Int. J. Communications, Network and System Sciences, **2010, 3, 32-37**
doi:10.4236/ijcns.2010.31003 Published Online January 2010 (http://www.SciRP.org/journal/ijcns/).

Exploiting Loop-Carried Stream Reuse for Scientific Computing Applications on the Stream Processor

Weixia XU, Qiang DOU, Ying ZHANG, Gen LI, Xuejun YANG
Institute of Computer, National University of Defense Technology, Changsha, China
Email: {*xuwx, douq, zhangying, genli, xjyang*}*@nudt.edu.cn*
Received September 19, 2009; *revised October* 22, 2009; *accepted December* 1, 2009

Abstract

Compared with other stream applications, scientific stream programs are usually bound by memory accesses. Reusing streams across different iterations, i.e. loop-carried stream reuse, can effectively improve the SRF locality, thus reducing memory accesses greatly. In the paper, we first present the algorism identifying loop-carried stream reuse and that exploiting the reuse after analyzing scientific computing applications. We then perform several representative microbenchmarks and scientific stream programs with and without our optimization on Isim, a cycle-accurate stream processor simulator. Experimental results show that our algorithms can effectively exploit loop-carried stream reuse for scientific stream programs and thus greatly improve the performance of memory-bound scientific stream programs.

Keywords: Stream Reuse, Loop-Carried, Stream Processor

1. Introduction

Now conventional architecture has been not able to meet the demands of scientific computing [1,2]. In all state-of-the-art architectures, the stream processor [3], as shown in Figure 1, draws scientific researchers' attentions for its processing computation-intensive applications effectively [4–8].

Compared with other stream applications, scientific computing applications have less computation density, i.e. the ratio of computations to memory accesses for involved data, especially for memory-bound scientific applications. Therefore, memory accesses totally dominant the performance of scientific stream programs.

The stream processor has three level memory hierarchies [12], local register files (LRF) near ALUs exploiting locality in kernels, global stream register files (SRF) exploiting producer-consumer locality between kernels, and streaming memory system exploiting global locality. The bandwidth ratio between three level memory hierarchies is large. In Imagine [9,10], the ratio is 1:13:218. As a result, how to enhance the locality of the SRF and LRF and consequently how to reduce the chip-off memory traffics become key issues to improve the performance of scientific stream programs constrained by memory access. Figure 2 shows a stream flows across three level memory hierarchies during the execution of a stream program. First, the stream is loaded from chip-off memory into the SRF and distributed into corresponding buffer. Then it is loaded from the SRF to LRF to supply operands to a kernel. During the execution of the kernel, all records participating in kernel and temporary results are saved in LRF. After the kernel is finished, the records are stored back to the SRF. If there is producer-consumer locality between this kernel and its later kernel, the stream is saved in the SRF. Otherwise, it is stored back to chip-off memory.

Reusing streams across different iterations, i.e., loop-carried stream reuse, can improve the SRF locality. In the paper, we present algorisms identifying loop-carried stream reuse and exploiting the stream reuse according to the analysis of typical scientific computing applications. We give the identification algorism to decide what applications can be optimized and the steps how to utilize loop- carried stream reuse to optimize stream organization. Then we perform several representative microbenchmarks and scientific stream programs with and without our optimization on a cycle-accurate stream processor simulator, Isim. Experimental results show that our algorithm can improve scientific stream program performance efficiently.

2. Loop-Carried Stream Reuse

Loop-carried stream reuse is defined as that between neighboring loop iterations of a stream-level program, input or output streams of kernels in the first iteration

Copyright © 2010 SciRes.

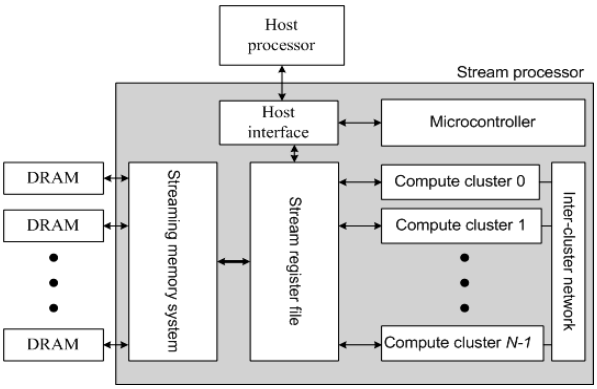

Figure 1. Block diagram of a stream processor.

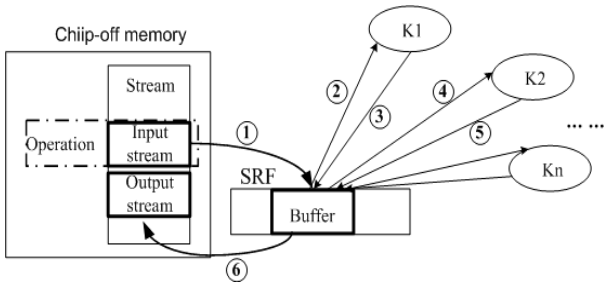

Figure 2. Stream flowing across the memory system.

```
DO J=1,NY
  DO I=1,NX
    L=J*NXD+I+1
    R(J*NXD+I+1)=CC(J*NX+I)*QP(L)+
                 TT(J*NX+I,1)*QP(L+1)+
                 TT(J*NX+I,2)*QP(L-1)+
                 TT(J*NX+I,3)*QP(L+NXD)+
                 TT(J*NX+I,4)*QP(L-NXD)
  ENDDO
ENDDO
```

(loop1: DO J=1,NY; loop2: DO I=1,NX)

Figure 3. Example code.

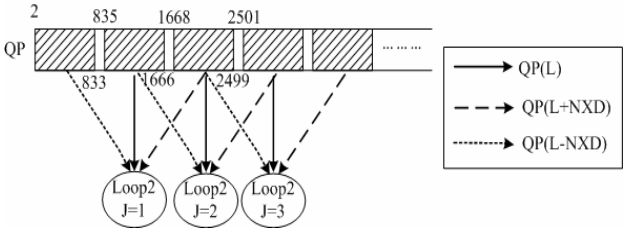

Figure 4. Data trace of array QP references.

can be used as input streams of kernels in the second iteration. If input streams are reused, we call it loop-carried input stream reuse. Otherwise, we call it loop-carried output stream reuse. The essential of stream reuse optimization is to enhance the locality of the SRF. Correspondingly, input stream reuse can enhance producer-producer locality of the SRF while output stream reuse can enhance producer-consumer locality of the SRF.

[1]In this paper, we assume the sequence of memory access is the leftmost dimension first just like as that in FORTRAN.

```
      DOI1-L1,U1
       DOI2-L2,U2
       ……
        DOID-LD,UD
I1    … -A(F1(I1,…ID),…,FM(I1,…ID)
I2    … =A(G1(I1,…ID),…,GM(I1,…ID)
        ENDDO
       ……
      ENDO
    ENDDO
```

Figure 5. A D-level nested loop.

Then we take code in Figure 3 from stream program MVM as example to depict our methods, where NXD equals to NX+2. In the paper, we let NX and NY equal to 832.

Figure 4 shows the data trace of QP(L), QP(L+NXD) and QP(L-NXD) participating in loop2 of Figure 3. QP(1668,2499) is QP(L+NXD) of loop2 when J=1, QP(L) of loop2 when J=2, and QP(L-NXD) of loop2 when J=3. So, stream QP can be reused between different iterations of loop1. If QP(1668,2499) is organized as a stream, it will be in the SRF after loop2 with J=1 finishes. Consequently, when loop2 with J=2 or J=3 running, it doesn't get stream QP(1668,2499) from chip-off memory but the SRF.

2.1. Identifying Loop-Carried Stream Reuse

Figure 5 shows a generalized perfect nest of D loops. The body of the loop nest reads elements of the m-dimensional array *A* twice. In the paper, we only consider linear subscription expressions, and the ith dimension subscription expression of the first array *A* reference is denoted as $F_i = \sum C_{i,j} * I_j + C_{i,0}$, where I_j is an index variable, $1 \le j \le D$, $C_{i,j}$ is the coefficient of I_j, $1 \le j \le D$, and $C_{i,0}$ is the remaining part of the subscript expression that does not contain any index variables. Correspondingly, the ith dimension subscription expression of the second array *A* reference is denoted as $G_i = \sum C'_{i,j} * I_j + C'_{i,0}$. If in the Pth level loop, the data trace covered by the leftmost Q dimensions[1] of one array *A* read references in the nest with I_P=i is the same to that of the other array *A* read references in the nest with I_P=i+d, where d is a const, and they are different to each other in the same nest, such as the data trace of array QP in loop2 in Figure 3, the two array *A* read references can be optimized by input stream reuse in respect of loop P and dimension Q. Then we give the algorism of ***I**dentifying **L**oop-carried **I**nput **S**tream **R**euse, ILISR*.

Algorithm ILISR. Two array references in the same loop body can be optimized by input stream reuse in respect of loop P and dimension Q if:

1)when $M = 1$, i.e. array *A* is a 1-D array, the subscript expressions of two array *A* references can be written as $F_1 = \sum C_{1,j} * I_j + C_{1,0}$ and $G_1 = \sum C'_{1,j} * I_j + C'_{1,0}$ respectively, and Q=1 now. The coefficients of F1 and G1 should satisfy following formulas:

Copyright © 2010 SciRes.

$$\forall j:((1\le j\le D)\wedge C_{1,j}=C'_{1,j}) \quad \text{(a)}$$

$$C_{1,0}=C'_{1,0}\pm d*C'_{1,P} \quad \text{(b)}$$

$$\forall j:((1\le j\le D)\wedge(C_{1,j}\ge\sum_{j=1}^{D}C_{1,j}*(U_j-L_j)) \quad \text{(c)}$$

Formulas (1a) and (1b) ensure when the loop indices of outmost *P*-1 loops are given and the loop body passes the space of innermost *D*-*P* loops, the data trace of one array *A* read reference in the nest with I_P=i is the same to that of the other array *A* read references in the nest with I_P=i+d. d is a const and we specify d=1 or 2 according to [13]. When I_1, …, I_{P-1}, are given and I_P, …, I_D vary, Formula (1c) restricts the data trace of one array *A* references in the nest with I_P=i from overlapping that of the other in the nest with I_P=i +d. This formula ensures the characteristic of stream process, i.e. data as streams is loaded into the stream processor to process and reloaded into the SRF in batches after process. For the stream processor, the cost of random access records of streams is very high.

2) when $M\neq 1$, i.e. array *A* is a multi-dimensional array, the subscript expressions of two array *A* read references should satisfy following conditions:

d) the Q*th* dimension subscript expression of one array is gotten by translating the index I_P in the dimension subscript expression of the other array by d, i.e. $G_Q=F_Q(I_P\pm d)$, and,

e) all subscript expressions of one array *A* reference are the same with those of the other except the Q*th* dimension subscript expression, i.e. $\forall i((i\neq Q)\wedge(F_i=G_i))$, and,

f) for the two array *A* references, the innermost index variable in one subscript expression will not appear in any righter dimension subscript expressions, i.e.

$$\forall i(\exists\, j(C_{i,j}\neq 0\wedge\forall j'(j'>j\wedge C_{i,j'}=0))\rightarrow$$
$$\forall i'(i'>i\wedge C_{i',j}=0))\wedge\forall i(\exists\, j(C'_{i,j}\neq$$
$$0\wedge\forall j'(j'>j\wedge C'_{i,j'}=0))\rightarrow\forall i'(i'>i\wedge C'_{i',j}=0))$$

It can be proved that data access trace of two array references decided by condition (2) satisfies condition (1), and when U_j-I_j is large enough, they are equivalent.

The algorism of ***I**dentifying **L**oop-carried **O**utput **S**tream **R**euse, ILOSR*, is similar to ILISR except that reusing stream mustn't change original data dependence. Then we give the ILOSR algorism without detailed specifications.

Algorithm ILOSR. We denote the subscript expressions of read references as F_i and those of write references as G_i. Two array references in loop body can be optimized by output stream reuse in respect of loop P and dimension *Q* if:

3) when $M=1$, the coefficients of F_1 and G_1 should satisfy following formulas:

$$\forall j:((1\le j\le D)\wedge C_{1,j}=C'_{1,j}) \quad \text{(g)}$$

$$C_{1,0}=C'_{1,0}+d*C'_{1,P} \quad \text{(h)}$$

$$\forall j:((1\le j\le D)\wedge(C_{1,j}\ge\sum_{i=1}^{D}C_{1,j}*(U_j-L_j)) \quad \text{(i)}$$

4) when $M\neq 1$, the subscript expressions of two array *A* read references should satisfy following formulas:

$$G_Q=F_Q(I_P+d) \quad \text{(j)}$$

$$\forall i((i\neq Q)\wedge(F_i=G_i)) \quad \text{(k)}$$

$$\forall i(\exists\, j(C_{i,j}\neq 0\wedge\forall j'(j'>j\wedge C_{i,j'}=0))$$
$$\rightarrow\forall i'(i'>i\wedge C_{i',j}=0))\wedge\forall i(\exists\, j(C'_{i,j}$$
$$\rightarrow\forall i'(i'>i\wedge C_{i',j}=0))\wedge\forall i(\exists\, j(C'_{i,j}$$
$$\forall i'(i'>i\wedge C'_{i',j}=0)) \quad \text{(l)}$$

Algorithm IsClean. Since the reuse happening in the SRF is value reuse, the reuse happens only when the values to be reused must not be changed before the reuse. For a reference to array A whose values are to be reused and a write reference, we denote the subscript expressions of the reuse source reference as F_i and those of write reference as G_i. The values generated by the reuse source are not changed by the write reference in the subsequent *d* iterations if:

5) when the reuse source is used earlier than the write reference

$$\max\{\sum_{Q=1}^{M}F_Q(I_P=\{L_P,U_P\})\times N_Q\}<$$
$$\min\{\sum_{Q=1}^{M}F_Q(I_{P+1}+i,I_P=\{L_P,U_P\})\times N_Q(0\le i<d)\} \quad \text{(m)}$$

$$\text{or}\quad \min\{\sum_{Q=1}^{M}F_Q(I_P=\{L_P,U_P\})\times N_Q\}>$$
$$\max\{\sum_{Q=1}^{M}F_Q(I_{P+1}+i,I_P=\{L_P,U_P\})\times N_Q(0\le i<d)\} \quad \text{(n)}$$

6) when the write reference is earlier

$$\max\{\sum_{Q=1}^{M}F_Q(I_P=\{L_P,U_P\})\times N_Q\}<$$
$$\min\{\sum_{Q=1}^{M}F_Q(I_{P+1}+i,I_P=\{L_P,U_P\})\times N_Q(0\le i<d-1)\} \quad \text{(o)}$$

$$\text{or}\quad \min\{\sum_{Q=1}^{M}F_Q(I_P=\{L_P,U_P\})\times N_Q\}>$$
$$\max\{\sum_{Q=1}^{M}F_Q(I_{P+1}+i,I_P=\{L_P,U_P\})\times N_Q(0\le i<d-1)\} \quad \text{(p)}$$

2.2. Exploiting Loop-Carried Stream Reuse

Then we present our algorithm of ***E**xploiting **L**oop-carried **S**tream **R**euse, ELSR*. The algorithm ELSR

Copyright © 2010 SciRes.

```
DOI1=1,N
 DOI2−1,N
  DOI3=1,N
   C(I3,I2,I1)=(A(I3,I2,I1)+B(I3,I2,I1))
                *A(I3,I2+1,I1)
  ENDDO
 ENDDO
ENDDO
```
(a)

```
DOI1=1,N
 DOI2−1,N
  DOI3=1,N
   C(I3,I2,I1)=(A(I3,I2,I1)+B(I3,I2,I1))
                *A(I3+1,I2,I1)
  ENDDO
 ENDDO
ENDDO
```
(b)

```
DOI1=1,N
 DOI2−1,N
  DOI3=1,N
   A(I3+1,I2,I1)=(A(I3,I2,I1)+B(I3,I2,I1))
                *C(I3,I2,I1)
  ENDDO
 ENDDO
ENDDO
```
(c)

```
DOI1=1,N
 DOI2−1,N
  DOI3=1,N
   C(I3,I2,I1)=(A(I3,I2,I1)+B(I3,I2,I1))
                *A(I3+2,I2,I1)
  ENDDO
 ENDDO
ENDDO
```
(d)

Figure 6. FORTRAN code of applications to be optimized.

Table 1. Benchmark programs.

Name	Description
P2Q2d1	P=Q=2, d=1, and optimized by input stream reuse.
P2Q2d1l2	same application as P2Q2d1 except that we don't optimize it by stream reuse but organize array references of the innermost 2 loops as streams
P2Q2d1l3	same as P2Q2d1l2 except that array references of all 3 loops are organized as streams
P3Q3d1	P=Q=3, d=1, and optimized by input stream reuse
P3Q3d1O	same as P3Q3d1 except that it is optimized by output stream reuse
P3Q3d2	same as P3Q3d1 except that d=2
QMR.	ab. of QMRCGSTAB, a subspace method to solve large nonsymmetric sparse linear systems[14] whose coefficient array size is 800*800
MVM	a subroutine of a hydrodynamics application and computing band matrix multiplication with the size of 832*832
Laplace	calculating the central difference of two-dimension array whose size is 256*256

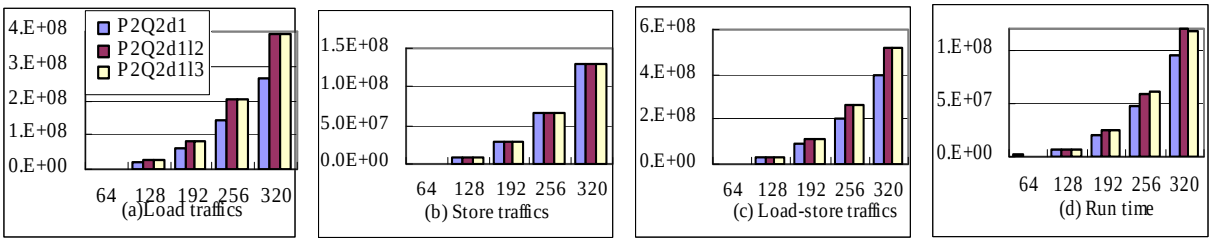

Figure 7. With the increase of array size the performance of different stream implementations of the application in 6(a) in respect of memory traffics (bytes) and run time (cycles).

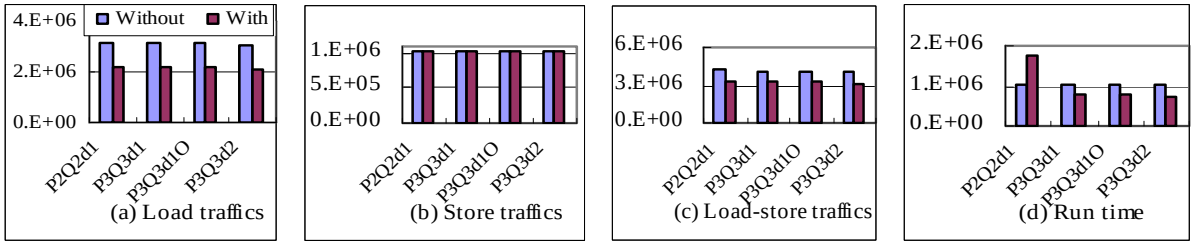

Figure 8. Performance of P2Q2d1, P3Q3d1, P3Q3d1O and P3Q3d2 with array size of 64 in respect of memory traffics (bytes) and run time (cycles).

consists of the following three steps to exploit the streams identified by algorithms ILISR, OLISR and IsClean:

Step A. Organize different array *A* references in the innermost D-P loops as stream A1 and A2 according their data traces.

Step B. Organize all operations on array *A* in the innermost D-P loops as a kernel.

Step C. Organize all operations in the outmost P loops as stream-level program.

When the nest with I_P=i of loop P in stream-level program operates on stream A1 and A2, one of them has been loaded into the SRF by the former nest, which means that the kernel doesn't get it from chip-off memory but the SRF. From the feature of the stream process-

Copyright © 2010 SciRes.

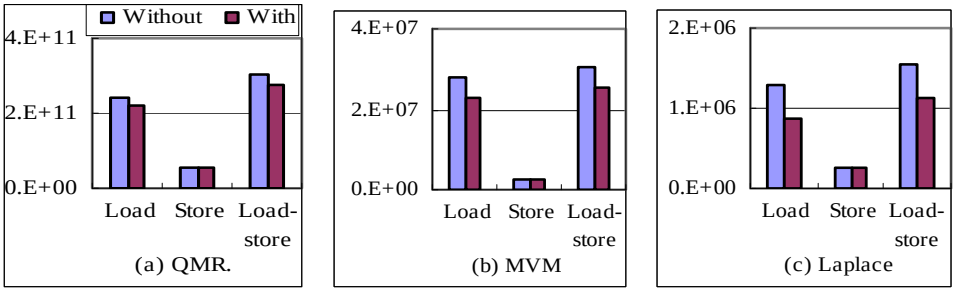

Figure 9. Effects of stream reuse on the memory traffics of scientific programs.

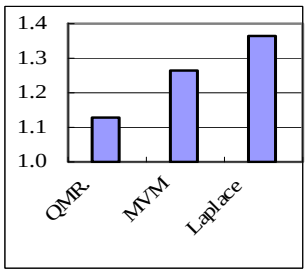

Figure 10. Speedup of scientific programs with stream reuse.

sor architecture, we can know the time to access chip-off memory is much larger than that to access the SRF, so the method of stream reuse can improve stream program performance greatly.

In stream program MVM unoptimized, we organize different array QP read in loop1 as three streams according their data trace ,and. organize operations in loop1 as a kernel The length of each stream is 832*832. When running, the stream program must load these three streams into the SRF, the total length of which is 692224 *3, nearly three times of that of array QP.

By the stream reuse method above, we organize different array QP read references in loop2 as three streams according their own data trace, organize operations in loop2 as a kernel, and organize operations in loop1 except loop2 as stream-level program. Thus there would be 832*3 streams in the stream program loop1, and the length of each is 832. So in stream program loop1, stream QP(L), QP(L+NXD) and QP(L-NXD) of neighboring iterations can be reused. As a result, the stream program only need load 832 streams with the length of 832 from chip-off memory to the SRF, the total length of which is 692224, nearly 1/3 of that of unoptimized program.

3. Experiment

We compare the performance of microbenchmarks and several scientific applications optimized and unoptimized by stream reuse. All applications are run on a cycle-accurate simulator for a single-node Imagine stream processor, Isim [9,10].

Table 1 summarizes the test programs used for evaluation. Microbenchmarks listed in the upper half of the table stress particular aspects of loop-carried stream reuse, e.g., if there is an input stream reuse between adjacent loop nests in respect of loop 2 and dimension 2, the benchmark is named P2Q2d1. All microbenchmarks are stream programs of applications in Figure 6 in FORTRAN code. P2Q2d1, P3Q3d1, P3Q3d1O and P3Q3d2 are corresponding stream programs of 6(a), 6(b), 6(c) and 6(d), which is optimized by loop-carried stream reuse. P2Q2d1l2 and P2Q2d1l3 are corresponding stream programs of 6(a) without optimization. There are 2*N out of 4*N streams that can be reused as N stream in the SRF in every microbenchmark except P2Q2d1, in which there are $2*N^2$ out of $4*N^2$ streams that can be reused as N^2 stream in the SRF. Scientific applications listed in the lower half of the table are all constrained by memory access. 14994 out of 87467 streams in QMR can be reused as 4998 streams in the SRF, 3 out of 8 streams in MVM can be reused as 1 stream in the SRF, and 3 out of 5 streams in Laplace can be reused as 1 stream in the SRF.

Figure 7 shows the performance of different stream implementations of the application in 6(a) with the increase of array size. Figure 7(a) shows chip-off memory load traffics, Figure 7(b) shows store traffics, Figure 7(c) shows the total chip-off memory traffics, and Figure 7(d) shows the run time of these implementations. In Figure 7(a), the load traffics of P2Q2d1 are nearly 2/3 of the other two implementations whatever the array size is. This is because input loop-carried stream reuse optimization finds the loop-carried stream reuse, improves the locality of the SRF and consequently reduces the load memory traffics. In Figure 7(b) the store traffics of different implementations are the same because there is only input stream reuse, which has no effect on store traffics. From Figure 7(c), we can see that because loop-carried stream reuse reuses 2 input streams as one stream in the SRF, it cut down the total memory traffics obviously. In Figure 7(d), when the array size is 64, the run time of P2Q2d1 is larger than the other two implementations. When the array size is 128, the run time of P2Q2d1 is a little larger than the other two implementations. The reason for above is that when the array size is small, the

Copyright © 2010 SciRes.

stream length of P2Q2d1 is much shorter than and the number of streams are larger than the other two implementations. As a result, the overheads to prepare to load streams from chip-off memory to the SRF weigh so highly that they can't be hidden, including the time the host writes SDRs(Stream Descriptor Register) and MARs(Memory Access Register).With the increase of the array size, the run time of P2Q2d1 is smaller and smaller than the other two implementations. This is because with the increase of the stream length, the overheads to load streams into the SRF weigh more and more highly and consequently the overheads to prepare loading streams can be hidden well. The memory traffics of P2Q2d1 are the least and consequently the P2Q2d1 program performance is the highest.

Figure 8 shows the performance of P2Q2d1, P3Q3d1, P3Q3d1O and P3Q3d2 with array size of 64. Figure 8(a) shows chip-off memory load traffics, Figure 8(b) shows store traffics, Figure 8(c) shows the total chip-off memory traffics, and Figure 8(d) shows the run time of them. These applications are representative examples of loop-carried stream reuse. In Figures 8(a), 8(b) and 8(c), chip-off memory load, store and total traffics have similar characteristics as those in Figure 7. In Figure 8(d), the performances of all applications except P2Q2d1 have been improved by stream reuse optimization. The reason for the reduction of P2Q2d1 performance has been given above. The results show that these representative applications optimized by loop-carried stream reuse all get similar performance increase as that in Figure 7.

Figure 9 shows the effects of stream reuse on the memory traffics of scientific programs used in our experiments. Figure 10 shows the speedup yielded by scientific applications with stream reuse over without. All these applications are optimized by input stream reuse. From results, we can see that because all these applications are constrained by memory access, the improvement of application performance brought by stream reuse is nearly in proportion to the amount of streams that can be reused.

4. Conclusions and Future Work

In this paper, we give the algorithms of identifying loop-carried stream reuse for scientific applications and of exploiting the stream reuse. Several representative microbenchmarks and scientific stream programs with and without our optimization are performed on Isim which is a cycle-accurate stream processor simulator. Simulation results show that the optimization method can improve the performance of scientific stream program constrained by memory access efficiently.

In the future, we are devoted to developing more programming optimizations to take advantage of architectural features of the stream processor for scientific computing applications.

5. References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," Computer Architecture News, Vol. 23, No. 1, pp. 20–24, 1995.

[2] D. Burger, J. Goodman, and A. Kagi, "Memory bandwidth limitations of future microprocessors," In Proceedings of the 23rd International Symposium on Computer Architecture, Philadelphia, PA, pp. 78–89, 1996.

[3] S. A. William, "Stream architectures," In PACT 2003, September 27, 2003.

[4] Merrimac–Stanford Streaming Supercomputer Project, Stanford University, http://merrimac.stanford.edu/.

[5] W. J. Dally, P. Hanrahan, *et al.*, "Merrimac: Supercomputing with streams," SC2003, Phoenix, Arizona, November 2003.

[6] M. Erez, J. H. Ahn, *et al.*, "Merrimac-supercomputing with streams," Proceedings of the 2004 SIGGRAPH GP^2 Workshop on General Purpose Computing on Graphics Processors, Los Angeles, California, June 2004.

[7] J. B. Wang, Y. H. Tang, *et al.*, "Application and study of scientific computing on stream processor," Advances on Computer Architecture (ACA'06), Chengdu, China, August 2006.

[8] J. Du, X. J. Yang, *et al.*, "Implementation and evaluation of scientific computing programs on imagine," Advances on Computer Architecture (ACA'06), Chengdu, China, August 2006.

[9] M. Rixner, "Stream processor architecture," Kluwer Academic Publishers, Boston, MA, 2001.

[10] P. Mattson, "A programming system for the imagine media processor," Department of Electrical Engineering, Ph.D. thesis, Stanford University, 2002.

[11] O. Johnsson, M. Stenemo, and Z. ul-Abdin, "Programming & implementation of streaming applications," Master's thesis, Computer and Electrical Engineering, Halmstad University, 2005.

[12] U. J. Kapasi, S. Rixner, *et al.*, "Programmable stream processor," IEEE Computer, August 2003.

[13] G. Goff, K. Kennedy, and C. W. Tseng, "Practical dependence testing," In Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM, New York, 1991.

[14] T. F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. H. TongSIAM, "A quasi-minimal residual variant of the bi-cgstab algorithm for nonsymmetric systems," Journal on Scientific Computing, 1994.

Copyright © 2010 SciRes.