

Scientific Computing Applications on the Imagine Stream Processor^{*}

Jing Du, Xuejun Yang, Guibin Wang, and Fujiang Ao

School of Computer, National University of Defense Technology, Changsha 410073, China
jdstarry@yahoo.com.cn

Abstract. The Imagine processor is designed to address the processor-memory gap through streaming technology. Good performance of most media applications has been demonstrated on Imagine. However the research whether scientific computing applications are suited for Imagine is open. In this paper, we studied some key issues of scientific computing applications mapping to Imagine, and present the experimental results of some representative scientific computing applications on the ISIM simulation of Imagine. By evaluating the experimental results, we isolate the set of scientific computing application characteristics well suited for Imagine architecture, analyze the performance potentiality of scientific computing applications on Imagine compared with common processor and explore the optimizations of scientific stream program.

Keywords: scientific computing application, Imagine, stream, three level parallelism, multinest.

1 Introduction

Scientific computing applications widely used to solve large computation problems are pervasive and computationally demanding. These applications require very high arithmetic rates on the order of billions of operations per second. But the performance of these applications is restricted by both the latency and bandwidth of memory accessing [1][2]. Scientific computing applications often exhibit large degrees of data parallelism, and as such maybe present great potential opportunities for stream architectures [3][4], such as Imagine architecture [4]. Imagine is a programmable stream processor aiming at media applications [5], which contains 48 arithmetic units, and a unique three level memory hierarchy designed to keep the functional units saturated during stream processing [6][7]. With powerful supports of the architecture, Imagine can exploit the parallelism and the locality of a stream program, and achieve high computational density and efficiency [8]. In this paper, we describe and evaluate the implementation of mapping scientific computing applications to stream programs formed of data streams and kernels that consume and produce data streams on the Imagine stream architecture, and compare our results on a cycle-accurate simulation of Imagine. The purpose of our work is to exploit the salient features of these unique scientific computing applications, isolate the set of application characteristics best

^{*} This work was supported by the National High Technology Development 863 Program of China under Grant No. 2004AA1Z2210.

suitable for the stream architecture by evaluating the experimental results, and explore the optimizations of scientific stream program.

2 The Imagine Stream Processing System

2.1 Imagine Architecture

The Imagine architecture developed at Stanford University is a single-chip stream processor that operates on sequences of data records called streams, supporting the stream programming system. It is designed for computationally intensive applications like media applications characterized by high data parallelism and producer-consumer locality with little global data reuse [6][7]. The Imagine processor consists of 48-ALUs arranged as 8 SIMD clusters and three level memory hierarchy to ensure the data locality and keep hundreds of arithmetic units efficiently fed with data. Several local register files (LRFs), directly feed those arithmetic units inside the clusters with their operands. A 128 KB stream register file (SRF) reads data from off-chip DRAM through a memory system interface and sequentially feeds the clusters [8][9]. Fig. 1 diagrams the Imagine stream architecture. One key aspect of Imagine is the concept of producer-consumer locality, where data is circulated between the SRF and arithmetic clusters, thereby avoiding expensive off-chip memory access overhead [10]. Based on the foregoing architecture supports, Imagine can efficiently exploit data parallelism along three levels: instruction-level parallelism (ILP), data-level parallelism (DLP), and task-level parallelism (TLP).

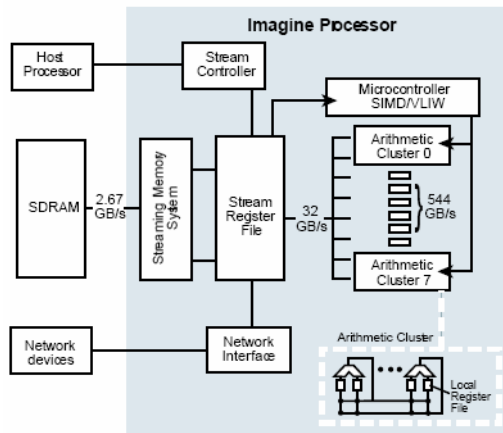


Fig. 1. The Imagine stream architecture

2.2 Imagine Programming Model

The programming model of Imagine is described in two languages: the stream level and the kernel level [11][12][13][14]. A stream level program is written in StreamC

language, which is derived from C++ language. A kernel level program of the clusters is written in KernelC language, which is C-like expression syntax. The StreamC program executed for the host thread represents the data communication between the kernels that perform computations. However, programmers must consider the stream organization and communication using this explicit stream model, increasing the programming complexity [15]. So the optimization for stream programming is important to achieve significant performance improvements on the Imagine architecture. The fine stream program can explore ILP, DLP and TLP to maximize performance, as it processes individual elements from streams in parallel.

3 Implementation of Scientific Computing Stream Applications

Imagine system promises to solve many computationally intensive problems much faster than their traditional counterparts. Scientific computing applications contain a great lot of loops possessing a high degree of instruction, data and task level parallelism that can be exploited by decomposing the scientific computing task into smaller subtasks, which are mapped into different computational elements, distributing the scientific stream to different processors. However, because a stream program is more complex than an equivalent sequential program, to realize this increase in speed some challenges must be overcome first [12].

3.1 Stream Level

The key tasks of stream level are partitioning kernels and organizing input streams. Since parallelizable parts focus on loops, we present corresponding streaming method based on different loop transformations. Aiming at exploiting ILP within a cluster, DLP among clusters and TLP of a multi-Imagine system, programmers distribute parallelizable data among the clusters and put the data that dependence can't be eliminated on the same cluster via loop analysis. Due to wire delay becoming increasingly important in microprocessor design, reducing inter-cluster communication must also be taken into account. It is necessary to modify the original algorithm when we write a stream program. We explicate our key methods in detail according to an example that is modified from a part of a scientific computing program named Capao introduced in the fourth section. Fig. 2 shows the example program including two main loops named loop1 and loop2 by us, and loop2 is a multinest with two inner loops labeled as loop3 and loop4 specially.

3.1.1 Multinest

In order to make the best use of the powerful computing ability of Imagine, kernel must process suitable granularity. Computationally intensive operations centre on multinest loops. It is a simple method to look upon each inner loop as a separate kernel. But this partition method brings memory access overhead due to replacing microcode frequently, and causes so much lower repeatable use ratio of SRF as to

```

1  alfa[0] = 0;
2  alfa[1] = b/c;
3  for (i=2;i<511;i++)
4      alfa[i]=b/(c-a*alfa[i-1]); } loop1
5  alfa[511]=0;
6  beta[0]=0;
7  for (j=1;j<511;j++)
8  {
9      for (i=1;i<511;i++)
10     {
11         f=t[i][j+1]-t[i][j];
12         beta[i]= (f+beta[i-1])/alfa[i-1]; } loop3
13     }
14     w[511][j]=0;
15     for (i=510;i>0;i--)
16         w[i][j]=alfa[i]*w[i+1][j]+beta[i]; } loop4
17     w[0][j]=0;
18 } } loop2

```

Fig. 2. Example program

make memory access become bottleneck. So that multinest loop is mapped into a big kernel results in better execution time than several small kernels. Because having more operations in one kernel gives more opportunities to parallel the operations and generates more compact schedules with better resource utilization. There are two key steps to partition multinest loops into kernel codes on Imagine.

- Loop combination

Combine the inner loops without array dependence by instruction scheduling. This way can increase the computing scale within kernels, and reduce the number of single instructions outside the inner loops.

- Loop splitting

If inner loops can't be combined, then consider splitting the multinest loop. In this way, the computing amount of outer loops can be involved in kernels, and accordingly parallelism of kernel level program can be improved. This method relates to array saving creating array copies. We can add one dimension based on original array to save the results of previous loops. It is a way that bartering space overhead for efficiency. For example, loop3 and loop4 in Fig. 2 exist array dependence. Hence we split the big multinest loop2 into two two-nest loops. The computing scale of kernels is increased from 510 to 510*510. For loop splitting, the dimension degree of array beta is increased to save the results of loop3, and prepare the input data for loop4. Fig. 3 shows loop2 is divided to two new multinest named loop3' and loop4' according to the dimension variety of array beta.

```

for (j=1;j<511;j++)
  for (i=1;i<511;i++)
  {
    f=t[i][j+1]-t[i][j];
    beta[i][j]=(f+beta[i-1][j])/alfa[i-1];
  }
for (j=1;j<511;j++)
{
  w[511][j]=0;
  for (i=510;i>0;i--)
    w[i][j]=alfa[i]*w[i+1][j]+beta[i][j];
  w[0][j]=0;
}

```

} loop3'

} loop4'

Fig. 3. Loop splitting

3.1.2 Coupled Dependent Loop

It is difficult that single loop existing data coupled dependence is parallelized. So expanding one dimension based on the original array within multinest to exploit parallelism on the new dimension is an optional means. Then we can choose multi-form methods of stream organization, aiming at exploiting parallelism among clusters and making full use of LRF according to the LRF capacity. For instance, in Fig. 2, the array beta in the twelfth row of the example code is expanded into two-dimension array. The new dimension direction j exists data coupled dependence, but there is independence between new columns. The coupled dependent code is as follows.

$$\text{beta}[i][j] = (f + a * \text{beta}[i-1][j]) / \text{alfa}[i-1].$$

For making full use of arithmetic units per cluster, we must avoid assigning the coupled dependent data to different clusters. Doing everything possible to place the coupled dependent data within a cluster can reduce the influence of wire delay, and improve parallelism on Imagine. There are two ways to solve this problem.

- Combine the coupled dependent record into a big record according to the capacity of LRF. Then make the big records form a new input stream. The implementation of this method is complex in some sort, but comprehended easily. We emphasize that the infilling of new records may flush the LRF. So the dependent records are loaded into LRF as successively as possible. At the same time, we must claim attention to save the array boundary of big record. Because the record may be as large as the capacity of LRF. When next record coming, we must save the previous record as the input data of the next operation to avoid record losing. Fig. 4 presents the stream organization of this method on Imagine with 8 clusters.
- Compared with the foregoing way, the second method is easy to implement, because the records of stream are not altered. We place the dependent record onto a cluster by a special index stream. Same as the foregoing method, every eight columns are treated as a group. The index stream is formed successively by row of independent records in a group. Each column of records is assigned onto a cluster. Fig. 5 presents the stream organization of this method.

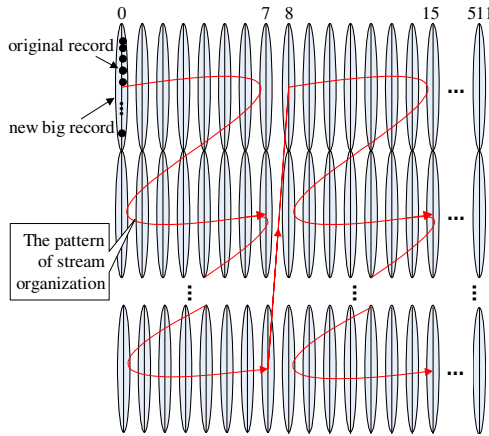


Fig. 4. The first method of coupled dependent loop mapping on Imagine

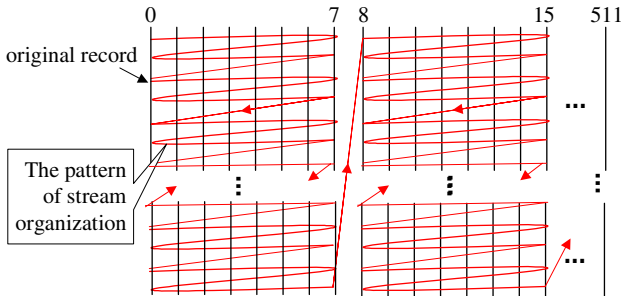


Fig. 5. The second method of coupled dependent loop mapping on Imagine

3.1.3 Single Instruction

If there are a great deal of single instructions in original program, whether they are within loops or not, the partition of kernel is influenced, and the kernel granularity can't be suitable. To solve the problem, we present two optimization methods.

- Loop expanding

In order to increase the computing scale of kernels and avoid using index stream that causes DRAM reordered overhead, some single instructions need to be expanded into appropriate loops. Then we can either use successive basic stream as input data or provide uniform loop variable for multinest combination. For instance, the second instruction of the example code in Fig. 2 is expanded into loop1.

- Instruction scheduling

When above factors are satisfied, on the premise of accuracy being ensured, this method may reduce the number of write times to the same record, and prepare for combining single instruction operations. For example, the fourteenth and seventeenth

instructions of the example code in Fig. 2 are scheduled out of loop2 to lessen the computing amount of loop4 so that loop4' in Fig. 3 can be mapped to stream program obviously. Fig. 6 illustrates the two methods of single instruction optimization.


<pre> alfa[0] = 0; alfa[1] = b/c; for (i=2;i<511;i++) alfa[i]=b/(c-a*alfa[i-1]); ... for (j=1;j<511;j++){ w[511][j]=0; for (i=510;i>0;i--) w[i][j]=alfa[i]*w[i+1][j]+beta[i]; w[0][j]=0; } </pre>		<pre> alfa[0] = 0; for (i=1;i<511;i++){ alfa[i]=b/(c-a*alfa[i-1]); w[511][i]=0; w[0][i]=0; } ... for (j=1;j<511;j++){ for (i=510;i>0;i--) w[i][j]=alfa[i]*w[i+1][j]+beta[i]; } </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 6. Single instruction optimizations

3.2 Kernel Level

An Imagine application is written as a sequence of smaller tasks, called kernels. A kernel operation performs a computation on a set of input streams to produce a set of output streams. Typically, kernels loop over an input stream, performing identical operations on each input element to produce their outputs. Each kernel runs on all eight clusters while processing its input streams and completes the processing of its input streams before the next kernel begins. In this way, producer-consumer locality is exploited by consuming the result of one kernel as soon as it is produced. As an example, Fig. 7 shows how the program in Fig. 2 is mapped to streams and kernels. In the event where inter-cluster communication is required, each cluster has a cluster id tag that can be used to identify the cluster and send/receive data to/from the right cluster. In order to expand the scale of kernel, a long stream is generally as input data. When computing data are not in native register, additional inter-cluster communications are required to transfer the data to the right cluster. And since all applications are not perfectly data parallel, many kernels require data reordering to place the data on the right clusters.

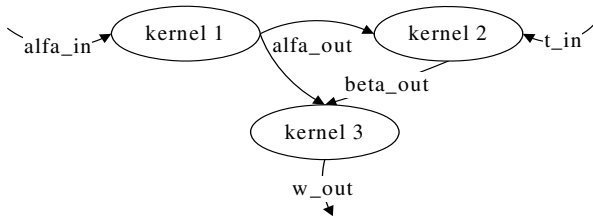


Fig. 7. Example program mapping to stream program model

4 Experimental Results and Analysis

We implement some scientific computing applications on ISIM that is a cycle-accurate simulator of Imagine [14], including 171.Swim in SPEC2000 and Capao.

4.1 Application Analysis

Swim is a weather prediction program for comparing the performance of current supercomputers. Fig. 8 shows data flow chart of Swim. Its main computing amount focuses on a loop of calculating fourteen arrays with 513×513 size. The data amount of Swim is large, but the computing operations are few correspondingly. The array access pattern presented in Fig. 9 is irregular.

Capao is an application on the field of optics. Its computing amount is very huge. According to its result of serial version, 65.49% of time overhead comes from subroutine dfft, and 13.36% comes from subroutine transp. So we just consider mapping the two subroutines to stream program so that improve performance of the whole application. The subroutine dfft possesses small computing amount and fine computation intensiveness. We implement two version of dfft. One that applies butterfly algorithm is called DFFTN in this section, and another formulized without any optimization is called DFFT. The computing amount is exponent distinction between DFFTN and DFFT. It is time-consuming on general scalar processors that DFFTN performs bit reverse operation. Imagine supports this operation on hardware level, so the performance of DFFTN may be increased. The experiment on DFFT purposes certifying powerful computing ability of Imagine.

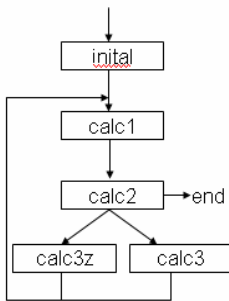


Fig. 8. The data flow chart of Swim

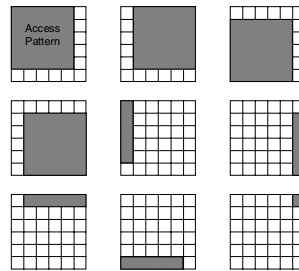


Fig. 9. Accessing pattern of Swim

4.2 Experimental Results

For comparison purpose, actual measurements of performance were taken using a general scalar processor system. Table 1 illustrates the result of a rough comparison between the performance of Imagine and the general scalar processor. It is obvious that Imagine provides high speedup of computationally intensive applications such as DFFTN, DFFT and Transp compared with general processor system in terms of number of cycles, due to the simple control logic and parallel processing ability of

many arithmetic units. And compared with highly sensitive to memory latency of general processor, these applications can hide latency to achieve good performance. But for data intensive applications such as Swim, the speedup is low due to irregular access pattern so that memory access latency can't be hidden.

Table 1. Comparison of different implementation for the scientific applications

	DFFTN	Swim	DFFT	Transp
Cycles (StreamC& KernelC)	52335	6689051178	1620615	9287445
Cycles (C code)	3705560	132895126660	28093910	158444624

Fig. 10 shows the three level bandwidth hierarchy of these applications. The LRF to memory bandwidth ratio are over 33:1, 70:1 and 592:1 across DFFTN, Transp and DFFT, due to the abundant memory access of these three applications focusing on LRF. So they can achieve good performance on Imagine with relatively low memory bandwidth for exposing a large register set with two levels of hierarchy to the compiler enables considerable locality to be captured that is not captured by a conventional cache. While the streams of Swim are very long, which can't be partitioned due to dependence, causing low locality of SRF and LRF. Notice that the bandwidth of LRF is much lower than that of SRF, because a mass of index streams derived dynamically inhabit the SRF space.

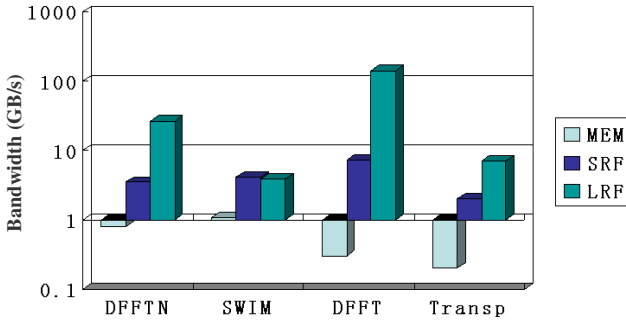


Fig. 10. Bandwidth hierarchy of applications

Table 2 presents the computation rate of these applications measured in the number of operations executed per second. Imagine achieves 16 GOPS ALU performance on media applications and sustains between 2% and 31% of the peak performance on these applications. On DFFT, Imagine averages 10 arithmetic operations per cycle across all the clusters for an aggregate rate of 5 GOPS. This high computation rate indicates that the stream programming system delivers high computational density on

the DFFT application. But for Swim, the computing time is 13%~38% of the whole run time. The great mass of work is to wait for result of memory accessing leading to inefficient performance.

Table 2. Computation rate of applications

	DFFT	Swim	DFFT	Transp
Cycle	52335	6733615303	1620615	9287515
Ops	209920	3409274134	16248880	8359056
GOPs	2.0	0.25	5.0	0.45

Fig. 11 shows the size of the computation kernel, as well as the number of arithmetic operations per memory access. Imagine's stream model requires large number of arithmetic operations per memory access to effectively use the underlying hardware. We can observe that Transp has enough bandwidth to sustain one operation per memory access, while DFFT and DFFT that are computationally intensive applications require high computation per memory access to amortize off-chip memory bandwidth. Swim characterized by irregular data access results in low computation per memory access, and the SRF is not used effectively since there is bad producer-consumer locality in this example. In conclusion, Swim is not well suited for the Imagine architecture. The performance is limited by memory bandwidth due to the relatively low computation per memory access.

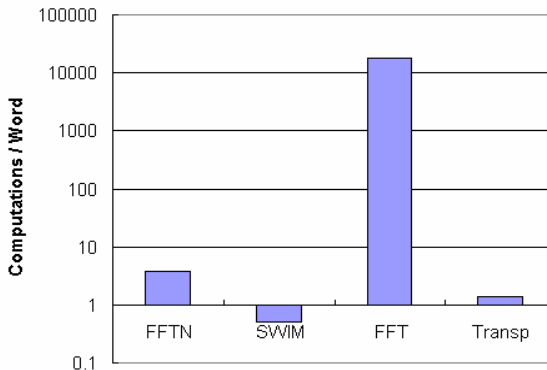


Fig. 11. Computational intensity of applications

4.3 Optimization

Aiming at solving the inefficiency problem of Swim, we apply some optimizations on the application. There are two levels of stream program optimized method.

4.3.1 Kernel Level Optimization

Computation is the bottleneck in the unoptimizable version of our stream programs not for saturation of the ALUs but for their poor utilization. The Imagine software environment allows for automatic code optimizations such as loop unrolling and software pipelining [12]. At the kernel level, the programmer can instruct the compiler to unroll/pipeline by simple compiler directives for program optimization. Then the loop in the cluster is unrolled and pipelined in order to achieve higher arithmetic intensity. The left part of Fig. 12 shows that the VLIW schedule of the unoptimizable code is quite sparse. The optimized schedule shown in the right part of Fig. 12 is dense. Fig. 13 presents that the computation time is reduced according to unrolling and pipelining of diverse times on identical program. We can conclude that unrolling four times is a critical point. Unrolling too many times increase loading overhead of the microcode with enlarging code amount.

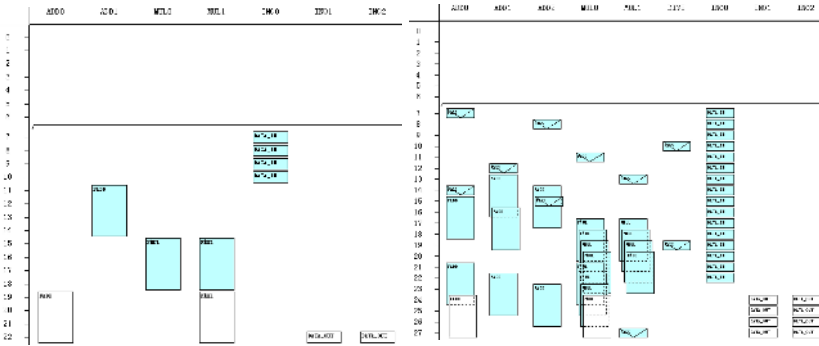


Fig. 12. Schedule diagram of kernel level optimization for Swim

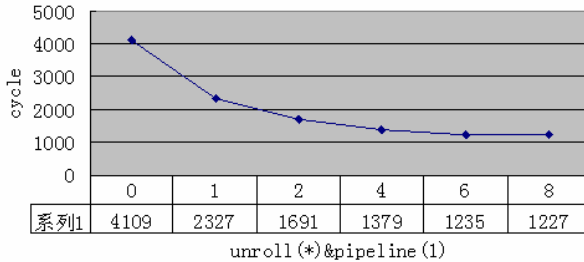


Fig. 13. Performance obtained from unrolling and pipelining optimizations

4.3.2 Stream Level Optimization

By exploiting kernel level optimization, the total execution time reduces. Based on the most perfect optimization in kernel, we adjust the input stream length to observe the performance variety. Fig. 14 shows that it gives more improvements with shorter input stream, and longer stream results in worse speedup. Specially, when the length

longer than 512×4 , performance is reduced sharply due to appearance of double-buffer. Optimization is invalid when the stream length greater than 512×32 , because the optimization increases microcode loading overhead with enlarging code amount.

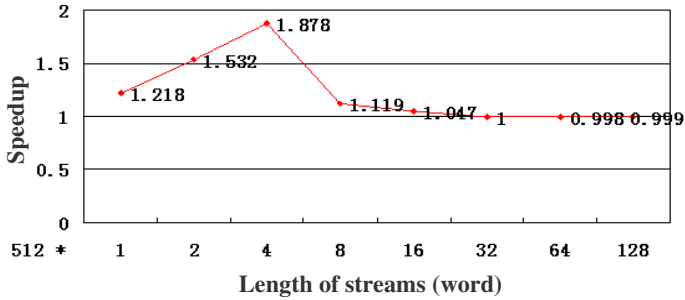


Fig. 14. Speedup obtained from varying stream length

Above analyses show that the organization of stream, especially the partition of long stream, influences on program performance deeply. To eliminate this bottleneck, it is necessary to reduce data transmission between memory and SRF so that the locality of SRF is enhanced. There are two optimizations of stream level accordingly, stripmining and Software pipelining.

The input streams of most applications are too large to fit the SRF directly. To solve this problem, stripmining is brought forward to process a great deal of input stream into small portions that fit in the SRF. Then the small input portions are applied to produce small portions of the final output that fit in the SRF. This optimization is important to achieve good performance [16].

Software pipelining divides a loop into sections so that the execution of one section in an iteration can be overlapped with execution of another section of another iteration. This optimization is implemented for exploiting producer-consumer locality and effectively hiding memory access overhead.

5 Conclusion and Future Work

In this paper, we explain the method of scientific computing applications mapping to stream programs, and present the experimental results. Partial programs fit for stream application, such as DFFT and DFFTN. For analyzing whether scientific computing applications are suited for stream architecture, we come up for discussion.

Three level parallelisms and two level data localities of Imagine architecture make the performance of scientific computing stream programs improve possibly. And the memory operations and computation overlapping can be propitious to cover the memory delay and implement optimizations, with the goal of keeping all the units busy at all times. Scientific computing applications often exhibit large degrees of data parallelism, and as such may be good candidates for SIMD stream applications. But comparing scientific computing applications with media applications, the former has irregular data organization, multiform data accessing pattern, new compiling

problems caused by large computing scale, much higher precision in calculation, and bandwidth in great demand. It is difficult to suit for the stream architecture completely. For the reason of making full use of the supports of stream architecture and exploiting the potentiality of scientific computing programs mapping to Imagine, we need to study the optimization algorithm of the existing stream programs. At present, the stream compiler is good at optimizing the stream code like loop unrolling, software pipelining, stripmining and so on, but the optimizations are restricted to algorithm of original stream programs. So we need to modify the original algorithm so that these optimizations can be performed effectively. For example, DFFTN achieves higher performance due to applying butterfly algorithm on DFFT. Stream organization and multi-level parallelism of the algorithm modified can exploit more potentiality of stream architecture, with higher computation per memory access and better data locality of LRF.

Through optimization of algorithm and compiler, there will certainly be a large class of scientific computing applications where stream architectures are more effective. Since there exists a lot of data parallelism in such applications, and the overhead of loading and changing kernels is amortized by large stream sizes [16][17][18]. Also, the memory operations and computation can overlap in order to hide the time spent in memory accesses, with large kernel of scientific computing stream programs. Furthermore, the amount of arithmetic units are enough to exploit data parallelism effectively, and memory accessing focuses on LRF and SRF mostly after optimizations to take advantage of consumer-producer locality so that make more efficient use of the memory bandwidth hierarchy. Powerful computational ability of stream architecture is emerged to sustain a high computation rate.

Future plans include exploiting common programming model to improve coding efficiency, due to existing program model exposing so many controls to programmers. Also, it is significant to construct a scientific computing kernel library that is valuable on algorithm design and shifting much of the complexity to the development of stream applications. This approach lowers the barrier to developer participation and can simplify collaborations among research teams by allowing each group to focus on their interests and expertise.

Acknowledgements. We gratefully thank the Stanford Imagine team for the use of their compilers and simulators and their generous help. We specifically thank lab 620 of School of Computer Science in National University of Defense Technology for helpful discussions and comments on this work. We also acknowledge the reviewers for their insightful comments. This work was supported by the National High Technology Development 863 Program of China under Grant No. 2004AA1Z2210.

References

1. W. A. Wulf, S. A. McKee. Hitting the memory wall: implications of the obvious. *Computer Architecture News*, 1995. 23(1): 20-24.
2. D. Burger, J. Goodman, A. Kagi. Memory bandwidth limitations of future micro-processors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, Philadelphia, PA, 1996.78-89.

3. Saman Amarasinghe, William. Stream Architectures. In PACT 2003, September 27, 2003.
4. B. Khailany et al. Imagine: Media processing with streams. *IEEE Micro*, 21(2): 35–46, March 2001.
5. Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucec Khailany, Jung Ho Ahn, Peter Mattson and John D.Owens. Programmable Stream Processors. *IEEE Computer*, pages 54-62, August , 2003.
6. Brucec Khailany, William J. Dally, Andrew Chang, Ujval J. Kapasi, Jinyung Namkoong, and Brian Towles. VLSI design and verification of the Imagine processor. In Proceedings of the IEEE International Conference on Computer Design, pages 289–296, September 2002.
7. Brucec Khailany. The VLSI Implementation and Evaluation of Area-and Energy-Efficient Streaming Media Processors. Ph.D. thesis, Stanford University, 2003.
8. Ujval J. Kapasi, William J. Dally, et al. The Imagine Stream Processor. In Processings of the 2002 International Conference on Computer Design, 2002.
9. Nuwan S. Jayasena. Memory Hierarchy Design for Stream Computing. Ph.D. thesis, Stanford University, 2005.
10. Scott Rixner, William Dally, Ujval J. Kapasi, Brucec Khailany, Abelardo Lopez-Lagunas, Peter Mattson and John D.Owens. Media processing applications on Imagine media processor. In Proceedings of the 2002 International Conference on Computer design, 2002.
11. Peter Mattson et al. Imagine Programming System Developer's Guide. <http://cva.stanford.edu>, 2002.
12. Peter Raymond Mattson. A Programming System for the Imagine Media Processor. Dept. of Electrical Engineering. Ph.D. thesis, Stanford University, 2002.
13. Saman Amarasinghe et al. Stream Languages and Programming Models. In PACT 2003, September 27, 2003.
14. Abhishek Das, Peter Mattson, et al. Imagine Programming System User's Guide 2.0. June 2004.
15. Ola Johnsson, Magnus Stenemo, Zain ul-Abdin. Programming & Implementation of Streaming Applications. Master's thesis, Computer and Electrical Engineering Halmstad University, 2005.
16. Jinwoo Suh, Eun-Gyu Kim, Stephen P. Crago, Lakshmi Srinivasan, and Matthew C. French. A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels. In ISCA03, 2003.
17. Mattan Erez, Jung Ho Ahn, Ankit Garg, William J.Dally et al. Analysis and Performance Results of a Molecular Modeling Application on Merrimac. In SC'04, Pittsburg, Pennsylvania, USA, November 6-12, 2004.
18. Jung Ho Ahn, William J. Dally, et al. Evaluating the Imagine Stream Architecture. In ISCA2004, 2004.