© 2014 IJIRT | Volume 1 Issue 6 | ISSN : 2349-6002

Compiler Vectorization For Architecture

Himanshu Kapoor,Amritanshu Sharan
Dronacharya College Of Engineering
Khentawas,Gurgaon

Abstract— This paper deals with compiler technology that aims on low-power DIGITAL SIGNAL PROCESSOR(DSP) architecture.The architecture includes obvious part of exploitation of data and Instruction Level Parallelism.For data manipulation it requires large register file with dynamically composed vectors.We also discuss how vector register file are used by optimizing compiler for data ccess pattern.We describe various new challenges that DSP architecture presents.It invokes various new opportunities for optimized compiler that is introduced.The optimized compiler aims such an architecture for achieving performance which must to be compared to hand generated code.Compiler technology in area of DSP compilation represents a state of art.The main motive is only to implement the process of vectorization by using vectorizing compiler.The most time consuming and error prone work is to write the vector codes in assembly language

Index Terms — Loop unrolling, Multimedia extension, Modulo scheduling, Parallelism, Software pipelining, Vectorization, Vector-length

1 INTRODUCTION

MULTIMEDIA extensions are the biggest advancement in processor architecture in past decade. Now a days they are predominant in embedded systems and also in general-purpose computers. The structure of multimedia applications often contains compute-intensive kernels that operate on independent streams of data. Multimedia extensions provide vector instructions that operate on relatively short vectors. Vector instructions works on SIMD (single instruction multiple data) architecture. Common examples are MMX, SSE and AltiVec . Over time multimedia extensions have grown increasingly complex. They offer extensive vector instruction sets, including support for floating-point computation. So, these short-vector instructions need some kind of improvement. Multimedia extensions may contain combination of vector and scalar instructions. Automatic compilation faces two major challenges: identifying vector instructions in sequential descriptions, and using this information to deal with short-vector instructions efficiently. There are two important issues with multimedia extensions- memory alignment and selection of vector code. This paper presents a review of different schemes which are available to address selection of code issues.

2 CODE SELECTION

When targeting the general-purpose architectures, the most important performance consideration is the overall code generation scheme. There are two classic techniques for extracting parallelism: vectorization and software pipelining.
Researchers first developed vectorizing technique [1],[2] for vector supercomputers such as Cray-1[3]. More recently this technique is adopted for compilation to multimedia extensions[4],[5],[6],[7],[8],[9],[10],[11]. Vectorization identifies the instructions in which the same operations operate on multiple data items concurrently.
Software pipelining is a method for exploiting ILP (instruction- level parallelism). Modulo scheduling is a popular approach of software pipelining. Most software pipelining implementations are based on Rau's iterative modulo scheduling heuristic. Iterative modulo scheduling was first developed for one of the first VLIW machine Cyndra 5 [12],[13].

3 VECTORIZATION

Vectorization is a process which converts a sequential loop into a parallel version that can utilizes a processor's vector instruction set. This transformation involves reordering of instructions. As the main work of vector instruction is to compute multiple values before committing any of the results. In normal programming language we write a loop as-

```
execute this loop 10 times
    read the next instruction and decode it
    fetch first number
    fetch second number
    add them
    store the result
end loop
```

After vectorization the above loop looks like-

```
read instruction and decode it
fetch 10 numbers
fetch another 10 numbers
add them
store result
```

Vectorization is only legal if it preserves dependencies in the original loop. Data flow analysis [14],[15] is a method which shows dependencies among scalar variables. There must be an accurate identification of dependences among memory operations as they can prevent vectorization.

Allen and Kennedy [1] and Wolfe [2] had provided overview of techniques for memory dependence analysis. Data dependences are of three types-

1. Flow (or true) dependences occur when op1 writes a value which is read by op2. If this type of dependency is present between instructions then they must be execute sequentially op1 and then op2.

2. Antidependences occur when op1 reads a location which op2 subsequently overwrites.

3. Output dependences occur when op1 writes a value which op2 subsequently overwrites.

Dependences in a loop further classified as loop-carried and loop-independent, depending upon whether they occur between operations in different iterations or the same iteration, respectively. For loop carried dependence we associate a distance vector with each loop.

```
for (i=0;i<n;i++)
  for(j=0;j<n;j++)
  for(k=0;k<n;k++)
       x[i+1][j+3][k+2]=x[i][j][k];
```

The above loop contains flow dependence from store to load with distance vector (1, 3, 2). Vectorization operates at the source level representation but low level vectorization is also possible if the compiler computes dependences early and saves the information.
The basic steps for vectorizing a loop are as follows. Consider the following loop-

```
for(i=0;i<n;i++)
{
  1.a[i]=b[i]+c[i];
  2.b[i+1]=a[i]+d[i];
  3.c[i+1]=e[i]+f[i];
}
```

The data dependence graph of the above loop is-

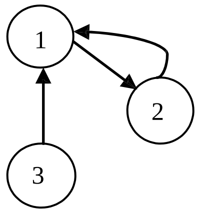

Fig. 1. Data dependence graph of the above loop

Then identify the strongly connected components in graph. Statements involved in cycle must execute sequentially and rests are vectorizable.
Statement 1 and 2 form a strongly connected component so must execute sequentially. Statement 3 is vectorizable.

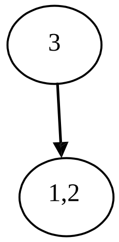

Fig. 2. After combining strongly connected components

The result after vectorization is

```
c[1:n+1]=e[0:n]+f[0:n];
for(i=0;i<n;i++)
{
     a[i]=b[i]+c[i];
     b[i+1]=a[i]+d[i];
}
```

The vector operation execute first, even though in the original code it appears later. This results from the topological sort and is necessary to preserve the dependences from statement 3 to statement 1.
Maximum number of multimedia architectures does not provide specialized hardware support for transferring data between scalar and vector registers. In this case the communication overhead is unavoidable.

4 SUPERWORD-LEVEL PARALLELISM

Vector supercomputer works more efficiently when operates on long vector. Superword-level parallelism is obtained from unrolling data-parallel loops. Often, it is present in multimedia codes that perform the same operation on the red, green and blue components of a pixel, as in alpha bending

```
for(i=0;i<n;i++)
{
   blend[i].r=a*fore[i].r+(1-a)*back[i].r;
   blend[i].g=a*fore[i].g+(1-a)*back[i].g;
   blend[i].b=a*fore[i].b+(1-a)*back[i].b;
}
```

In cases where data parallelism is available across loop iterations, unrolling converts loop-level parallelism to superword-level parallelism. Consider a loop

```
for(i=0;i<n;i++)
{
    a[i]=b[i]+c[i];
}
```

Unrolling this loop by a factor of 2

```
for(i=0;i<n;i+=2)
{
     a[i+0]=b[i+0]+c[i+0];
     a[i+1]=b[i+1]+c[i+1];
}
```

SLP can be implemented by different algorithms. The algorithm, using simple greedy heuristic [16], locates groups of isomorphic expressions and replaces them with vector opcodes. Isomorphic expressions are vectorizable when they are independent. Data dependence analysis is simpler for SLP than for traditional vectorization because it can ignore loop-carried dependences. Independent isomorphic expressions may not vectorizable in all cases. Consider the following loop:

```
for(i=0;i<n;i++)
{
   a[i+1]=b[i];
   b[i+1]=a[i];
}
```

Suppose we have architecture of vector length of 2. So we can unroll the loop by factor 2.

```
for(i=0;i<n;i++)
{
    a[i+1]=b[i];
    b[i+1]=a[i];
    a[i+2]=b[i+1];
    b[i+2]=a[i+1];
}
```

In the above code there is no dependences from statement 1 to statement 3, a single vector can execute both. Same is true for statement 2 and 4. Combining both pairs wll results the following code:

```
for(i=0;i<n;i++)
{
    a[i+1:i+2]=b[i:i+1];
    b[i+1:i+2]=a[i:i+1];
}
```

There is a dependence cycle in the original loop which means full vectorization is impossible. So the above code is invalid. The SLP heuristic [16] might produce the following:

```
for(i=0;i<n;i++)
{
    b[i+1]=a[i];
    a[i+1:i+2]=b[i:i+1];
    b[i+2]=a[i+1];
}
```

To determine when vectorizable is profitable, superword-parallelization employs an unsophisticated cost metric.

5 SOFTWARE PIPELINING

Software pipelining is a static scheduling technique for overlapping iterations of a loop. The technique takes advantage of ILP hardware to execute operations from multiple iterations concurrently.
The blocks in Fig 3(a) represent the dependence between operations of an inner loop. If we schedule the loop in a traditional manner, the blocks must execute sequentially in order to follow dependences. Fig 3 (b) represents parallel execution of instructions with software pipeline. The schedule preserves dependences between instructions since each state operates in a different iteration. The kernel executes repeatedly and forms the steady-state of the software pipeline. The prolog and epilog fill and drain the pipeline, respectively.

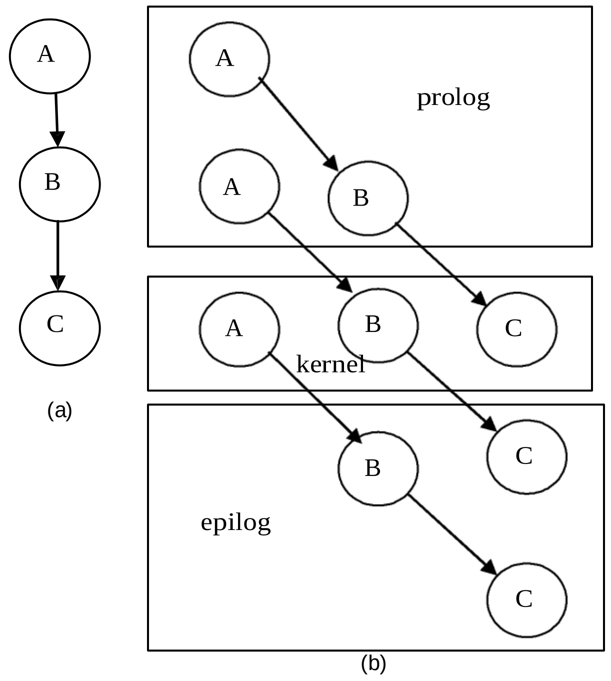

Fig.3.(a) dependence graph of inner loop, (b) dependence graph after applying software pipelining

Most existing algorithms generate modulo schedules. A modulo schedule is an ordering of instructions in an inner loop which preserves dependences and no resource conflicts. Rau’s iterative modulo scheduling heuristic [19], [20] forms the basis of many software pipelining implementations. Modulo scheduling is applicable to any architecture providing ILP hardware. When available, however it benefits from rotating registers and predication [12], [13]. Rotating registers provide a mechanism to queue multiple writes to the same nominal register. If they are unavailable in the target architecture, the compiler can employ unrolling and scalar renaming [18]. Predication provides a means for modulo scheduling loops with control flow.

6 SELECTIVE VECTORIZATION [17]

The aim of selective vectorization is to divide the operations between scalar and vector operations in order to maximize loop performance. The algorithm [22] is concerned with balancing resources and ignores the latency of all operations. If dependence cycle is present in the graph then full vectorization is not possible unless the dependence distance is greater than or equal to the vector length. For example, a loop statement a [i+3] =a[i] has a dependence cycle but can be vectorized for vector lengths of four or less. Selective vectorization algorithm is based on Kernighan and Lin's two-cluster partitioning heuristic [21]. The algorithm divides instructions between scalar and vector partition. All operations are originally placed in scalar partition. Operations are moved one at a time between the partitions, searching for a division that minimizes cost function. Once each operation has been repartitioned, the configuration with the lowest cost is used as the starting point for the next iteration. The algorithm stops when an iteration fails to improve the starting point. Operations remaining in the vector partition are vectorized. Consider the following loop:

```
for(i=0;i<n;i++)
{
   s=s+x[i]*y[i];
}
```

Dependence graph of the above loop

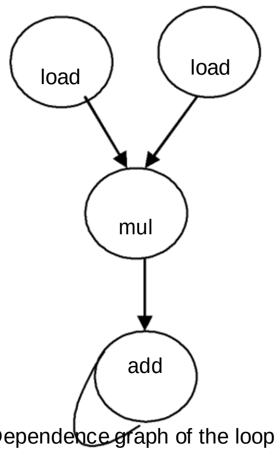

Fig. 4. Dependence graph of the loop

Now we create modulo schedule of the above loop with an II of 2.

TABLE 1
MODULO SCHEDULE OF THE LOOP

Cycle	Slot1	Slot2	Slot3
1	Load(1)	Load(2)	
2	Mul(1)		
3	Load(2)	Load(2)	Add(1)
4	Mul(2)		
…	…	…	…

Modulo schedule table for full vectorization

TABLE 2
MODULO SCHEDULE USING FULL VECTORIZATON

Cycle	Slot1	Slot2	Slot3
1	VLoad(1-2)		
2	VLoad(1-2)		
3	VMul(1-2)		
4	VLoad(3-4)	Add(1)	
5	VLoad(3-4)	Add(2)	
6	VMul(3-4)		
…	…	…	…

Table 2 shows that if we use full vectorization it gives an II of 1.5. Now distribute the loop into vector and scalar loops by using selective vectorizaton.

```
for(i=0;i<n;i+=2)
{
    t[i:i+1]=x[i:i+1]*y[i:i+1];
}

for(i=0;i< n;i++)
{
     s=s+t[i];
}
```

The modulo scheduling table for the above loop is

TABLE 3
MODULO SCHEDULE USING SELECTIVE VECTORIZATION

Cycle	Slot1	Slot2	Slot3
1	VLoad(1-2)	Load(1)	
2		Load(2)	
3	VLoad(3-4)	Load(3)	
4	VMul(1-2)	Load(4)	
5	VLoad(5-6)	Load(5)	Add(1)
6	VMul(3-4)	Load(6)	Add(2)
…	…	…	…

By analyzing table 3 we can say that it gives an II of 1.0.

4 CONCLUSION

We have shown different techniques of compilation of short-vector instructions. These instructions are common in multimedia extensions and now days they are important part of embedded and general-purpose computers. These techniques are differing in their work and complexity. Selective vectorization promises to give the best result but the algorithm is very much complex. Different heuristics can be used in selective vectorization. The selection of techniques depends on the underlying architecture.

ACKNOWLEDGMENT

The authors would like to acknowledge MHRD, Govt. of India for providing support by Teaching Assistantship scheme.

REFERENCES

[1] Randy Allen and Ken Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, San Francisco, California, 2001.

[2] Michael J. Wolfe, *High Performance Compilers for Parallel Computing* . Addison-Wesley, Redwood City, California, 1996

[3] Richard M. Russel, –The CRAY-1 Computer System‖. *Communications of the ACM*,21(1):63{72, January 1978.

[4] Aart J.C.Bik, *The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance.* Intel Press, Hillsboro, OR, 2004.

[5] Gerald Cheong and Monica Lam, –An Optimizer for Multimedia Instruction Sets*". In Second SUIF Compiler Workshop*, August 1997.

[6] Derek J. DeVries, –A Vectorizing SUIF Compiler: Implementation and Performance,‖ Master's thesis, University of Toronto, June 1997.

[7] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien, –Vectorization for SIMD Architectures with Alignment Constraints,‖ *Proc. SIGPLAN '04 Conference on Programming Language Design and Implementation*, pp. 82-93, Washington,DC, June 2004.

[8] Dorit Naishlos,‖ Autovectorization in GCC,‖ *Proc. of the 2004 GCC Developers Summit*, pp. 105-118, 2004.

[9] Dorit Naishlos, Marina Biberstein, Shay Ben-David,and Ayal Zaks, –Vectorizing for a SIMD DSP Architecture,*" Proc. of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 2-11, San Jose,CA, October 2003.

[10] N. Sreraman and R. Govindarajan,‖ A Vectorizing Compiler for Multimedia Extensions,‖*International Journal of Parallel Programming*, 28(4):363-400, August 2000.

[11] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao , –An Integrated Simdization Framework Using Virtual Vectors,‖ *Proc. of the 19th ACM Inter-national Conference on Supercomputing* , pp.169-178, Cambridge, MA, June 2005.

[12] James C. Dehnert, Peter Y.T. Hsu, and Joseph P. Bratt,‖Overlapped Loop Support in the Cydra 5,‖ *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26-38, Boston, MA,April 1989.

[13] B. Ramakrishna Rau, David W.L.Yen, Wei Yen, and Ross A.Towle, –The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions and Trade-offs,‖ *Computer*, 22(l):12-35, Jan. 1989.

[14] Alfred V. Aho, Ravi Sethi, and J. D. Ullman, *Compilers Principles, Techniques,and Tools* Addison-Wesley, 1986.

[15] Steven S. Muchnick, *Advanced Compiler Design and Implementation* Morgan Kaufmann,San Francisco, California, 1997.

[16] Samuel Larsen and Saman Amarasinghe –Exploiting Superword Level Parallelism with Multimedia Instruction Sets,‖ *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 145-156, Vancouver, BC, June 2000.

[17] Samuel Larsen,‖Compilation Techniques for Short-Vector Instructions‖, Ph.D. Thesis,Dept. of Electrical Engg, & Computer Science, Massschusetts institute of technology,April 2006

[18] Monica Lam,‖ Software Pipelining: An Effective Scheduling Tech- nique for VLIW Machines,‖ *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 318-328, Atlan- ta, GA, June 1988.

[19] B. Ramakrishna Rau,‖Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops,‖ *Proc. of the 27th Annual International Symposium on Microarchitecture*, pp. 63-74, San Jose, CA, November 1994.

[20] B. Ramakrishna Rau,‖Iterative Modulo Scheduling,‖Technical Report HPL-94-115, Hewlett Packard Company, November 1995

[21] B. Kernighan and S. Lin,‖An Efficient Heuristic Procedure for Partitioning Graphs,‖ Bell System Technical Journal,49:291–307, February 1970.

[22] Samuel Larsen, Rodric Rabbah and Saman Amarasinghe,‖Exploiting Vector Parallelism in Software Pipelined Loops,‖*Proc. of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005.