Optimizing Scientific Application Loops on Stream Processors

Li Wang [*] Xuejun Yang[*] Jingling Xue[†] Yu Deng[*] Xiaobo Yan[*]
Tao Tang[*] Quan Hoang Nguyen[†]

National laboratory for Parallel and Distributed Processing, School of Computer, NUDT, China[*]
Programming Languages and Compilers Group, School of Computer Science and Engineering, UNSW, NSW 2052, Australia[†]
{xjyang, xbyan, yudeng}@nudt.edu.cn, {lwang, jingling, quanhn}@cse.unsw.edu.au

Abstract

This paper describes a graph coloring compiler framework to allocate on-chip SRF (Stream Register File) storage for optimizing scientific applications on stream processors. Our framework consists of first applying enabling optimizations such as loop unrolling to expose stream reuse and opportunities for maximizing parallelism, i.e., overlapping kernel execution and memory transfers. Then the three SRF management tasks are solved in a unified manner via graph coloring: (1) placing streams in the SRF, (2) exploiting stream use, and (3) maximizing parallelism. We evaluate the performance of our compiler framework by actually running nine representative scientific computing kernels on our FT64 stream processor. Our preliminary results show that compiler management achieves an average speedup of 2.3x compared to First-Fit allocation. In comparison with the performance results obtained from running these benchmarks on Itanium 2, an average speedup of 2.1x is observed.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers and optimization; B.3.2 [*Memory Structures*]: Design Styles—Primary memory

General Terms Algorithms, Languages, Performance

Keywords Stream processor, streaming, loop optimization, data reuse, prefetching, graph coloring, software-managed cache

1. Introduction

The stream architecture [6, 22, 28, 31] offers high performance by exploiting parallelism through organizing computations on clusters of ALUs in a SIMD fashion and achieving data reuse by incorporating a bandwidth hierarchy of local register files, a global software-managed on-chip stream register file (SRF) and off-chip memory. In stream programming, a program is expressed as a sequence of data streams flowing through computation kernels executed on the stream processor. The data records in all streams consumed or produced by a kernel must be made available in the SRF before they are accessed (as the SRF is non-bypassing). Therefore, efficient SRF management is crucial for good performance.

The stream architecture represents a promising alternative to cache-based architectures in achieving high performance in signal processing, multimedia and graphics [21, 22, 26]. However, its enormous potential in scientific computing awaits further research. In [31], we introduced the design and fabrication of FT64, the first 64-bit stream processor for scientific computing. This paper is concerned with the development of compiler techniques for optimizing scientific applications on FT64-like stream processors. Specifically, we describe and evaluate our compiler framework used for automatic management of on-chip SRF.

In scientific computing, loops often dominate the performance of many scientific and engineering programs. Thus, optimizing loops on stream processors is critical in achieving good performance. To minimize program execution times, compiler-directed SRF management must address the three key issues: (1) efficient allocation of the scarce on-chip SRF, (2) exploiting stream reuse, and (3) maximizing parallelism. For (1), the allocation of streams in the SRF must be automated. For (2), the streams that are already in the SRF should not be unnecessarily spilled to off-chip memory in order to reduce off-chip bandwidth. For (3), kernel execution and stream transfer operations between the SRF and off-chip memory should be overlapped to hide memory latency. Prior to SRF allocation, enabling optimizations such as loop unrolling can be applied to expose more optimization opportunities. These three optimization goals often impose conflicting constraints on SRF allocation algorithms and loop optimizations to be applied.

In this paper, we present a graph coloring compiler framework to automate on-chip SRF allocation for optimizing scientific applications on stream processors. The main contribution of this paper is a graph coloring approach for solving the three above-mentioned SRF management tasks in a unified manner. We also show that loop unrolling can be applied beneficially as an enabling transformation to expose temporal stream reuse and parallelism inherent among different loop iterations. In addition, the prior work on optimizing media applications on stream processors considers output-input spatial reuse only [21, 22, 26] while this work exploits also input-input temporal reuse. Finally, our preliminary results obtained from actually running nine representative scientific computing kernels on our FT64 stream processor [31] show that significant performance speedups can be achieved in our framework.

The rest of this paper is organized as follows. Section 2 introduces the stream programming model used for stream processors and motivates this work by an example. Section 3 describes our compiler framework. Section 4 presents and analyzes our experimental results obtained from nine representative scientific kernels. Section 5 reviews the related work. Section 6 concludes the paper.

2. Background

2.1 Stream Programming Model

In the stream programming model illustrated in Figure 1, an application program is restructured into two programs, a *stream program*, which runs on the host processor, and a *kernel program*, which runs on the stream processor [20]. Stream and kernel programs are coded in different languages and compiled by different compilers. The stream program specifies stream movement and initiates kernel operations on streams. The kernel program, which consists of a number of kernels, performs the computations on streams. Each stream is a sequence of data records of the same type and each kernel is a program running on the stream processor that performs the same set of operations on each input stream element and produces one or more output streams.

This paper is concerned with SRF allocation, which is realized as an important component in the compiler for stream programs. As a result, kernel programs will not be discussed any further. It is sufficient to regard kernel calls in a stream program as normal subroutine calls (except that the kernels are run on the stream processor).

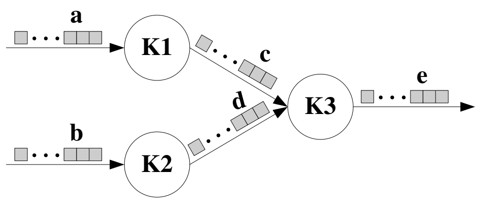


Figure 1: Stream programming model.

As shown in Figure 1, a stream program decouples the computation and memory operations in a kernel by boosting the memory reads before the computation and postponing the memory writes after the computation. As a result, stream programming tackles the memory latency problem by solving a memory bandwidth problem. There are three kernels in Figure 1. *K1* takes stream *a* as input and produces stream *c* as output. *K2* reads *b* and writes *d. K3* has two input streams, *c* and *d*, and one output stream *e*.

In a stream processor (cf. Figure 17), the SRF is shared by all its clusters of ALUs. As the SRF is non-bypassing, the data records in all streams accessed by a kernel must be made available in the SRF before they are accessed. During program execution, all clusters are simultaneously executing one kernel at a time in a SIMD fashion. The execution of a kernel can be overlapped with two or more concurrent stream transfer requests that are used to proactively fetch streams to be used by the ensuing kernel execution.

In our stream-level programming language, the stream transfer operations between the SRF and off-chip memory are not exposed to the programmer. Therefore, SRF management is left to the compiler's discretion. The API provides the following stream operations to manipulate the stream flows between the host memory (of the host processor) and the off-chip memory (of the stream processor). *streamInit* initializes a stream residing in off-chip memory by gathering data from a section in an array residing in host memory. Conversely, *streamSave* scatters data from a stream residing in off-chip memory to a section in an array residing in host memory. *streamCopy* makes a (value) copy of a stream in off-chip memory. Finally, *Kernel* is a special function call to initiate the execution of a kernel in the kernel program running on the stream processor.

As a result, the language requires the programmer to be responsible for organizing application I/O operations, namely gathers and scatters between host memory and off-chip memory. The compiler will insert explicit stream transfers between SRF and off-chip memory by using two more stream operations, *streamLoad* and *streamStore*. streamLoad loads a stream from off-chip memory into the SRF with its semantics similar to the load for loading a scalar from main memory to register. streamStore spills a stream from the SRF to off-chip memory with its semantics similar to a scalar spill.

```
do j = 2, n-1
     do i = 2, n-1
         d(i, j) = s(i-1, j) + s(i, j-1)+
               s(i+1, j) + s(i, j+1) – 4*s(i, j)
    enddo
enddo
```

Figure 2: Laplace loop.

```
1   float SMAT[(NR+1)*NC];
2   float OMAT[NR*NC)];
3   stream<float> a1(NC), a2(NC), a3(NC), a4(NC);
4   stream<float> o1(NC), o2(NC);
5   UC<float> m1, m2, m3, m4;

6   dataLoad('matrix.dat', (NR+1)*NC, SMAT);
7   streamInit(SMAT[0*NC, 1*NC], a1);
8   streamInit(SMAT[1*NC, 2*NC], a2);

9   for (k = 0; k < NR; k += 2 ) {
10      streamInit(SMAT[(k+2)*NC, (k+3)*NC], a3);
11      streamInit(SMAT[(k+3)*NC, (k+4)*NC], a4);
12      m1 = SMAT[k*NC+NC-1];
13      m2 = SMAT[(k+1)*NC+NC-1];
14      m3 = SMAT[(k+2)*NC];
15      m4 = SMAT[(k+3)*NC];
16      Kernel('lap', m1, m2, m3, m4, a1, a2, a3, a4, o1, o2);
17      streamSave(o1, OMAT[k*NC, (k+1)*NC]);
18      streamSave(o2, OMAT[(k+1)*NC, (k+2)*NC]);
19      streamCopy(a3, a1);
20      streamCopy(a4, a2);
21  }

22   dataSave('omatrix.dat', NR*NC, OMAT);
```

Figure 3: Stream program for Laplace.

2.2 Example

Our motivating example is the *Laplace* transformation, which is frequently used in scientific applications such as PDE (Partial Differential Equation) solvers and digital signal processing. Its main computation is expressed as a two-deep loop nest shown in Figure 2. The stream program implementing the *Laplace* kernel is shown in Figure 3 and the kernel program is omitted.

Consider Figure 2. In lines 1 and 2, two arrays with sizes of *(NR+1)*NC* and *NR*NC*, respectively, are declared, where *NR* and *NC* are compile-time constants. In lines 3 and 4, six streams of size *NC* each are declared. In line 5, four UC variables (i.e., so-called microcontroller variables) are declared. In stream programming, UC variables are passed to a kernel loop as scalar arguments, which are often used in scientific algorithms as the coefficients of math equations or the results of vector reductions. In line 6, the function *dataLoad* is called to initialize array *SMAT*. In lines 7 and 8, the data from the first two sections in *SMAT* are gathered into streams *a1* and *a2*. This will result in the loading of the data from *SMAT* in host memory into the space allocated to the two streams in off-chip memory. In fact, each stream here is a column of array *SMAT*. In lines 10 and 11, streams *a3* and *a4* are initialized similarly. In lines 12 – 15, the four UC scalars are initialized. In line 16, the kernel named *lap* is called to perform the core computation of Laplace on the stream processor. The four streams *a1*, *a2*, *a3* and *a4* and the four scalars are inputs and the two streams *o1* and *o2* are outputs. In lines 17 and 18, these two output streams are stored from off-chip memory back into *OMAT* in host memory. In lines 19 and 20, the program prepares for the next iteration of the *for* loop in line 9 by copying *a3* and *a4* into *a1* and *a2*, respectively, using streamCopies.

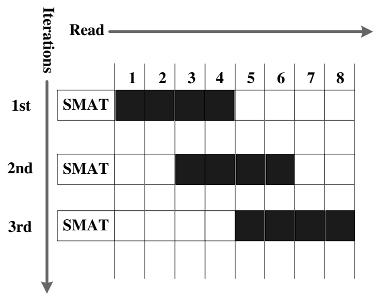


Figure 4: Data reuse pattern among the four input streams *a1*, *a2*, *a3* and *a4* generated during three consecutive iterations of the *for* loop in the stream program for Laplace given in Figure 3.

2.3 Motivation

The Laplace stream program in Figure 3 that is initially written by the programmer suffers from a number of inherent performance problems by itself and other problems if a simple-minded SRF management scheme is used. Below we examine these performance-impacting issues.

A kernel call to *lap* is made in each iteration of the *for* loop (line 9). During each call, four input streams, *a1*, *a2*, *a3* and *a4*, which represent four consecutive data sections (i.e., columns) in array *SMAT*, are consumed. Figure 4 depicts the cross-iteration reuse pattern among these streams. The data sections that are accessed in a loop iteration via *a3* and *a4* are accessed again in the following iteration via *a1* and *a2* (due to the presence of streamCopies in lines 19 and 20). However, such (hidden, input-input) temporal reuse cannot be realized as explained below. This and other performance issues has to be examined under a concrete SRF allocation scheme. For simplicity, we do not consider output streams *o1* and *o2* here.

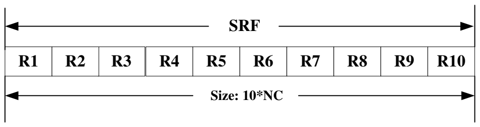


Figure 5: An SRF partition.

Suppose that the SRF has a size of *10*NC* elements and is partitioned into 10 equal-sized SRF registers with each being capable of holding one stream as shown in Figure 5. Suppose further that the SRF allocation algorithm used is First Fit: every stream is allocated to the free SRF register with the smallest index. The allocation result is $\{a1{:}R1,\ a2{:}R2,\ a3{:}R3,\ a4{:}R4\}$. In addition, the compiler inserts streamLoads/streamStores by using a simple heuristic: a streamLoad is inserted just before a kernel call to load an input stream into the SRF and a streamStore is inserted after to store an output stream from the SRF to off-chip memory. The execution trace for Laplace under this simple SRF allocation scheme is illustrated in Figure 6(a). To focus on the key performance-impacting factors only, we assume that all independent operations at a time step can be started simultaneously and that a kernel call takes more time to execute than a single streamCopy operation.

The execution trace in Figure 6(a) reveals two performance problems with the initial stream program in Figure 3. First, parallelism is not exploited since no memory operations can overlap with kernel execution. Second, the data reuse illustrated in Figure 4 is not realized. Under $\{a1{:}R1,\ a2{:}R2,\ a3{:}R3,\ a4{:}R4\}$, the two streamCopies introduced in lines 19 and 20 in Figure 3 are translated into $R1 \leftarrow R3$ and $R2 \leftarrow R4$, which must each be implemented through off-chip memory communications (i.e., via a streamStore followed by a streamLoad). While exhibiting similar semantics as register copies in the scalar architecture, stream copies incur significantly higher overhead. So the total number of streamCopy operations in a stream program must be minimized.

If we unroll the loop in the Laplace stream program by using a factor of 2, we can capture the stream reuse depicted in Figure 4. This is illustrated in Figure 6(b). As a result, the two streamCopies in the stream program have been eliminated. To improve parallelism, however, we must rename *R1* and *R2*, the two of the four input streams for the middle kernel call shown in Figure 6(b) and allocate them to different SRF spaces, say, *R5* and *R6*, in order to eliminate the parallelism-inhibiting data dependences. The execution trace obtained after these optimizations is depicted in Figure 6(c). As a result, the loads into *R5* and *R6* can overlap with the execution of the first kernel call at the cost of two extra streamCopies. Finally, to achieve the best performance for the example, we should choose an unroll factor of 3 as shown in Figure 6(d). In this final solution, all streamCopies are eliminated and the overlap between kernel execution and memory operations is fully exploited.

This example demonstrates that optimizations such as loop unrolling are useful in exposing opportunities for improving stream reuse and parallelism. Given that there can be a large number of streams used in up to hundreds of kernels in a program initially and that more streams may be introduced by various optimizations subsequently, a good SRF allocation strategy is performance-critical.

2.4 Problem Statement

Given a stream program to be compiled for a stream processor (cf. Figure 17), we address the problem of developing a compiler approach to automating its on-chip SRF allocation in order to minimize the overall execution time of the program. Prior to SRF allocation, enabling optimizations such as loop unrolling are applied to expose the stream reuse and parallelism inherent in the stream program. Then SRF allocation is solved as a graph coloring problem. All streams accessed in all kernels are identified as live ranges to be placed in the SRF. Exploiting reuse among streams means coalescing their corresponding live ranges so that the coalesced live range can be assigned to a common SRF space. To enable kernel execution and memory operations to be overlapped, i.e., to maximize parallelism, certain live ranges can be extended to introduce artificial live range interferences. Therefore, the three SRF management tasks, (1) placing streams in the SRF, (2) exploiting stream reuse, (3) and maximizing parallelism, are solved in a unified manner.

The novelty of our approach lies in mapping SRF allocation into a well-studied graph coloring register allocation problem rather than inventing a new ad hoc one. We exploit reuse and parallelism based on the traditional notion of interference graph for capturing live range interferences and a new intermediate representation called *the stream operation dependence graph* (SODG) for capturing stream data flow. Due to the practical efficiency of graph coloring, our approach achieves good performance results as validated by running scientific kernels on our FT64 stream processor.

3. A Compiler Framework

Figure 7 depicts our framework. The four components that are highlighted inside the dashed box are described below. In particular, there are three major phases in compiler-directed SRF management: data reuse exploitation, parallelism exploitation and unroll factor decision. Data reuse exploitation improves performance by capturing stream reuse and thus reducing off-chip bandwidth. Parallelism exploitation improves performance by overlapping kernel execution and memory operations. In the unroll factor decision phase, the best unroll factor for a loop is picked so that the reuse and parallelism inherent in the loop can be maximally exposed.

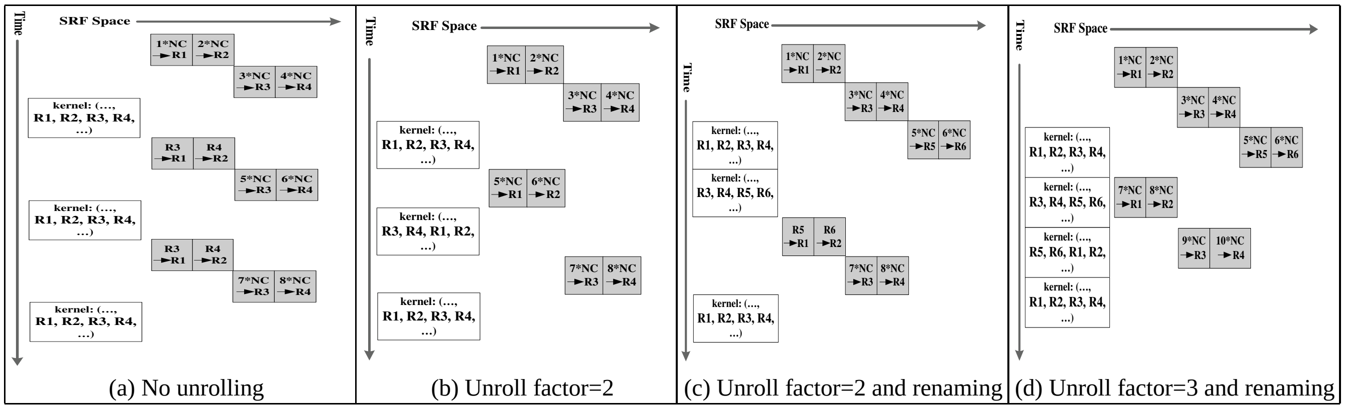


Figure 6: Execution traces of three consecutive loop iterations for Laplace (with two concurrent memory operations assumed).

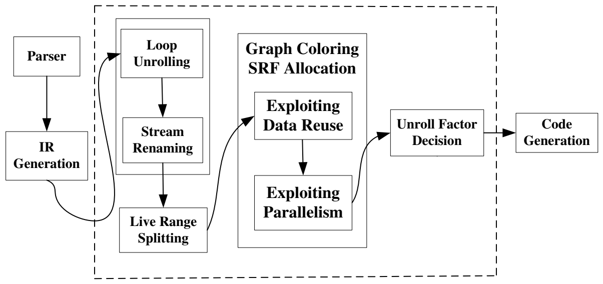


Figure 7: A compiler framework for SRF allocation.

3.1 Loop Unrolling and Stream Renaming

A loop is unrolled by a factor of $\mathcal{J}$, where $\mathcal{J}$ is decided in Section 3.4. Before unrolling, all iterations of the loop share the same loop body and thus the same set of stream variable names. After unrolling, the new loop body is a concatenation of $\mathcal{J}$ instances of the old loop body. The stream variables in the new loop body are renamed so that each instance of the old loop body uses a set of stream variable names that is disjoint from that used by another instance. Renaming is beneficial in eliminating loop-carried dependences, thereby exposing opportunities for overlapping kernel execution and memory operations. The Laplace stream program unrolled with $\mathcal{J} = 3$ and renaming is shown in Figure 8.

```
streamInit(SMAT[0*NC, 1*NC], a1);
streamInit(SMAT[1*NC, 2*NC], a2);

for (k = 0; k < NR; k += 6 ){
    streamInit(SMAT[(k+2)*NC, (k+3)*NC], a3);
    streamInit(SMAT[(k+3)*NC, (k+4)*NC], a4);
    Kernel('lap', ..., a1, a2, a3, a4, o1, o2);
    streamSave(o1, OMAT[k*NC, (k+1)*NC]);
    streamSave(o2, OMAT[(k+1)*NC, (k+2)*NC]);
    streamCopy(a3, a5);
    streamCopy(a4, a6);
    streamInit(SMAT[(k+4)*NC, (k+5)*NC], a7);
    streamInit(SMAT[(k+5)*NC, (k+6)*NC], a8);
    Kernel('lap', ..., a5, a6, a7, a8, o3, o4);
    streamSave(o3, OMAT[(k+2)*NC, (k+3)*NC]);
    streamSave(o4, OMAT[(k+3)*NC, (k+4)*NC]);
    streamCopy(a7, a9);
    streamCopy(a8, a10);
    streamInit(SMAT[(k+6)*NC, (k+7)*NC], a11);
    streamInit(SMAT[(k+7)*NC, (k+8)*NC], a12);
    Kernel('lap', ..., a9, a10, a11, a12, o5, o6);
    streamSave(o5, OMAT[(k+4)*NC, (k+5)*NC]);
    streamSave(o6, OMAT[(k+5)*NC, (k+6)*NC]);
    streamCopy(a11, a1);
    streamCopy(a12, a2);
}
```

Figure 8: Unrolled and renamed Laplace stream program ($\mathcal{J} = 3$).

3.2 Live Range Splitting

This is applied to all kernel calls and streamCopies in a stream program. The objective is to identify the program points at which a stream should be potentially loaded into and evicted from the SRF. As a result, all live ranges to be assigned to the SRF are identified.

For every stream *a* consumed or produced by a kernel call, its live range is split before and after the call. We add appropriate streamLoads/streamStores to connect these split live ranges. If *a* is an input stream, we add *streamLoad(a, Sa)* before the kernel call to load *a* from off-chip memory into *Sa*, which is a pseudo SRF register, i.e., a live range to be placed in the SRF. If *a* is an output stream, we add *streamStore(Sa, a)* after the kernel call to store the pseudo SRF register *Sa* into *a* residing in off-chip memory. In addition, every reference to *a* in the kernel call is replaced by *Sa*.

Every *streamCopy(a, b)* is replaced with "*streamLoad(a, Sa); streamStore(Sa, b)*", i.e., a streamLoad followed by a streamStore.

After splitting, all pseudo SRF registers (whose names begin with an 'S') are the live ranges to be assigned to the SRF. Note that different live ranges of a stream are renamed as is done in the SSA form in order to precisely characterise live range interferences.

Applying live range splitting to Laplace gives rise to Figure 9. Let us explain the reasons behind our splitting scheme. All pseudo SRF registers are the live ranges to be assigned to the SRF. All the others are assumed to be spilled to off-chip memory. In addition, all pseudo SRF registers must be assigned to the SRF, i.e., cannot be spilled as the SRF is non-bypassing. These two properties are what distinguish the splitting strategy used here from those applied to register allocation for scalars [5] and arrays [17]. Therefore, inserting streamLoads before a kernel call ensures that the data records in its input streams can be loaded into the SRF before they are accessed. For a similar reason, streamStores after a kernel call are inserted for its output streams. Furthermore, the programmer typically calls *streamSave* after some kernel calls to save application results to the host memory (lines 10 an 11 in Figure 9). Inserting streamStores this way also guarantees program correctness.

Performing splitting around each kernel makes explicit all live ranges that must be assigned to the SRF, ensuring that an SRF allo-

```
...
streamInit(SMAT[(k+2)*NC, (k+3)*NC], a3);
streamInit(SMAT[(k+3)*NC, (k+4)*NC], a4);
streamLoad(a1, Sa1);
streamLoad(a2, Sa2);
streamLoad(a3, Sa3_1);
streamLoad(a4, Sa4_1);
Kernel('lap', ..., Sa1, Sa2, Sa3_1, Sa4_1, So1, So2);
streamStore(So1, o1);
streamStore(So2, o2);
streamSave(o1, OMAT[k*NC, (k+1)*NC]);
streamSave(o2, OMAT[(k+1)*NC, (k+2)*NC]);
streamLoad(a3, Sa3_2);
streamStore(Sa3_2, a5);
streamLoad(a4, Sa4_2);
streamStore(Sa4_2, a6);
...
```

Figure 9: Laplace stream program obtained from applying live range splitting to Figure 8 (with only the transformed code corresponding to lines 4 – 10 in Figure 8 being shown explicitly).

cation can be easily found for the non-bypassing SRF. In addition, stream reuse and parallelism can be exploited by manipulating the interference graph formed by all live ranges. Finally, unnecessary splits can be coalesced, i.e., removed during graph coloring.

3.3 Graph Coloring SRF Allocation

Once live range splitting is done, the interference graph (IG) for all pseudo SRF registers, i.e., all live ranges in a stream program is built in the standard manner: a node denotes a live range and an edge connects two nodes if the two corresponding live ranges overlap, i.e., interference. As a result, no two interfering nodes can be assigned to overlapping SRF spaces in the SRF.

To assign pseudo SRF registers in the SRF, we adopt the scratchpad partitioning idea introduced in [17] for allocating arrays in embedded programs to scratchpad memory. First, all pseudo SRF registers introduced in live range splitting are clustered into equivalent classes so that all streams of a common size are in the same class. Before doing so, the sizes of streams may be aligned, i.e., normalized to a pre-defined constant value as a pseudo SRF register may be required to start at an address that is a multiple of some constant value. This normalization step also helps minimize the number of equivalent classes generated. Then the SRF is partitioned into a register file as follows. For each stream class of a particular size, the SRF is partitioned into a register class such that each register in the class can hold exactly one stream of that size. All registers in the same register class are interchangeable. However, two SRF registers in different classes may be aliased to each other if their SRF spaces overlap. Therefore, SRF partition has resulted in a register file with interchangeable and aliased SRF registers.

For the Laplace program given in Figure 8, all streams have the same length, *NC*. So there is only one equivalent class that consists of all streams in the program. The register file obtained when the SRF size is assumed to be *10*NC* is shown in Figure 5.

Given an IG built for pseudo SRF registers and an SRF partitioned into SRF registers, the SRF allocation problem has therefore been formulated as a graph coloring problem. The problem can now be solved by applying a graph coloring algorithm as generalized in [24] to handle interchangeable and aliased registers.

Recall that there are three SRF management problems: (1) placing streams in the SRF, which amounts to mapping every pseudo SRF register to an SRF register, (2) exploiting stream use, and (3) maximizing parallelism. We solve the three problems in two stages, *reuse exploitation* (for (1) and (2)) and *parallelism exploitation* (for (3)). The primary reason for adopting this two-stage approach is that the first stage generally delivers far greater performance improvements than the second. The secondary reason is that both optimizations affect colorability in a different way. In reuse exploitation via graph coloring, reuse is achieved by coalescing. Coalescing two live ranges implies that the two live ranges will be assigned to the same SRF register. It is well known that coalescing has both positive and negative impact on colorability [23]. On the one hand, coalescing may degrade the colorability by increasing the degrees of some nodes. On the other hand, coalescing may improve the colorability by reducing the degrees of some neighboring nodes that are interfering with the two live ranges being coalesced. In parallelism exploitation via graph coloring, some live ranges are extended in order to overlap kernel execution with memory operations. As a result, live range extension always degrades the colorability. Given these reasons, reuse is exploited first to produce an initial SRF allocation and parallelism is then exploited by fine tuning the initial SRF allocation.

3.3.1 Exploiting Data Reuse

We can now find an SRF allocation with a good degree of stream reuse by applying any iterative-coalescing graph coloring algorithm as generalized [24]. In comparison with iterative-coalescing register allocation in allocating scalars to registers [5] and arrays to scratchpad memory [17], there are two differences detailed below.

The first difference is that the SRF is non-bypassing while scalar registers and scratchpad are generally not. Thus, all streams accessed by a kernel must be made available in the SRF before the kernel is executed. In other words, no live range in the IG should be spilled during coloring — spilling has already been done in live range splitting. To guarantee the existence of an SRF allocation, the double buffering technique as described in [7] can be used. The basic idea is to allocate a double-buffered stream to an SRF register smaller than its size. When the kernel processes the stream in one half of the register, the next part of the stream is concurrently fetched into the other half. Technically, double buffering ensures that a valid SRF allocation can always be found for a program. In practice, however, double buffering should be avoided, if possible, since it can be costly. Instead, enabling transformations such as loop tiling [30] can be applied to reduce the data set accessed by a kernel before an SRF allocation is searched for again.

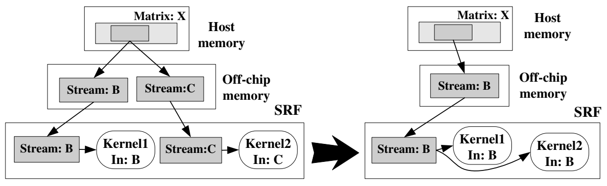


Figure 10: Input-input reuse and live range coalescing.

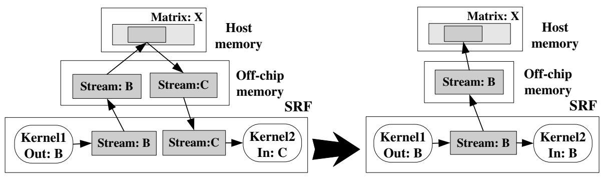


Figure 11: Output-input reuse and live range coalescing.

The second difference is that we need to develop a different coalescing heuristic to coalesce streams that are reuse-related. The concepts of input-input and output-input reuse for streams are carried over directly from those for scalars and arrays. These two kinds of stream reuse are illustrated in Figures 10 and 11. Obviously, coalescing, i.e., removing unnecessary live range splits avoids redundant memory operations, reducing off-chip bandwidth. To this end, we introduce a new type of so-called *reuse-related* edges to the IG. Every reuse-related edge connects two streams with either input-input or output-input reuse. A number of coalescing techniques can be used to coalesce these reuse-related nodes to exploit data reuse. However, conservative coalescing [2] gives up coalescing too early, losing many opportunities for coalescing that would turn out to be safe. On the other hand, optimistic coalescing [23], which performs first aggressive coalescing and then de-coalescing, can be too costly since a large number of nodes can be reused-related. Furthermore, in both cases, the structure of the program is not considered.

Algorithm 1 Reuse Exploitation

```
 1: procedure Alloc_and_Reuse_Exploit (IG)
 2: while (there are unprocessed reuse-related edges in IG) do
 3:     Incremental_Coalesce();
 4:     Build();
 5:     Simplify();
 6:     Potential_Spill();
 7:     if ($Select() == false$) then
 8:         Decoalesce();
 9:         Freeze();
10:     end if
11: end while
12: end procedure
```

Our algorithm for SRF allocation and reuse exploitation is given in Algorithm 1. We introduce an *incremental coalescing* heuristic to coalesce reuse-related nodes. We add all reuse-related edges to the IG. The IG thus constructed for Laplace is depicted in Figure 12. We sort all reuse-related node pairs by their execution distances. The *execution distance* of a pair of reuse-related nodes is defined to be the (average) number of kernel calls executed between the two stream operations associated with the two reuse-related nodes. In the presence of if statements, the execution distance is the weighted average distance of all possible paths (with their execution frequencies being their weights). The node pair with the smallest execution distance has the highest priority to be coalesced. In line 3, we pick from the current IG an unprocessed reuse-related node pair with the smallest execution distance. In line 4, *Build* is called to rebuild the IG. In lines 5 and 6, *Simplify* and *Potential_Spill* perform their usual duties as described in [9]. In line 7, *Select* returns *false* if a coloring without spilling, i.e., an SRF allocation cannot be found, in which case, we give up the coalescing in lines 8 and 9 by ignoring this reused-related edge. We repeat this process until all reused-related nodes have been processed. If a valid allocation is not found, loop tiling or double buffering as discussed earlier can be applied before coloring is re-started. In future work, this part will also be automated.

Suppose the SRF considered is as given in Figure 5. For the Laplace program listed in Figure 9, the SRF allocation result is shown in Figure 13. For this example, all reuse-related live ranges have been successfully coalesced. For example, live ranges *Sa3_1*, *Sa3_2* and *Sa5* are coalesced. As a result, the memory operations in lines 12 and 13 shown in Figure 9 can be eliminated.

3.3.2 Exploiting Parallelism

To maximize parallelism, we should fully utilize the SRF space by allocating streams to non-overlapping SRF registers, i.e., regions so that kernel execution and memory accesses can be carried out concurrently. In reuse exploitation via graph coloring, two non-interfering streams may be assigned the same color, i.e., the same SRF register. As a result, the corresponding load or store memory operations are sequentialized. In parallelism exploitation, one of the live ranges for the two streams can be extended to create an artificial interference between the two streams. By re-applying graph coloring to the modified IG, both streams may be assigned to different SRF registers if the SRF is not fully utilized.

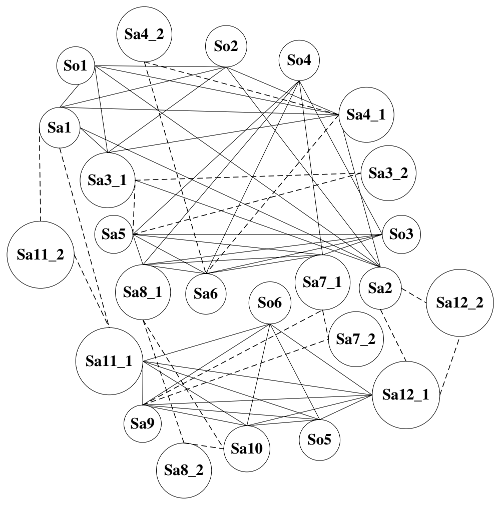


Figure 12: IG with reuse-related edges for Figure 9 (with all reuse-related edges shown in dashed lines).

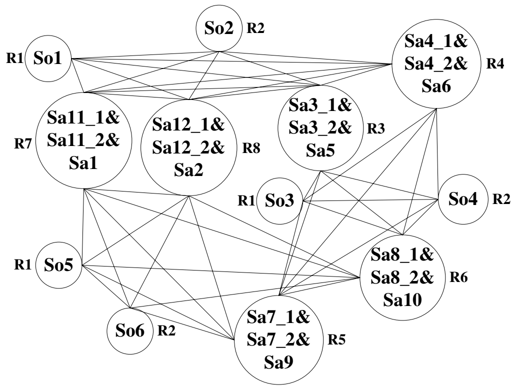


Figure 13: SRF allocation after reuse exploitation for the Laplace program given (partially) in Figure 9 w.r.t. the SRF in Figure 5.

After data reuse exploitation is done, an SRF allocation has been obtained. Parallelism exploitation is performed in the same iterative-coalescing graph coloring framework, starting with the IG that gives rise to the SRF allocation obtained in reuse exploitation. The basic idea is to perform live range extension on the IG to uncover more parallelism while maintaining its colorability. Certain live ranges are extended, one at a time. Graph coloring is re-started on each modified IG. A live range extension is successful if an SRF allocation can still be found and rejected otherwise. More parallelism is introduced on each successful live range extension.

In loop-oriented scientific programs, most of their execution times are spent on a few compute-intensive loops. So parallelism exploitation is applied to individual loops. For each loop that is already transformed according to the SRF allocation found in reuse exploitation, we introduce a so-called *Stream Operation Dependence Graph* (SODG) to represent the data flow between its stream operations. In the graph, a node represents a stream operation and a directed edge between two stream operations represents their data

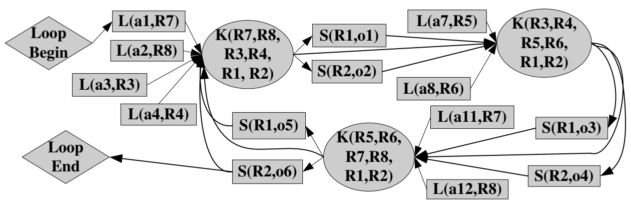


Figure 14: SODG for Figure 9 after reuse exploitation.

dependences. The SODG for the unrolled *Laplace* stream program given in Figure 9 and transformed according to the SRF allocation given in Figure 13 is shown in Figure 14. To save space in drawing SODGs, some abbreviations are used so that *streamLoad(a1, R7)* is written as *L(a1, R7)*, *streamStore(R1, o1)* to *S(R1, o1)* and *Kernel('lap', ..., R7,R8,R3,R4,R1,R2)* to *K(R7,R8,R3,R4,R1,R2)*.

In the SODG, each node Ri represents a stream that is already allocated to the SRF register with that name. In reuse exploitation, assigning two streams that are not reuse-related to the same register introduces a name or memory dependence rather than a value dependence via that register. Consider output streams *o1*, *o3* and *o5* in the Laplace stream program in Figure 8, where oi is the first of the two output streams of the i-th kernel call made in each iteration of the Laplace loop. In reuse exploitation, their split live ranges *So1*, *So3* and *So5* given in Figure 9 are all allocated to the same SRF register, *R1*, as shown in Figure 13. As a result, as shown in the SODG depicted in Figure 14, any of the three kernels cannot start until the streamStore for storing the data produced into *R1* by the previous kernel has completed. For example, there is a dependence from *S(R1, o1)* to the second kernel call, *K(R3,R4,R5,R6,R1,R2)*. So both operations cannot be overlapped. However, if *So1*, *So3* and *So5* can be allocated to different registers, then operations such as *S(R1, o1)* and *K(R3,R4,R5,R6,R1,R2)* can be overlapped. In this case, we would like to extend the live ranges of *So1*, *So3* and *So5* downwards to obtain delayed stores so that these stores can be overlapped with kernel execution. Conversely, extending the live ranges of some input streams upwards across a kernel call will result in prefetching load operations (if the SRF has enough room).

Our iterative algorithm for exploiting parallelism is presented in Algorithm 2. In lines 3 - 8, the amount of SRF space taken by all streams consumed and produced by each kernel is computed to see if the SRF has room for hiding memory access latency (by tolerating prefetching loads or delayed stores). In lines 9 - 18, all kernels are processed one after another such that a kernel that takes less SRF space is processed earlier than a kernel that takes more. Let k be the current kernel being examined. For each streamLoad that depends on k in the SODG (lines 11 - 13), we try to extend the live range of the corresponding input stream upwards to see if it can be turned into a prefetching load (which can then run concurrently with k). For each streamStore on which k depends in the SODG (lines 14 - 16), we try to extend the live range of the corresponding input stream downwards to see if it can be turned into a delayed store (which can then run concurrently with k).

The procedure *TryToParallelize*(k, p) (not given here) checks if the live range of the stream associated with p can be safely extended. Two conditions must be met. First, the stream cannot be aliased with any stream consumed or produced by k. This is accomplished by a data-flow analysis on streams. If p is a load, then the live range is extended upwards to just pass k. If p is a store, then the live range is extended downwards to pass just k. Second, the colorability of the IG must be retained. This is ensured by applying graph coloring to the modified IG. The live range extension is accepted if both conditions are satisfied and ignored otherwise.

Algorithm 2 Parallelism Exploitation

```
 1: procedure Parallelism_Exploit (IG)
 2: Build a SODG, G(V, E), from the given IG
 3: for every kernel node $k \in V$ do
 4:     $S_k.size = 0$
 5:     for every live range $s$ that is live at $k$ do
 6:         $S_k.size += s.size$
 7:     end for
 8: end for
 9: repeat
10:     Select an unprocessed kernel $k$ such that $S_k.size$ is the
        smallest
11:     for every streamLoad node $p$ that depends on $k$ do
12:         $TryToParallelize(k, p)$
13:     end for
14:     for every streamStore node $p$ on which $k$ depends do
15:         $TryToParallelize(k, p)$
16:     end for
17: until all kernel nodes have been processed
18: end procedure
```

For the Laplace program, the final SRF allocation after parallelism exploitation is shown in Figure 15. From the SRF allocation found by reuse exploitation and shown in Figure 13, we see that the first output streams of the three kernel calls, *So1*, *So3* and *So5*, are allocated to *R1* and the second output streams of the three kernel calls, *So2*, *So4* and *So6*, are allocated to *R2*. After parallelism exploitation, *So3* is colored differently from *So1* and *So5*, and *So4* is colored differently from *So2* and *So6*. As a result, the better overlap between these store operations and kernel execution can be expected. (Given a larger SRF, all these live ranges can be assigned to different SRF registers, resulting in even more parallelism.)

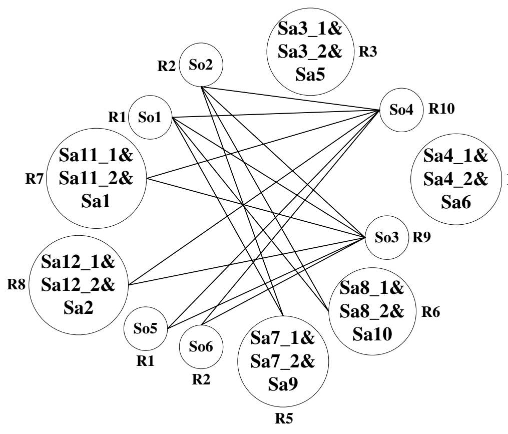


Figure 15: SRF allocation for Figure 13 after parallelism exploitation. To avoid cluttering, the interference edges in the IG in Figure 13 are omitted. The only new ones introduced are shown.

3.4 Unroll Factor Decision

After SRF allocation is done for a program, the generated code is evaluated. Presently, a simple strategy is used. Given the SODG for a loop with its *Loop Begin* and *Loop End* nodes, we find a longest path in between under the assumption that all edges have the same weight of 1. We then obtain a time indicator by dividing the length of the longest path by the unroll factor used. For example, the final SODG for *Laplace* (with $\mathcal{J} = 3$) after parallelism exploitation is shown in Figure 16. The longest path is: Loop Begin→L(a1, R7)→K→K→K→S(R2, o6) →Loop End, with a length of 6. The time indicator is $6/\mathcal{J} = 2$. The unroll factor with the smallest time indicator will be selected. Alternative methods for choosing unroll factors will be investigated in future work.

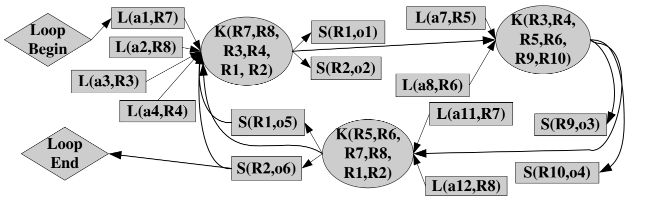
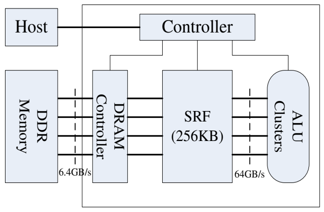


Figure 16: SODG for Figure 9 after parallelism exploitation.

4. Evaluation

We evaluate our compiler framework by running scientific kernels on FT64, a 64-bit stream processor for scientific computing that we have recently designed and fabricated [31]. A simplified sketch of FT64 is shown in Figure 17. FT64 runs as a co-processor to a host processor. The SRF (of 256KB) is the nexus of FT64. FT64 is composed of four clusters (with eight ALUs in each cluster), which execute the same kernel concurrently in a SIMD fashion. The compiler framework described in this paper is part of two modules, Code Optimization and SRF Allocation, in the stream compiler developed for FT64 [31] as highlighted in gray in Figure 18.



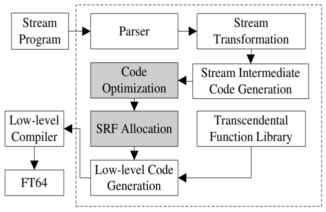


Figure 17: FT64 processor. Figure 18: FT64 compiler.

The nine representative scientific kernels used are given in Table 1. (`NLAG-5` is a nonlinear algebra solver for two-dimensional nonlinear diffusion of hydrodynamics.) These benchmarks, which are initially in FORTRAN, have been rewritten into stream programs. Stream programs are executed on FT64 as well as simulated by a cycle-accurate simulator. From simulations, we are able to collect statistics such as memory access information and execution traces. For comparison purposes, the original FORTRAN programs are compiled by Intel's *ifort* compiler under *-O3* and then executed on a 1.6GHz single-core Itanium 2 server equipped with a 4GB off-chip memory with the bandwidth of 6.4GB/s. The cache sizes are 16KB each for both L1 instruction and data caches, 256KB for the unified L2 cache and 6MB for the unified L3 cache.

Benchmark	Source	Problem Size
Swim-calc1	Spec2000	512×512
Swim-calc2	Spec2000	512×512
EP	NPB	1024×128
FFT	-	4096
GEMM	BLAS	512×512
Jacobi	-	512×512
Laplace	NCSA	512×512
NLAG-5	-	512×512
MG	NPB	$128 \times 128 \times 128$

Table 1: Scientific applications.

Figure 19 shows the performance improvements of the nine programs on FT64 relative to Itanium 2. Good speedups are observed in all except `Swim-calc1` and `NLAG-5`. FFT is the best performer since it also benefits from the hardware support in FT64 for the FFT butterfly computation and bit-reversing operations.

Figure 20 gives the same performance results of running the nine programs on FT64 measured in GFLOPS.

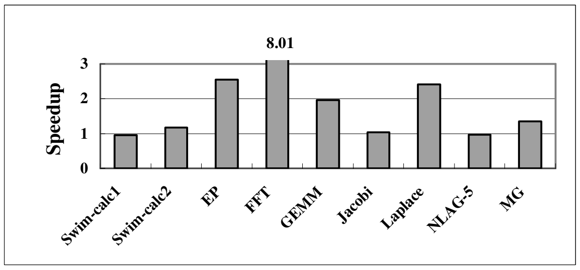


Figure 19: Performance speedups on FT64 relative to IA64.

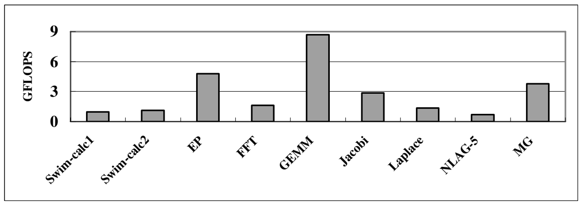


Figure 20: Performance results on FT64 in GFLOPS.

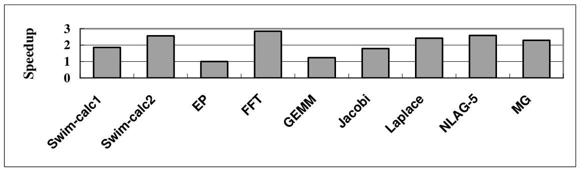


Figure 21: Performance speedups of graph coloring over First-Fit.

Figure 21 shows the performance improvements achieved by our compiler framework over First-Fit Allocation. Good speedups are observed in all programs except `EP`. This shows that our graph coloring formulation is generally effective in placing streams in the SRF, exploiting stream reuse and maximizing parallelism.

Figure 22 presents the performance results achieved on FT64 under three different unroll factors (with $\mathcal{J} = 1$ denoting the absence of loop unrolling). The execution time of a benchmark under an unroll factor is normalized to that achieved when $\mathcal{J} = 1$. In general, a benchmark runs faster as $\mathcal{J}$ increases. However, `EP` and `Jacobi` defy this trend. The reasons behind are analyzed below. `EP` exhibits strong loop-carried dependences. As shown in Figure 23, there are hardly any memory operations that be overlapped with kernel execution. As for `Jacobi`, its performance improves when $\mathcal{J}$ increases from 1 to 2 but drops slightly when $\mathcal{J}$ increases from 2 to 3. A similar analysis reveals that the maximum parallelism is achieved when $\mathcal{J} = 2$. Furthermore, since the loop trip count is not divisible by 3, the prolog and epilogue code causes some slightly more performance degradation when $\mathcal{J} = 3$.

We have also evaluated the individual and combined effects of exploiting reuse and parallelism on performance in Figure 24. In Algorithm 1, reuse exploitation can be switched off if coalescing is not applied, i.e., if lines 3, 8 and 9 are ignored. In this case, the algorithm is only responsible for allocating streams to the SRF.

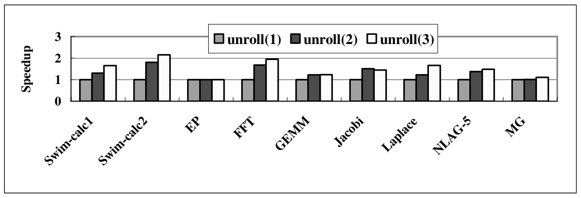


Figure 22: Performance speedups on FT64 under three different unroll factors (relative to when the unroll factor is $\mathcal{J} = 1$).

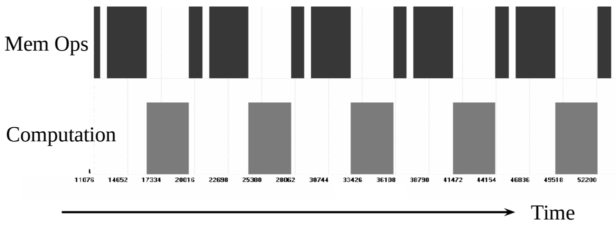


Figure 23: Execution trace across five loop iterations in `EP` ($\mathcal{J} = 3$). Memory units are shown at the top while clusters at the bottom (with dark bands for busy cycles and light regions for stalls).

The execution time of a benchmark is normalized to that achieved when neither optimization is performed. In general, data reuse exploitation is more beneficial than parallelism exploitation. However, this does not mean that parallelism exploitation is not important. For some programs, a large amount of ILP and DLP can already be exploited during kernel execution. As a result, any further performance improvement obtained from exploiting parallelism at SRF allocation can be limited. In particular, we also discover that `NLAG-5` achieves the best performance when reuse exploitation is applied alone. The reasons behind are analyzed below.

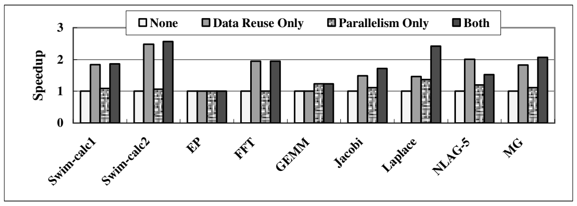


Figure 24: Effects of exploiting reuse and parallelism on performance with $\mathcal{J} = 3$ (against the baseline when neither is applied).

In `NLAG-5`, the reuse pattern among its input streams is similar to that in *Laplace* (Figure 4). When parallelism is exploited, the data sets for the three kernel calls in the unrolled loop are: $\{s1, s2, s3\}$, $\{s2, s3, s4\}$ and $\{s3, s4, s1\}$. The prefetching of *s4* and *s1* is done at the cost of two streamCopies, *streamCopy(s1, s2)* and *streamCopy(s4, s1)*. If parallelism is not exploited, the data sets become $\{s1, s2, s3\}$, $\{s2, s3, s1\}$ and $\{s3, s1, s2\}$. Here, no prefetches are performed but no extra streamCopies are needed either. When $\mathcal{J} = 3$, exploiting both reuse and parallelism is not as fruitful as exploiting reuse alone. However, both optimizations are beneficial at the optimal unroll factor $\mathcal{J} = 4$.

5. Related Work

There have been some attempts on improving scientific computing performance on stream processors [6, 4, 28, 31]. However, their main focuses are on evaluating their architectures rather on developing compiler optimizations. A study reported in [12] is concerned with mapping stream programs to general-purpose processors. Liao et al [27] apply data and computation transformations to map stream programs to multiprocessors rather than stream processors. As a result, their focus is on cache optimizations.

The StreamIt project [1, 11, 26] has resulted in a number of novel compiler optimizations targeted at the Raw architecture [25] with emphasis on exploiting the parallelism within and across the filters in a program. Data reuse exploitation is less relevant. The Raw architecture is tile-based and exposes gate, wire and pin resources to software. So it is different from many other stream processors [6, 22, 28, 31] that allow themselves to be mapped easily to the stream virtual machine architecture described in [15].

Stream scheduling introduced in [7] was earlier implemented in the StreamC compiler to compile stream programs for the Imagine stream processor [7, 22]. Stream scheduling captures the sizes and access patterns of stream accesses and the data flow between kernels by partial evaluation. All stream accesses to the same stream are associated (if possible) with the same buffer in the SRF. All such SRF buffers are placed in the SRF by applying packing-like heuristics. Buffers can be extended with so-called shadows to maximize parallelism. This paper presents a graph coloring approach to automating this process. Furthermore, loop unrolling is applied to uncover the stream reuse and parallelism inherent in a program. Instead of just output-input reuse, input-input reuse (which is rare in media applications and thus ignored in [7] but can be abundant in scientific applications) is exploited. Knight et al [14] present compiler techniques for explicitly managed memory hierarchies and discuss some data reuse optimizations such as Copy Elimination. They perform this optimization by using a tree of memories as an execution model of the underlying machine with IR transformations being defined with respect to this execution model.

In [19, 29], compiler techniques for compiling kernel programs to stream processors are introduced.

Graph coloring is a popular technique used in register allocation. Based on Chaintin's original formulation [3], a variety of graph coloring based register allocators have been developed [2, 5, 9]. Recently, Smith, Ramsey and Holloway [24] present a generalized algorithm for irregular architectures with register aliases and non-disjoint register classes. Li, Gao and Xue [17] apply this generalized algorithm to assign arrays in embedded programs to scratchpad memory (SPM). The key idea behind is to partition the SPM into a register file and then apply graph coloring to assign arrays to the SPM registers thus created. This paper applies graph coloring to SRF allocation. However, there are two fundamental differences between the two approaches. First, target architectures are different. In [17], embedded uniprocessors with SPMs are assumed. In this work, stream processors are considered. Second, optimization objectives are different. In [17], the optimization problem is somewhat simpler since maximizing the SPM hit rate amounts to maximizing program performance. In this work, we need to improve the performance of a program by not only maximizing the data reuse among streams in the SRF but also aggressively overlapping kernel execution and memory operations. As a result, optimizations such as loop unrolling and live-range extension that are performance-critical to stream processors are not considered in [17].

A traditional approach to managing data aggregates such as arrays is to formulate the problem as an interval coloring problem. Fabri [8] discovered earlier the connection between interval coloring and compile-time memory allocation. Since then some approximation algorithms have been proposed [10, 13]. Lefebvre and Feautrier [16] use interval coloring to minimize the number of data structures to rename in storage management for parallel programs. Li, Nguyen and Xue [18] allocate arrays to SPMs by taking advantage of the fact that the IGs in many embedded applications are superperfect graphs or nearly so.

6. Conclusion

In this paper we have presented a graph coloring compiler framework to automate on-chip SRF storage allocation for optimizing scientific applications on stream processors. We have evaluated its performance by running a number of scientific programs on a 64-bit stream processor, FT64, that we have recently designed and fabricated. Our results show that graph coloring compiler management is more effective than simple-minded heuristics such as First-Fit. Together with enabling optimizations such as loop unrolling to expose the stream reuse and parallelism inherent in a program, our graph coloring SRF allocator can achieve good performance results on our stream processor (and better results overall than those on Itanium 2). To the best of our knowledge, this is the first paper applying graph coloring to SRF management and demonstrating its usefulness in optimizing scientific applications on a 64-bit stream processor. This also appears to be the first paper applying loop unrolling and evaluating its effectiveness to SRF management in this setting. Although our experiments are carried out on FT64, we expect our optimizations to be also useful to other stream processors such as Merrimac [6], Imagine [22], Cell [28], and any other processor that embraces a software-managed on-chip memory that serves as a nexus for the processor.

7. Acknowledgement

This work is supported by a NSFC grant (60621003) and an Australian Research Council (ARC) Grant (DP0881330).

References

[1] Sitij Agrawal, William Thies, and Saman Amarasinghe. Optimizing stream programs using linear state space analysis. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 126–136, 2005.

[2] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[3] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, New York, NY, USA, 1982.

[4] Sourav Chatterji, Manikandan Narayanan, Jason Duell, and Leonid Oliker. Performance evaluation of two emerging media processors: Viram and imagine. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 229.1, Washington, DC, USA, 2003. IEEE Computer Society.

[5] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12 (4):501–536, 1990.

[6] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, 2003.

[7] Abhishek Das, William J. Dally, and Peter 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.

[8] Janet Fabri. Automatic storage optimization. *SIGPLAN Not.*, 14(8): 83–91, 1979.

[9] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[10] Jordan Gergov. Algorithms for compile-time memory optimization. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 907–908, Philadelphia, PA, USA, 1999.

[11] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[12] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2005.

[13] H. A. Kierstead. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Math.*, 87(2-3):231–237, 1991.

[14] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 226–236, New York, NY, USA, 2007.

[15] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The stream virtual machine. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–277, Washington, DC, USA, 2004. IEEE Computer Society.

[16] V. Lefebvre and P. Feautrier. Storage management in parallel programs. Technical report, Laboratory PRiSM, University of Versailles, France, 1996.

[17] Lian Li, Lin Gao, and Jingling Xue. Memory coloring: A compiler approach for scratchpad memory management. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 329–338, 2005.

[18] Lian Li, Quan Hoang Nguyen, and Jingling Xue. Scratchpad allocation for data aggregates in superperfect graphs. volume 42, pages 207–216, New York, NY, USA, 2007. ACM.

[19] Peter Mattson, William J. Dally, Scott Rixner, Ujval J. Kapasi, and John D. Owens. Communication scheduling. *SIGARCH Comput. Archit. News*, 28(5):82–92, 2000.

[20] Peter Raymond Mattson. *A programming system for the imagine media processor*. PhD thesis, Stanford University, Stanford, CA, USA, 2002. Adviser-William J. Dally.

[21] John D. Owens. *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, November 2002.

[22] John D. Owens, Ujval J. Kapasi, Peter Mattson, Brian Towles, Ben Serebrin, Scott Rixner, and William 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] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[24] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.

[25] Michael Bedford Taylor, Jason Kim, and Jason Miller et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[26] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[27] Shih wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan 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.

[28] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[29] Nan Wu, Mei Wen, Ju Ren, Yi He, and Chunyuan Zhang. Register allocation on stream processor with local register file. In *ACSAC '06: Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference*, pages 545–551, 2006.

[30] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Boston, 2000.

[31] Xuejun Yang, Xiaobo Yan, Zuocheng Xing, Yu Deng, Jiang Jiang, and Ying Zhang. A 64-bit stream processor architecture for scientific applications. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 210–219, 2007.