Scientific Computing Applications on a Stream Processor

Ying Zhang, Xuejun Yang, Guibin Wang, Ian Rogers†, Gen Li, Yu Deng and Xiaobo Yan
PDL, School of Computer, National University of Defence Technology, ChangSha, China
†The University of Manchester, Manchester, UK
Email: {zhangying, xjyang, gbw, genli, dy, yxb@nudt.edu.cn}, ian.rogers@manchester.ac.uk

Abstract

Stream processors, developed for the stream programming model, perform well on media applications. In this paper we examine the applicability of a stream processor to scientific computing applications. Eight scientific applications, each having different performance characteristics, are mapped to a stream processor. Due to the novelty of the stream programming model, we show how to map programs in a traditional language, such as FORTRAN. In a stream processor system, the management of system resources is the programmers' responsibility. We present several optimizations, which enable mapped programs to exploit various aspects of the stream processor architecture. Finally, we analyze the performance of the stream processor and the presented optimizations on a set of scientific computing applications. The stream programs are from 1.67 to 32.5 times faster than the corresponding FORTRAN programs on an Itanium 2 processor, with the optimizations playing an important role in realizing the performance improvement.

1. Introduction

Scientific computing plays an important role in the research and industry; it features massive amounts of data, intensive computations, and large amounts of parallelism. Currently general purpose architecture processors cannot meet some of the demands of the scientific computing applications, such as large amounts of bandwidth, large amounts of processing capability, low power and low price. Stream processors [1]–[4] have demonstrated significant performance advantages in media applications [5]–[7]. Many researchers are are interested in the applicability of stream processors to scientific computing applications [4], [8]–[10]. This paper provides an experimental evaluation of scientific computing applications on a stream processor.

The stream processor architecture is designed to implement the stream programming model [11], which has many differences from the architecture of a conventional system. Although language implementations, such as streamC/kernelC [12], Brook [13], and Sequoia [14], exploit the model's features well, they do so at such a comparatively low-level; it is mainly the programmer's responsibility to manage system resources. Moreover, compared to other stream applications, such as media applications, scientific computing applications have more complex data traces and stronger data dependence. Therefore, writing a high-performance scientific stream program is rather hard and important to get right.

In this paper, we use the language streamC/kernelC [12] to map FORTRAN versions of scientific applications to the stream processor. A general method is first given to map applications to the stream processor; optimizations are then proposed to improve the overall performance of the mapped stream programs. Finally, the applicability of the stream processor for scientific applications and the effectiveness of our optimizations are measured through a number of experiments. In addition, we discuss the implementation of our optimizations in a compiler.

The rest of this paper is organized as follows: Sect. 2 presents a background to stream processing; Sect. 3 describes the general implementation of scientific computing applications on the stream processor; Sect. 4 details the optimizations used in this paper; In Sect. 5, we evaluate the performance of a stream processor compared to a general purpose processor and the effect of the presented optimizations; Sect. 6 discusses the automatic implementation of the presented optimizations in a compiler; Finally Sect. 7 draws conclusions from this work.

2. Background

Stream Programming Model

In the stream programming model as depicted in Fig.1, the data primitive is a *stream*, an ordered set of data of an arbitrary type. Operations in the stream programming model are expressed as operations on entire streams. These operations include stream loads/stores from/to the memory, stream transfers over a multi-node network, and computations in the form of kernels.

A *kernel* is a computation-intensive function that performs computations on entire streams by applying a function to each element of the stream in sequence. Kernels operate on one or more streams as inputs and produce one or more streams as outputs. Kernels are restricted to only operate on local data; a kernel's outputs are functions only of their inputs, and kernels may not make arbitrary memory references.

978-1-4244-2232-6/08/$25.00 ©2008 IEEE

A stream program is constructed by chaining stream operations together. Programs expressed in this model are specified at two levels: the stream level and the kernel level. A simple stream program that transforms a series of points from one coordinate space to another, for example, might consist of three stream operations specified at the stream level: a stream load to bring the input stream of records onto the on-chip memory, a kernel to transform those records to the new coordinate system, and a stream save to put the output stream of transformed points back into off-chip memory.

There are two kinds of streams: basic streams and derived streams. A basic stream is an array of records, defined by a size. A derived stream is a reference to a subset of the records in a basic stream, defined by a basic stream and a start, end, and access pattern within that stream. The start is the index of the first record in the stream. The end is the index of the record after the last record that could be in the stream. The access pattern defines which records, between the start and the end, are in the stream.

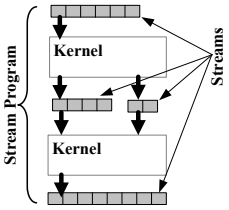


Fig. 1. Stream processing model.

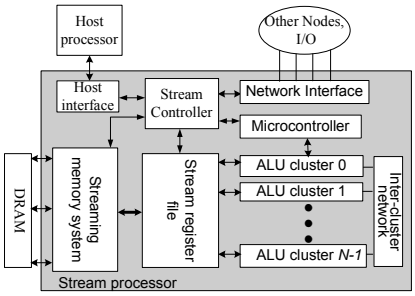


Fig. 2. Simplified diagram of FT64.

Stream Architecture

The stream programming model has been shown to efficiently map to stream architectures, such as Imagine [1], FT-64 [4] and Merrimac [10]. The architecture for a stream processor is illustrated in Fig.2. Identical clusters of ALUs operate in parallel on sequential records of streams, in a SIMD fashion. That is, each cluster is executing the same instruction at the same time. Instruction-Level Parallelism (ILP) in kernels is exploited by the use of Very Long Instruction Words (VLIW), with each very long instruction word encoding multiple instructions that are executed in parallel across the cluster's ALUs. There is a three-level memory hierarchy. Only data in the Local Register Files (LRFs), immediately adjacent to the arithmetic units, can be used by the clusters during each kernel execution. The Stream Register File (SRF), an on-chip storage, is used to read and write streams between kernels. Off-chip memory bandwidth is used only for an application's inputs, outputs and intermediate streams that cannot fit in the SRF. The memory system can concurrently support two stream transfer requests.

Streams and Scientific Computing Applications

On the one hand, the stream programming model with its hardware implementation, the stream processor, is a good choice for scientific computing applications for several reasons. First, the use of streams exposes the parallelism found in scientific computing applications at three levels: Instruction level parallelism(ILP) in kernels, Data level parallelism (DLP) among records of streams and Task level parallelism (TLP). The ILP is exploited by the multiple ALUs in each cluster executing part of the VLIW concurrently, and the DLP is exploited by the clusters operating in a SIMD fashion, and the TLP is exploited by the multiple stream processors running one kernel or successive kernels in a pipeline. Second, the memory wall is significantly relieved in the stream processor. The three-level memory hierarchy captures the reuse in scientific computing applications well: at the top level the main memory offers space for large, infrequently accessed data; at the intermediate level, the SRF allows for the on-chip reuse of streams; finally at the local and fast level, the LRFs allow for the reuse of temporary values used by kernel computations. Furthermore, because the kernels in the stream programming model are restricted to operate only on local data, the kernel execution is both fast and efficient. In addition, the high latency of memory access is well hidden in the stream processor by executing kernel computations and stream transfers, without any dependences, concurrently. Simple and explicit stream access patterns can allow for greater concurrency.

On the other hand, the stream programming model is quite new and requires a lot of manual intervention to reuse data, to enhance the parallelism and to organize the data. Therefore, mapping scientific computing applications to the stream processor and achieving good performance are quite difficult.First, the data reuse in scientific computing applications is much more complicated and irregular than that in other stream applications. This brings new challenges to programmers to preserve the reuse of data among streams and the reuse among records within streams. Second, the more complex control-flow of scientific computing applications makes it difficult to improve the parallelism in their stream applications. Third, the stream organization effects the exploitation of the architecture's feature directly. The stronger data dependence and more complex data traces of scientific computing applications make it difficult to organize data as streams efficiently. Finally, compared with other stream applications, the scientific computing applications have larger data sizes.

3. Mapping Scientific Computing Applications to the Stream Processor

The implementation of scientific computing stream programs can be thought of as a code transformation on programs that consist of a series of loops that process arrays of records. The access pattern of each loop with respect to each array is extracted into one or more streams, and the computations performed by each loop are encapsulated inside a kernel. The remaining code composes the stream level program. However, implementing more efficient stream programs requires analyzing the dataflow through and from the desired algorithm. In reality, most applications need to be restructured to make efficient use of the stream programming model.

This section describes the general implementation steps of generating stream programs and the transformations of FORTRAN code in each step.

Selection of Code Segment to be Mapped

First of all, the parts of a program to be executed on the stream processor should be selected. Generally, this will be the code with intensive computations. Many scientific computing applications are characterized by the well-known 80%-20% rule: 80% of the CPU time is spent in 20% of the code, i.e. the DO-loops. Therefore, we choose these intensive computations to be executed on the stream processor.

Stream Level

The stream level program defines the high-level control- and data-flow between kernels. When implementing a stream level program, we should first know the data access patterns and data dependence of the selected code. Then, with the objective to improve the ratio of computations to memory transfers and the parallelism across clusters, we restructure the code by the way of data dependence analysis and loop transformations. The transformations used to restructure the code in Fig.3(a) are given below.

- Loop fusion. *Loop fusion* involves joining loops that iterate over the same iteration space. This method can expose expressions within the loops that can be shared. For the stream processor architecture, this transformation can allow for the reuse of the LRFs amongst loops and reduce the number stream operations necessary to carry out a computation. Loop1 and Loop2 in Fig.3(a) can be joined together because they are dependence free loops with the same iteration space, while Loop2 and Loop3 cannot. The code transformed by loop fusion is shown in Fig.3(b). Generally, as many loops as possible should be fused. The opportunities to fuse loops are increased by other loops optimizations such as loop reversal and loop peeling [15].
- Loop replication. If the body of a block contains loops which can be fused with loops near the conditional statement, such as Loop3 and Loop12 in Fig.3(b), we should replicate the loops into the conditional block, as shown in Fig.3(c).
- Iterative loop optimizations. The methods highlighted above are used repeatedly to make a loop contain as many as possible computations. The final code shown in Fig.3(d), where we have reduced the number of loops to two and all conditions are outside of the outermost loops.

Stream Organization

The organization of arrays with respect to access mode into streams decides the number of stream memory transfers, the amount of inter-LRF communication and the utility of the SRF; it is key in ensuring good program performance. First, through trace analysis, arrays with the same data trace, such as $CC(LQ)$, $TT(LQ,1)$, $TT(LQ,2)$, $TT(LQ,3)$ and $TT(LQ,4)$, should be organized as a stream whose record is made up of all elements referred to by the same index variable's value. Therefore, the data used in a single kernel iteration, instead of being distributed, is combined into records at consecutive addresses in off-chip memory. This reduces memory access overheads.

If the working set of the selected loop is beyond the SRF capacity, strip-mining should be applied. Loop strip-mining converts a single loop into a nested loop where the inner loop iterates over a subset of the iteration space. The size of the strip in the inner loop is such that the SRF capacity is not exceeded. The strip size should also be determined with the consideration for reserving SRF space for loop-carried stream reuse.

A stream with both constant start and end bounds is called a *constant-bound stream*, such as the stream reference $a(0,64)$ and the basic stream s whose start bound is 0 and end bound is the stream length. Correspondingly, a stream with variable start or end bound is called a *variable-bound stream*. When possible, references to variable-bound streams should be avoided by replacing such references to constant-bound streams. Avoiding referring to variable-bound streams helps the data-flow analysis of the compiler analyze dependencies and increase reuse.

Additionally, good stream organization can reduce inter-LRF communication and improve the utility of the SRF, which will be described in Sect. 4.2 and Sect. 4.3, respectively.

Kernel Level

Intensive computations on streams are organized into kernels. For the code in Fig.3(d), the computations in Loop123 and Loop12' are written into kernels respectively.

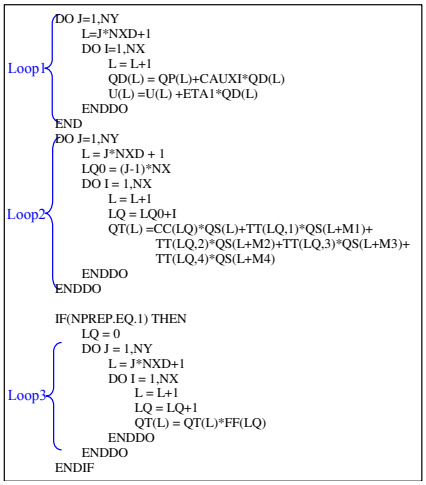
We have mapped scientific computing applications from FORTRAN to the stream processor. Table 1 summarizes the transformations used in each benchmark that will be part of our presented results.

	QMR.	MVM	Laplace	Swim	MG	FFT	LUD	GEMM
Loop fusion	√		√	√	√		√	
Loop replication	√					√	√	
Strip-mining	√	√	√	√	√	√	√	√

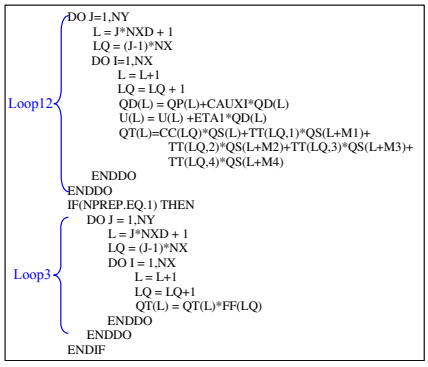
Table 1. Transformations used in each benchmark.

4. Optimization for the Stream Programming Model

StreamC/kernelC provides two methods, *unroll* and *pipeline*, to optimize kernels, and two methods, *doUnroll* and *doSoftwarePipeline*, to optimize stream level programs [16]. These optimizations are inherited from the scalar and vector programming models and have been widely used to improve the performance of stream programs [9], including ours. However, we do need some further optimizations specialized for the stream programming model. This section proposes several other transformations aimed at optimizing the following aspects: utilizing high bandwidth from the SRF to the LRFs and vice versa, reducing inter-LRF communication, exposing stream reuse and avoiding resource conflict during prefetching.

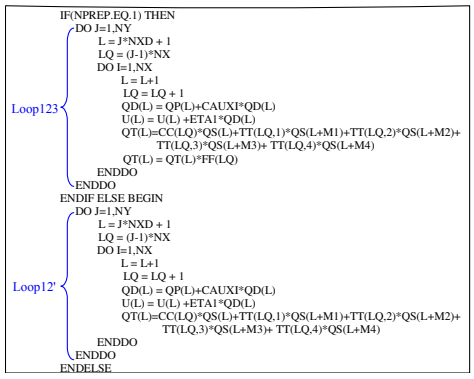


(a) Example code.



(b) Code after loop fusion

(c) Code after loop replication



(d) Final code

Fig. 3. Example code 1 - use of loop transformations to improve code layout for the stream processor.

4.1. Stream Splitting

The total number of streams that may be referred to within a kernel is limited by hardware design. The Imagine stream processor has a limit of 8 streams [2], [17]. For a kernel that accesses less than 8 streams, *stream splitting* involves dividing each stream into two or more even parts and updating the kernel to process double or more streams at a time. Each part of the stream acts as an new input or output stream of the transformed kernel. As work on the streams must be inherently parallel, stream splitting does not bring about any data dependence violations. Note that the number of streams accessed by the transformed kernel must obey the limit.

```
for(i=0; i<16; i++){
   a0 = a((i*strip),(i*strip+strip));
   b0 = b((i*strip),(i*strip+strip));
   product(a0, b0, product);
}
```

```
for(i=0; i<16; i++){
   a0_upper = a((i*strip),(i*strip+strip/2));
   b0_upper = b((i*strip),(i*strip+strip/2))
   a0_lower = a((i*strip+strip/2),(i*strip+strip));
   b0_lower = b((i*strip+strip/2),(i*strip+strip));
   product_new(a0_upper, b0_upper, a0_lower,
  b0_lower, product);
}
```

Fig. 4. Example of stream splitting

```
const int N = 256;
stream<float> a((N+2)*(N+2)),b(N*N);
stream<float> a0, a1, a2, a3,a4,b0;
float B[N][N];
for(int i = 0; i<N; i++){
   a0 = a(i * N, (i + 1) * N);
   a1 = a((i + 1) * N, (i + 2) * N);
   a2 = a((i + 2) * N, (i + 3) * N);
   a3 = a(i * N + 1, (i + 1) * N + 1);
   a4 = a(i * N + 2, (i + 1) * N + 2);
   b0 = b(i * N, (i + 1) * N);
   example(a0,a1,a2,a3,a4,b0);
}
streamSave(b, B);
```

Fig. 5. Example code 2

The code in the top of Fig.4 computes the product of streams a and b, with the loop having already transformed by strip-mining. Because the kernel product has only two stream arguments, the transformation can be applied, with the transformed code shown in the bottom of Fig.4. The original input streams, a_0 and b_0, are split into two even streams, a_0_upper, b_0_upper, a_0_lower and b_0_lower, respectively. The kernel $product$ is updated into the kernel $product\_new$ which calculates $product\_upper$, the product of a_0_upper and b_0_upper, and $product\_lower$, the product of a_0_lower and b_0_lower, and $product$, the sum of $product\_upper$ and $product\_lower$. As the code is still below the 8 stream limit, stream splitting can be applied to the code again (although not shown here).

Stream splitting improves the utilization of the bandwidth

from the SRF to LRFs and vice versa by allowing more concurrent stream transfers. Furthermore, stream splitting has the advantages brought by unrolling a kernel loop, such as less loop overhead, increased opportunity to perform local optimizations and thereby potentially increased ILP. Stream splitting can be seen as a form of manually strip-mining the kernel loop and fusing the mined loops.

4.2. Stream Transposition

Stream transposition involves redistributing records of a stream among the SRF lanes to make consecutive records on the same lane. The SRF is banked into lanes such that each cluster contains a single lane. Records of a stream are interleaved among lanes. A cluster gets records on other lanes only by inter-lane communication. Fig.6 shows the distribution of the stream a_0 in Fig.5 among the lanes, with neighboring records on neighboring lanes.

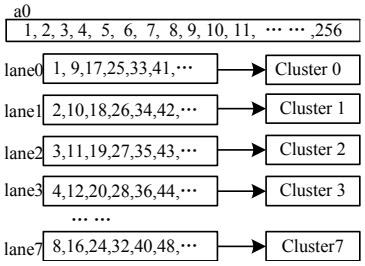


Fig. 6. Distribution of a_0 among lanes.

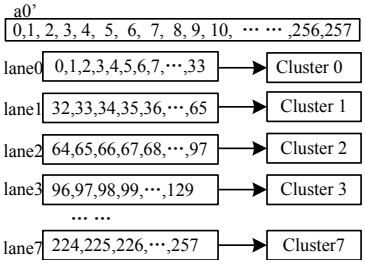


Fig. 7. Distribution of a'_0 among lanes after being transposed.

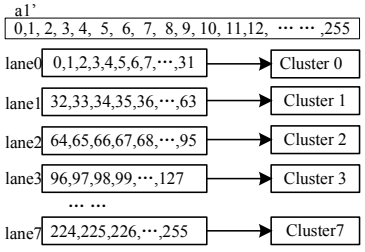


Fig. 8. Distribution of a'_1 among lanes after being transposed.

In the original FORTRAN program of the code shown in Fig.5, three sequential elements of array a are involved in each iteration of the kernel loop, i.e. $a(i,j)$, $a(i+1,j)$ and $a(i+2,j)$. When mapping such loops, we have two choices:

- Organize data as different streams to start and to end at different offsets into the original data arrays, with the stride skipping over the unneeded elements. The stream level program in Fig.5 is generated this way. The data covered by the three above array references in the loop are organized into the streams a_0, a_3 and a_4 in referred order. Although the data in the records of every stream are almost the same, just displaced, all the streams must be loaded from off-chip memory. The increase in memory transfers consequently makes memory accesses the performance bottleneck of this approach.
- Organizing data covered by the three array references in the loop as a single stream. In this way, the stream references a_0, a_3 and a_4 in Fig.5 are merged into one stream $a'_0 = a(i \times N, (i+1) \times N + 2)$. However, during the kernel *example* execution, $cluster_i$ must communicate with $cluster_{i+1}$ and $cluster_{i-1}$ to gather neighboring records. Inter-lane communication causes clusters to stall while waiting for the data collection and as such becomes the performance bottleneck of this approach.

```
const int N = 256;
stream<float> a((N+2)*(N+2)),b(N*N);
stream<float> a0', a1', a2', b0';
float B[N][N];
for(int i = 0; i<N; i++){
   a0' = a(i *N, (i+1)*N+2, N/Ncluster, N/Ncluster+2);
   a1' = a((i+1)*N, (i+2)*N, N/Ncluster, N/Ncluster);
   a2' = a((i+2)*N, (i+3)*N, N/Ncluster, N/Ncluster);
   b0' = b(i*N, (i+1)*N, N/Ncluster, N/Ncluster);
   example'(a0',a1',a2',b0');
}
streamSave(b, B);
```

Fig. 9. Code optimized by stream transposition.

Stream transposition reorganizes the data distribution, with adjacent records distributed on the same lane. Thus, the optimized stream program has the same number of memory transfers with the second choice, but does not require any inter-LRF communication. The steps of the stream transposition for Fig.5 are given below.

Step A. Organize all records covered by three array references into a stream named a'_0, with the same order as the array a.

Step B. Set the stride of a'_0 be $Length_{stream}/N_{cluster}$ and the record length be $Length_{stream}/N_{cluster} + 2$, with $N_{cluster}$ representing the number of clusters and $Length_{stream}$ representing the stream length. This means the original records from $i \times Length_{stream}/N_{cluster}$ to $i \times Length_{stream}/N_{cluster} + Length_{stream}/N_{cluster} + 2$ become the *ith* record of the new derived stream, distributed on $cluster_i$. The distribution of a'_0 among lanes is shown in Fig.7. Data distribution is transposed as shown, with neighboring records distributed on the same lane, such that $cluster_i$ gets neighboring records from the lane of itself without any inter-lane communication.

Step C. Divide all other stream references into $N_{cluster}$ parts by setting the stride be $Length_{stream}/N_{cluster}$ and the record length be $Length_{stream}/N_{cluster}$. The distribution of a'_1 is shown in Fig.8. As shown, data distribution of other streams is also transposed.

Step D. Update original kernel to process the records in the corresponding new order.

After the optimization, the final stream program, shown in Fig.9, has few inter-cluster communication and less memory transfers.

4.3. Stream Reuse

Reuse among streams occurs in the SRF when a stream reference in a stream level loop accesses the same values in different iterations, or different stream references access the same values. For loop-independent stream reuse, i.e. different stream references accessing the same values in the same iteration, the stream compiler utilizes it as producer-consumer locality. But for loop-carried stream reuse, the utilization is no longer straightforward.

```
for(int i = 0; i<N; i++){
    a0'= a(i *N, (i+1)*N+2, N/Ncluster, N/Ncluster+2);
    a1"= a((i+1)*N, (i+2)*N+2, N/Ncluster, N/Ncluster+2);
    a2"= a((i+2)*N, (i+3)*N+2, N/Ncluster, N/Ncluster+2);
    b0'= b(i*N, (i+1)*N, N/Ncluster, N/Ncluster);
    example"(a0',a1",a2",b0');
}
streamSave(b, B);
```

Fig. 10. Changed loop.

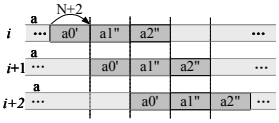


Fig. 11. Relationship among the access locations of the stream references a'_0, a_1" and a_2".

The stream length and record length of the streams a'_1 and a'_2 in Fig.9 are changed to match those of the stream a'_0, with the changed streams named a_1" and a_2". Correspondingly, the kernel is updated to process the correct records. Fig.10 shows the loop in the changed code. For the stream references a'_0, a_1" and a_2", the relationship among their accessing locations relative to the start of their basic stream is described in Fig.11. It is shown that the stream reference a_2" in iteration i, the stream reference a_1" in iteration $i+1$ and the stream reference a'_0 in iteration $i+2$ access the same locations. As the values of the basic stream a are unchanged, the stream reference a_1" does not require accessing off-chip memory but accesses the SRF to achieve the values that are used by the stream reference a_2" in a previous iteration. Similarly, a'_0 accesses the SRF to achieve the values that are used by the stream reference a_2" in the two previous iterations.

The transformation of a stream level program, to make the stream compiler capture loop-carried stream reuse, is discussed below.

Step A. Replace variable-bound stream references with constant-bound stream references. Because the compiler can only identify the reuse supplied by constant-bound stream references, all variable-bound stream references with constant length should be transformed to constant-bound stream references by copying variable-bound streams to constant-bound streams and replacing references involved by references to the constant-bound streams. Fig.12(a) shows the transformed code; the function $streamCopy(s,t)$ copies records of s to t. An SRF-to-memory copy generates a save of s to memory; a memory-to-SRF copy generates a load of s to the SRF; an SRF-to-SRF copy generates a save of s to memory and a load of s to the SRF buffer that holds t.

Step B. When the stream s_1 is reused as s_2 in the next iteration, remove the load of s_2, inserting the function

```
for(int i = 0; i<N; i++){
    stream<float> a00[N],a01[N],a02[N],b00[N];
    … //definitions of a0', a1", a2" and b0'
    streamCopy(a0', a00);
    streamCopy(a1", a01);
    streamCopy(a2", a02);
    example"(a00,a01,a02,b00);
    streamCopy(b00, b0')
}
streamSave(b, B);
```

(a) Transformed code after step A.

```
streamCopy(a(0*N, 1*N+2, N/Ncluster, N/Ncluster+2), a00);
streamCopy(a(1*N, 2*N+2, N/Ncluster, N/Ncluster+2), a01);
for(int i = 0; i<N; i++){
    stream<float> a00[N],a01[N],a02[N],b00[N];
    … //definitions of a0', a1", a2" and b0'
    streamCopy(a2", a02);
    example"(a00,a01,a02,b00);
    streamCopy(b00, b0')
    streamCopy(a01, a00);
    streamCopy(a02, a01);
}
streamSave(b, B);
```

(b) Transformed code after step B.

```
streamCopy(a(0*N, 1*N+2, N/Ncluster, N/Ncluster+2), a00);
streamCopy(a(1*N, 2*N+2, N/Ncluster, N/Ncluster+2), a01);
for(int i = 0; i<N-N%3; i++){
    stream<float> a00[N],a01[N],a02[N],b00[N];
    … //definitions of a0', a1", a2" and b0'
    streamCopy(a2", a02);
    example"(a00,a01,a02,b00);
    streamCopy(b00, b0')
    i = i + 1;
    … //definitions of a0', a1", a2" and b0'
    streamCopy(a2", a00);
    example"(a01,a02,a00,b00);
    streamCopy(b00, b0')
    i = i + 1;
    … //definitions of a0', a1", a2" and b0'
    streamCopy(a2", a01);
    example"(a02,a00,a01,b00);
    streamCopy(b00, b0')
}
for(int i = N-N%3; i < N; i++){
    … //definitions of a0', a1", a2" and b0'
    streamCopy(a0',a00);    streamCopy(a1",a01);
    streamCopy(a2", a02);
    example"(a00,a01,a02,b00);
    streamCopy(b00, b0')
}
streamSave(b, B);
```

(c) Final code.

Fig. 12. Transformation of stream reuse.

$streamCopy(s_1, s_2)$ at the end of the loop body to reuse s_1 as s_2 in the next iteration, and modify the loads of the data referred to by s2 in the first iteration just before the loop body. The code in Fig.12(a) is changed to that in Fig.12(b) by inserting the function $streamCopy$ (a_{01}, a_{00}) to reuse a_{01} as a_{00} and the function $streamCopy$ (a_{02}, a_{01}) to reuse a_{02} as a_{01}, removing the load of a_{00} and a_{01}, and adding two stream loads of a'_0 and a_1" just before the loop body.

Step C. Eliminate stream copies. An SRF-to-SRF stream copy generates a stream load and a stream store, so they must be eliminated. Since these copies implement a permutation of values in the SRF, we can eliminate the need for copies by unrolling to the cycle length of the permutation, with the stream references replaced by the right streams. The final code is shown in Fig.12(c). The first loop does not require any SRF-to-SRF copies and the number of its iterations is a multiple of 3. The second loop, called *epilogue loop*, runs iterations left out of the unrolled loop. It is straightforward to prove that the final code calculates the desired answer.

The stream compiler can capture reuse in transformed stream programs, thus efficiently reducing off-chip memory transfers.

FORTRAN programs, with records in the same row but in different columns (such as $a(i,j)$ and $a(i,j+1)$) involved in the loop body to be transformed to a kernel, have loop-carried stream reuse. This transformation allows the compiler to reuse streams and thereby reduce off-chip memory transfers.

4.4. Ensuring Prefetching

The concurrency of kernel computations and memory transfers is the most important way of hiding memory access latency in a stream processor. Only stream operations without any dependence can be executed concurrently. In this way, during the execution of a kernel, the input streams of next kernel can be prefetched if the stream transfers do not have any dependenceies that interfere with the kernel execution. Explicit and simple memory access patterns, and minimal data- and resource- dependences, produce many opportunities for prefetching, and make it a key optimization for stream processors. To ensure the prefetching of input streams, we should avoid conflicts in the SRF between a kernel execution and the stream loads of the next kernel's inputs.

Fig.13(a) shows the time sequence graph of the first two iterations' execution of the code in the bottom of Fig.4. The *Clusters* column shows which kernels were running at what time, and the *MEM1* and *MEM2* columns show the streams passing between the stream processor and its off-chip memory. With four SRF buffers, named $buffer_0$, $buffer_1$, $buffer_2$ and $buffer_3$, allocated to the four input streams a_0_upper, b_0_upper, a_1_lower and b_1_lower, when the kernel $product\_new$ of the first iteration is running, these four buffers are used by the kernel. At the same time, the prefetching of a_0_upper and b_0_upper for the kernel of the second iteration requires $buffer_0$ and $buffer_1$. As a result, the prefetching fails and there is no overlap between kernel execution and stream transfers as shown in Fig.13(a).

When the prefetching fails due to an SRF conflict, our optimization declares new SRF space to hold the data to be prefetched, hence the name "ensuring prefetching". For code at the bottom of Fig.4, the ensuring prefetching transformation unrolls the loop two times. The stream compiler will allocate different stream buffers for the different unrolled kernels' work sets, thus avoiding an SRF conflict. The time sequence graph of the first two kernels' execution is shown in Fig.13(b). For more complex code in Fig.12(c), ensuring prefetching declares an extra constant-bound stream to hold the stream to be prefetched and replaces all stream references involved by references to this stream. In addition, the approach of software pipelining [16] can be used in the transformed code to overlap the last kernel execution with the prefetching of the streams for the first kernel in the next iteration.

5. Evaluation

In order to evaluate the performance of scientific computing applications on the stream processor, we perform tests on eight typical scientific application kernels as specified in

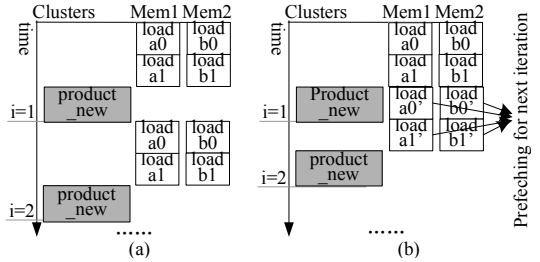


Fig. 13. Execution time sequence graph without (a) and with (b) the ensuring prefetching optimization.

Table 2, each having different performance characteristics. QMRCGSTAB, denoted as QMR., is a solver of large non-symmetric linear systems in subspace iteration method [18], and MVM calculates multiplication of two band matrices. The row *max(computation density)* is used to evaluate the relationship between computations and memory transfers. It is quantified with the number of the computations per word transfer, assuming that each element is loaded from off-chip memory at most once.

Number of clusters	8
Operating frequency	1GHz
Capacity of the LRFs	9.6KB
Capacity of the SRF	128KB
Bandwidth of the LRFs	1088GB/s
Bandwidth of the SRF	64GB/s
Bandwidth of chip-off DRAM	4GB/s

Table 3. Baseline parameter of Isim.

Operating frequency	1.6GHz
Capacity of Level 1 Data/Instruction Cache	16KB
Capacity of Level 2 Data/Instruction Cache	256KB
Capacity of Level 3 Data/Instruction Cache	6MB
Bandwidth of chip-off DRAM	6.4GB/s

Table 4. Properties of the Itanium 2 processor.

In our experiments, we use Isim, a cycle-accurate simulator for the Imagine stream processor [2], [17], to get the performance of the stream programs. The baseline configuration of the simulated stream processor and its memory system is detailed in Table 3, and is used for all experiments unless noted otherwise. For comparison, the original FORTRAN programs are compiled by Intel's compiler *ifort* with -O3 optimization, and then executed on a single-core Itanium 2 server, whose properties is shown in Table 4. If the data size of the program is small, we eliminate the extra overheads (such as system calls) by means of executing them multiple times and getting the average time as the final result. .

5.1. Overall Performance

The performance of scientific stream programs with all available optimizations is presented first to evaluate the stream processor's applicability to scientific computing. Fig.14 shows the speedup yielded by Isim over the Itanium 2. All applications get better performance on Isim, which indicates stream processors can be successfully applied to scientific computation. FFT enjoys the highest speedup, due to its data access pattern in the butterfly and bit-reverse transformations

	QMR.	MVM	Laplace	Swim	MG	FFT	LUD	GEMM
Source	–	–	NCSA	Spec2000	NPB	HPCC	BLAS	BLAS
Prob. Size	800×800	832×832	1024×1024	512×512	$128 \times 128 \times 128$	4096	128×128	512×512
max(comp. density)	1.85	1.5	2.5	3.67	3.67	44	56	341

Table 2. Specifications of eight scientific kernel benchmarks.

that are specially supported by the stream processor. Other computation-intensive applications, i.e. LUD, and GEMM, get larger speedups, due to the increased number of ALUs in the stream processor. Memory-intensive applications, i.e. QMRCGSTAB, MVM, Laplace, Swim and MG, also get better performance on Isim, mainly due to the stream reuse in the SRF and the overlap of kernel execution with memory transfers.

Fig.15 presents the applications' performance on Isim in GFLOPS. Memory-intensive programs, i.e. QMR., MVM, Laplace, Swim and MG, gain 5.3%∼13.0% of the Isim peak performance, and compute-intensive programs, i.e. FFT, LUD and GEMM, gain 10.4%∼44.3% of the Isim peak performance.

One key to achieving high performance on the stream processor is making kernel execution and memory transfers occur concurrently. Fig.16 demonstrates the distribution of kernel execution time and memory access time. The difference between their sum and 1 equals the overlapping time of the kernel execution with memory transfers. Memory-intensive programs, i.e. QMR., MVM, Laplace, Swim and MG, attain good concurrency. In particular, the kernel execution time and memory access time of Laplace are almost evenly distributed, indicating perfect concurrency. But computation-intensive programs, i.e. FFT, LUD and GEMM, yield little concurrency. This is because there are few stream transfers in the kernel execution, except for the initial input and a final output.

Fig.16 also shows the time taken for applications to execute kernels and to access memory, respectively. It still takes memory-intensive applications much time to access memory. However, due to the reuse of streams and the concurrent memory accesses, memory access delays become less. In particular, Laplace is now bounded by both computing resources and memory access performance. Surprisingly, computation-intensive applications are now somewhat subject to memory access performance. An abundance of ALUs in the stream processor execute the computations quickly and memory access becomes a performance bottle neck. The initial input and final output of FFT, for example, take so much time that the performance of FFT is bounded by the time to access memory.

A second key to performance is to reuse the streams in the SRF. Fig.17 shows the computation density with the computations per word transfer, indicating the degree of stream reuse. In fact, except for LUD and GEMM, all applications nearly achieve the max computation density(see Table 2). For example, Laplace which calculates the difference of an element with its 4 adjacent elements in 2-dimensions, could get at most 2.5 computations per word transfer and at least 0.83 computation per word transfer; its stream program achieves the upper bound. As the reuse in LUD and GEMM is exploited over strips, some reuse does not exist in the innermost loop in these applications.

Scientific applications also map well to the bandwidth hierarchy. Achieved bandwidths for the benchmarks are shown in Fig.18. The difference between the data bandwidths required at two adjacent levels of the hierarchy is at least an order of magnitude. This indicates that the bandwidth hierarchy effectively captures the locality exhibited by these applications.

5.2. Effect of Optimizations

In this section we discuss the effect of the optimizations we present on the application performance. We quantify this goal by comparing application results without corresponding optimization to results presented in Sect. 5.1. Table 5 summarizes the optimizations applicable to each benchmark.

	QMR.	MVM	Laplace	Swim	MG	FFT	LUD	GEMM
Stream Splitting	√		√			√		
Stream transposition	√	√	√					
Stream Reuse	√	√	√	√	√	√	√	√
Ensuring Prefetching			√					√

Table 5. Optimizations available for each benchmark.

Stream Splitting

We first demonstrate the effectiveness and importance of the stream splitting optimization presented in Sect. 4.1. As this optimization improves the kernel locality and ILP, and accelerates stream access, we quantify this optimization by kernel execution time and stream access time with and without this optimization. Fig.19(a) and Fig.19(b) show the kernel execution time and stream access time. These results show that stream splitting does improve the kernel execution and stream access although it does not reduce any operations or memory transfers. Fig.19(c) shows the program performance speedup, which demonstrates the effectiveness of the optimization. Stream splitting has little impact on other aspects of the stream processor, such as the transfers of three hierarchies. Therefore, these parameters are not shown here.

Stream transposition

We now demonstrate the effectiveness of the stream transposition transformation that reorganizes streams to reduce off-chip memory transfers. Fig.20(a) clearly illustrates the ability of the stream transposition transformation to reduce memory transfers. Fig.20(b) shows the effectiveness of the optimization on the computation density, indicating the ability to capture the stream reuse. Fig.20(c) demonstrates marginal speed-up, due to the reduction of memory transfers.

Stream Reuse

As a stream processor reduces memory transfers only by capturing the reuse among streams in the SRF, the stream reuse transformation is important. We evaluate the impact of the stream reuse transformation on program performance. All the selected benchmarks benefit from this optimization.

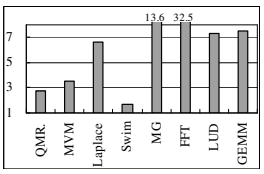


Fig. 14. Speedup yielded by Isim over Itanium 2.

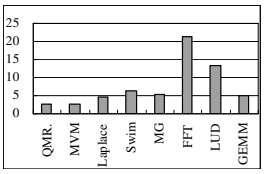


Fig. 15. Performance gain of the applications on Isim (GFLOPS).

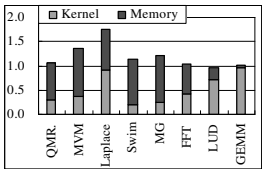


Fig. 16. Distribution of kernel execution time and memory access time (normalized to the total execution time).

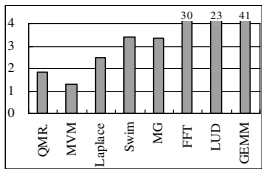


Fig. 17. Computations per word transfer.

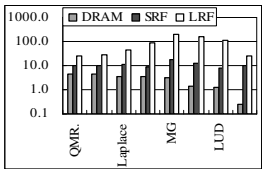
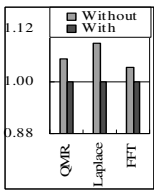
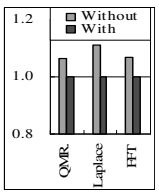


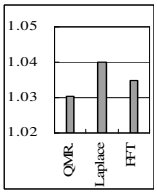
Fig. 18. Achieved bandwidths for the benchmarks(GB/s).



(a) Kernel execution time (normalized to that with stream splitting)

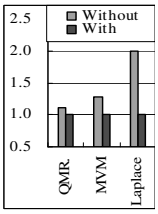


(b) Stream access time (normalized to that with stream splitting)

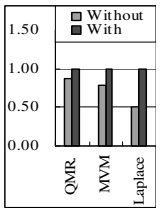


(c) Speedup from stream splitting

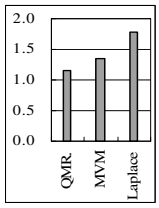
Fig. 19. Effectiveness of the stream splitting optimization.



(a) Memory transfers (normalized to those with stream transposition).



(b) Com./word (normalized to those with stream transposition).



(c) Speedup from stream transposition.

Fig. 20. Effectiveness of the stream transposition optimization.

Fig.21(a) shows the reduction of memory transfers. For the computation-intensive applications FFT and LUD, each input stream of the kernel is the whole or part of an output stream of the kernel in the previous iteration. Without the optimization, the stream compiler cannot identify the reuse, all output streams are written back to off-chip memory and each kernel must wait until its input streams are loaded from off-chip memory. Stream reuse removes the appearance of variable-bound streams, thus making the stream compiler able to identify reuse. Fig.21(b) depicts the effectiveness of the optimization on the computation density, indicating the reuse among streams is exploited well. Fig.21(c) demonstrates the speedup attained with this optimization. The application Swim yields the lowest speedup. This is because the reduced memory transfers were already overlapped with kernel execution.

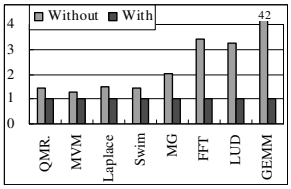
Ensuring Prefetching

We evaluate the effectiveness of the ensuring prefetching transformation that improves the concurrency of kernel execution and memory transfers by exchanging computing time for SRF consumption. Fig.22 shows the speedup attained with ensuring prefetching, thus demonstrating its effectiveness in exposing concurrency. In spite of performance improvements, the optimization has little effect on kernel execution or memory transfers.

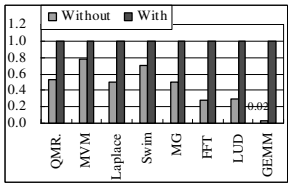
6. Discussion

As the existing programming language implementations for the stream programming model expose low-level features of the architecture, programmers have to manually adjust stream programs to make them exploit the stream architecture's characteristics well. The implementation in the compiler of the before-mentioned optimizations will relieve programmers of this burden greatly and this is the focus of our research. We consider here the implementation of the optimizations in the compiler.

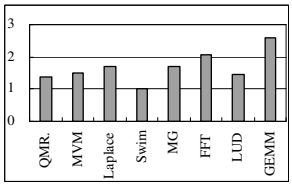
- Stream Splitting can be implemented in the compiler as follows. For a loop to be transformed to a kernel, if the number of streams that will be the kernel's arguments is less than 8, the iteration space of the loop is divided into two or more even parts, each part becoming a new loop. Then all generated loops are merged into one large loop and the new loop is mapped to a corresponding stream program. As the dependencies of the original program are not violated prior to mapping to the stream program, the transformation is safe; however, pipelining may be required to handle loop carried dependencies. Note that the final generated kernel has at most 8 stream arguments due to the number of clusters within the Imagine processor.
- The stream transposition optimization can be applied automatically as follows. When FORTRAN programs are being mapped, the compiler identifies when adjacent records of an array are referred to in the loop to be transformed. Then the compiler reorganizes the kernel's streams and updates the kernel as described in Sect. 4.2.
- Automatically capturing the reuse of streams on the SRF can be implemented as follows. The compiler identifies the stream reuse first, with the aid of data dependence analysis. Then, the code is transformed as described in Sect. 4.3 to capture the reuse. The transformation done by this optimization has some similarity to the classic *scalar replacement* [19] transformation.
- For the ensuring prefetching transformation, the compiler evaluates the conditions when the SRF conflict disturbs the prefetching, and then declares new SRF buffers for the streams to be prefetched into.



(a) Memory transfers (normalized to those with stream transposition).



(b) Computations per word transfer(normalized to those with stream transposition).



(c) Speedup from stream reuse transformation.

Fig. 21. Effectiveness of the stream reuse optimization.

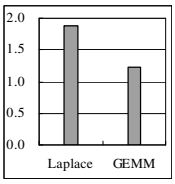


Fig. 22. Speedup yielded with the ensuring prefetch optimization.

7. Conclusion

This paper has presented an experimental evaluation of scientific computing applications on a stream processor. Eight different scientific applications were used for the experiments, each having different performance characteristics. These tests show that both memory-intensive applications and computation-intensive applications achieve good performance on the stream processor. The stream programs yield a speedup from 1.67 to 32.5 over corresponding FORTRAN programs run on an Itanium processor. The results indicate the applicability of scientific computing applications to the stream processor.

Several optimizations are presented to exploit aspects of the stream processor: exploiting the bandwidth, reducing inter-LRF communication, minimizing memory transfers, and exposing the concurrency between kernel execution and memory transfers. These optimizations deliver significant performance improvements. In addition, we discuss how to optimize applications automatically in a compiler with our methods.

Acknowledgment

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

References

[1] S. Rixner, *Stream Processor Architecture*. Kluwer Academic Publishers, 2001.

[2] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B. Khailany, "The Imagine Stream Processor," *Proceedings 2002 IEEE International Conference on Computer Design*, pp. 282–288, 2002.

[3] M. Taylor, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal *et al.*, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[4] X. Yang, X. Yan, Z. Xing, Y. Deng, J. Jiang, and Y. Zhang, "A 64-bit stream processor architecture for scientific applications," *Proceedings of ISCA*, vol. 35, pp. 210–219, 2007.

[5] M. Gordon, D. Maze, S. Amarasinghe, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong *et al.*, "A stream compiler for communication-exposed architectures," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 291–303, 2002.

[6] J. Owens, S. Rixner, U. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. Dally, "Media processing applications on the Imagine stream processor," *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 295–302, 2002.

[7] J. Owens, "COMPUTER GRAPHICS ON A STREAM ARCHITECTURE," Ph.D. dissertation, STANFORD UNIVERSITY, 2002.

[8] M. Fatica, A. Jameson, and J. Alonso, "STREAMFLO: an Euler solver for streaming architectures," *submitted to AIAA Conference*.

[9] M. Erez, J. Ahn, A. Garg, W. Dally, and E. Darve, "Analysis and Performance Results of a Molecular Modeling Application on Merrimac," *SC04*, 2004.

[10] W. Dally, T. Knight, U. Kapasi, F. Labonte, A. Das, P. Hanrahan, J. Ahn, J. Gummaraju, M. Erez, N. Jayasena *et al.*, "Merrimac: Supercomputing with Streams," *Proceedings of the ACM/IEEE SC2003 Conference-Volume 00*, 2003.

[11] U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. Ahn, P. Mattson, and J. Owens, "Programmable Stream Processors," *Computer*, vol. 36, no. 8, pp. 54–62, 2003.

[12] P. Mattson, "A PROGRAMMING SYSTEM FOR THE IMAGINE MEDIA PROCESSOR," Ph.D. dissertation, STANFORD UNIVERSITY, 2002.

[13] I. Buck, "Brook Spec v0. 2," 2003.

[14] K. Fatahalian, D. Horn, T. Knight, L. Leem, M. Houston, J. Park, M. Erez, M. Ren, A. Aiken, W. Dally *et al.*, "Memory—Sequoia: programming the memory hierarchy," *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[15] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, 1994.

[16] P. Mattson *et al.*, "Imagine Programming System Developers Guide."

[17] "The imagine project, stanford university, http://cva.stanford.edu/imagine/."

[18] T. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. Tong, "A quasi-minimal residual variant of the Bi-CGSTAB algorithm for nonsymmetric systems," *SIAM J. Sci. Comput*, vol. 15, no. 2, pp. 338–347, 1994.

[19] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pp. 53–65, 1990.