each outer loop once. Fig. 12 shows the SRG of the first QMR iteration. As shown, the reuse relationship among stream references is complicated. Table 4 summarizes the characteristics of the SRG of each loop of QMRCGSTAB. It has 41 loop-dependent stream references in total. Each loop requires distinct unrolling factors. This further demonstrates the complex reuse relationship in real applications. Therefore it would be extremely difficult to reuse the streams manually.

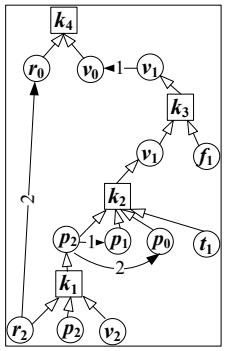


Figure 12: SRG of QMR loop.

Loop No.	1	2	3	4
# Stream references	14	6	17	6
# sub-RGs	8	5	9	5
Unrolling factors	3	1	3	1

Table 4: Characteristics of SRGs

Table 5 shows the results from when our method is applied to QMRCGSTAB. The application achieves a 28.2% speedup and reduces the number of loads by 30.5%. All possible stream reuse within the application has been exploited.

8. SUMMARY

In this paper, we have introduced the problem of stream reuse for stream processors. We have built and analyzed methods to optimize stream reuse based around the SRG. Our approach has considered whole and partial stream reuse. Based on the SRG, we proposed a source-to-source transformation, called constant-bound stream replacement, that makes the reuse supplied by loop-dependent stream references useful for the stream compiler, thus reducing unnecessary memory loads. Our optimization is implemented in the StreamC/KernelC compiler. Results demonstrate that our techniques can lead to a dramatic increase in performance. For 9 benchmarks, on the Imagine stream processor, a speedup of 1.14 to 2.54 times was achieved.

	Before	After	% Reduction
Loads (words)	35630663	24767143	30.5
Execution time (cycles)	36918146	29686296	19.6
% Comp. density	53	99.6	-

Table 5: Effect of our method on the performance of QMRCGSTAB.

9. ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their feedback and improvements to this paper. This work was supported by NSFC (60621003).

10. REFERENCES

[1] J. H. Ahn, M. Erez, and W. J. Dally. The design space of data-parallel memory systems. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 80, New York, NY, USA, 2006. ACM Press.

[2] J. R. Allen. *Dependence analysis for subscripted variables and its application to program transformations*. PhD thesis, Houston, TX, USA, 1983.

[3] R. Allen and K. Kennedy. Vector register allocation. *IEEE Trans. Comput.*, 41(10):1290–1317, 1992.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 53–65, New York, NY, USA, 1990. ACM.

[6] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Softw. Pract. Exper.*, 24(1):51–77, 1994.

[7] T. F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. H. Tong. A quasi-minimal residual variant of the bi-cgstab algorithm for nonsymmetric systems. *SIAM J. Sci. Comput.*, 15(2):338–347, 1994.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K Zadeck. Efficiently computing static single assignment form and the control dependence graph. Technical report, Providence, RI, USA, 1991.

[9] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. K., and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society.

[10] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 33–42, New York, NY, USA, 2006. ACM Press.

[11] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *J. Parallel Distrib. Comput.*, 5(5):587–616, 1988.

[12] N. Jayasena, M. Erez, J. H. Ahn, and W. J. Dally. Stream register files with indexed access. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 60, Washington, DC, USA, 2004. IEEE Computer Society.

[13] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[14] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The imagine stream processor. In *ICCD*, pages 282–288, 2002.

[15] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[16] S. W. Liao, Z. H. Du, G. S. Wu, and G. Y. Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[17] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 1998. ACM.

[18] J. Lu and K. D. Cooper. Register promotion in c programs. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 1997. ACM.

[19] P. Mattson. *A programming system for the imagine media processor*. PhD thesis, Computer Systems Laboratory, Stanford University, 2002. Adviser-William J. Dally.

[20] P. Mattson and et al. *Imagine Programming System Developer's Guide*, 2004.

[21] M. Narayanan, L. Oliker, A. Janin, P. Husbands, X Ye, and S. Li. Scientific kernels on viram and imagine media processors. Lawrence Berkeley National Laboratory, 2002.

[22] J. D. Owens, U. J. Kapasi, P. Mattson, B. Towles, B. Serebrin, S. Rixner, and W. J. Dally. Media processing applications on the Imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 295–302, September 2002.

[23] S. Rixner. *Stream Processor Architecture*. Kluwer Academic Publishers, 2002.

[24] A. V. S. Sastry and R. D. C. Ju. A new algorithm for scalar register promotion based on ssa form. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 15–25, New York, NY, USA, 1998. ACM.

[25] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[26] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[27] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM.

[28] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[29] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. In *10th Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 1366, pages 16–33, Minneapolis, Minn., 1997. Springer-Verlag.

[30] X. J. Yang, X. B. Yan, Z. C. Xing, Y. Deng, J. Jiang, and Y. Zhang. A 64-bit stream processor architecture for scientific applications. In *ISCA '07*, volume 35, pages 210–219, New York, NY, USA, 2007. ACM Press.