Compilers II

Chapter 1:
Dependence analysis in loops with array assignments, and its applications

Lecturer: *Paul H J Kelly*

Spring 1997

Department of Computing
Imperial College

Room 423
email: phjk

Introduction

In this segment of the course we consider loops involving arrays. We will see

- How execution order of a loop is constrained,
- How a compiler can extract dependence information, and
- How this can be used to optimise a program.

Transforming the execution order of a loop enables various architectural features to be used efficiently, in particular:

- Pipelined processors.
- CPUs which can process more than one instruction at a time (superscalar and VLIW).
- Systems which rely heavily on caches.
- CPUs with special instructions for vectors.
- Multiprocessors.

- Here we consider a special kind of optimisation, which is currently performed only by specialist compilers — "restructuring compilers".
- Conventional optimisations (see the Dragon book) must also be performed.

The difference is this:

- ⋆ Conventional optimisations reduce the amount of work the computer has to do at run-time.
- ⋆ Restructuring aims to do the work in an order which suits the target architecture better.

Course structure:

This segment of the course is structured as follows:

- **Motivation:** execution order really matters, even on fairly "ordinary" architectures.
- **Dependence analysis:** how a compiler can analyse array subscript expressions in order to determine whether restructuring would be valid - e.g.
  - Loop interchange
  - Loop fusion
  - Loop reversal
  - Loop parallelisation (for parallel or pipelined machines).
- **Loop transformation using unimodular matrices:** given the dependence structure, how does the compiler efficiently find the optimum loop transformation?

Motivation: an example

- We will begin by looking at matrix multiplication.
- We will study the performance of a 500×500 double-precision floating point matrix multiply.
- We will investigate the performance of various versions in order to determine what transformations a compiler should be applying.

Initial version

```
/*
 * mm: Multiply A by B leaving the
 *     result in C.
 *     The result matrix is assumed
 *     to be initialised to zero.
 */
void mm1(A,B,C)
     double A[500][500],
            B[500][500],
            C[500][500];
{
  int i, j, k;

  for (i = 0; i < 500; i++)
    for (j = 0; j < 500; j++)
      for (k = 0; k < 500; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```


Architectural details

The experiments were performed on a SPARC 10 model 41 processor with 64 MB RAM:

- This machine has a 40MHz "SuperSPARC" pipelined RISC CPU.
- There are separate floating point units for add/subtract and multiply/divide.
- FP add: one cycle latency, FP multiply: three cycles latency (both fully pipelined).

The compiler used was the GNU C compiler `gcc`, version 2.5.7.

Memory system

- The chip includes a 20KB instruction cache (32-bit block, 5-way set-associative) and a 16KB data cache (32-bit block, 4-way set-associative).
- The system includes a direct-mapped write-back 1MB secondary cache with 4-word cache blocks.
- Note that the matrix occupies $500^2 \times 8 \approx 2$MB.
- Each row of the matrix occupies $500 \times 8 \approx 4$KB.

Performance

- The initial version runs in 118 seconds.
- The matrix multiplication takes 500^3 steps, each involving two floating-point operations, an add and a multiply, i.e. 250×10^6.
- This loop achieves a computation rate of 250/118=2.1 MFLOPs.
- This machine is advertised as having a peak performance of more than 10 MFLOPs.

  How are we going to get value for money?

If you want to play...

- The source code for the experiments can be found in the following files in the directory `~phjk/MatrixMultiplyExample`:

  `mm-interchanges.c`
  Compile using `gcc -O`.

  `mm-experiment.c` **and** `innerloop.s`
  Compile together using `gcc -O`.

  The source code includes the full results tables.
- Details of the SPARCstation 10 SuperSPARC architecture are given in the Postscript file `SuperSPARCWhitePaper.ps` which can be viewed using `ghostview`.

Accumulate into a scalar

```
for (i = 0; i < 500; i++)
  for (j = 0; j < 500; j++){
    r = 0.0;
    for (k = 0; k < 500; k++){
      r += A[i][k] * B[k][j];
    }
    C[i][j] = r;
  }
```

- Avoid repeated assignments to memory (note write-through primary cache).
- 94 seconds (2.65 MFLOPS vs 2.1 MFLOPS: 25% faster).

Interchange loops

```
for (i = 0; i < 500; i++)
  for (k = 0; k < 500; k++){
    r = A[i][k];
    for (j = 0; j < 500; j++)
      C[i][j] += r * B[k][j];
  }
```

- Why might this be a good idea?
- 45 seconds (5.5 MFLOPS).
- How might a compiler perform this transformation?
- Can we do better still?

Examine the inner loop

This is the innermost loop after strength reduction and induction variable elimination:

```
L113:
   ldd [%o0],%f2
   fmuld %f6,%f2,%f2
   ldd [%o1],%f4     ! uses res of prev instn
   faddd %f2,%f4,%f2 ! uses res of prev instn
   std %f2,[%o1]     ! uses res of prev instn
   add %o0,8,%o0
   cmp %o0,%o2
   blu L113
   add %o1,8,%o1     ! (delay slot)
```

- Register `o0` points to `B[k][j]`.
- Register `o1` points to `C[i][j]`.
- Register `o2` points to `C[i][500]`.
- Register `f6` holds `r`.
- There is some freedom in the order of instructions...

Instruction scheduling

We can reorder the instructions so that useful work is done while the floating point operations and loads take place:

```
L313:
   ldd [%o0],%f2
   fmuld %f6,%f2,%f2
   ldd [%o1],%f4
   add %o1,8,%o1    ! fwd 2 to fill ldd delay
   faddd %f2,%f4,%f2
   add %o0,8,%o0    ! fwd 3 to fill addd delay
   cmp %o0,%o2
   blu L313
   std %f2,[%o1-8] ! bwd 4 to avoid addd delay
```

- Instruction schedule depends on hardware structure.
- Note need to adjust address in `std`.
- 33.5 seconds (7.46 MFLOPS).
- Note that there is still a stall for the first load...

Software pipelining

We could avoid the stall due to the first load by executing it earlier — i.e. during the previous iteration. What we want to do is this:

```
   ldd [%o0],%f2   ! load f2 for first itern
L313:
   fmuld %f6,%f2,%f2
   ldd [%o1],%f4
   ldd [%o0+8],%f2 ! load f2 for next itern
   add %o1,8,%o1
   faddd %f2,%f4,%f2
   add %o0,8,%o0
   cmp %o0,%o2
   blu L313
   std %f2,[%o1-8]
```

This doesn't quite work as the load clobbers `f2`. To fix this we must modify the register allocation: use `f8` for the result of the load.

We get

```
   ldd [%o0],%f8   ! load f8 for first itern
L313:
   fmuld %f6,%f8,%f2
   ldd [%o1],%f4
   ldd [%o0+8],%f8 ! load f8 for next itern
   add %o1,8,%o1
   faddd %f2,%f4,%f2
   add %o0,8,%o0
   cmp %o0,%o2
   blu L313
   std %f2,[%o1-8]
```

- Note that work is being done concurrently on both iteration `j` and on iteration `j+1`.
- This idea is called "software pipelining". It is possible to spread a loop over several iterations — with ever more complicated run-up and run-down code.
- 33.2 seconds (7.53 MFLOPS).

Have we reached the limit?

Software pipelining didn't help us much in this example. Have we reached the peak performance for this loop?

Try more aggressive scheduling:

```
   ldd [%o0],%f8   ! load f8 for first itern
L313:
   ldd [%o1],%f4
   fmuld %f6,%f8,%f2 ! use f8 from prev itern
   ldd [%o0+8],%f8   ! load f8 for next itern
   add %o1,8,%o1
   add %o0,8,%o0
   faddd %f2,%f4,%f2 ! use res of fmul and ldd
   cmp %o0,%o2
   blu L613
   std %f2,[%o1-8]   ! use res of fadd
```

- No measurable performance improvement.

What about unrolling?

Unrolling the loop should have two benefits: it should allow even better scheduling, and it should reduce the amount of loop management overhead per iteration. E.g.

```
   ldd [%o0],%f2   ! load f2 for first itern
L513:
   fmuld %f6,%f2,%f2
   ldd [%o1],%f4
   faddd %f2,%f4,%f2
   ldd [%o0+8],%f8 ! from itern below
   std %f2,[%o1]
   fmuld %f6,%f8,%f8
   ldd [%o1+8],%f4
   add %o1,16,%o1
   faddd %f8,%f4,%f8
   ldd [%o0+16],%f2 ! move fwd from next itern
   add %o0,16,%o0
   cmp %o0,%o2
   blu L513
   std %f8,[%o1-8]
```

- 31.1 seconds (8.0 MFLOPS).

Optimising memory accesses

- What is the data cache behaviour of our program?
- How can it be improved?
- Why do cache hits occur?
  - Spatial locality
  - Temporal locality
- Why do cache miss occur?
  - Compulsory (never seen before)
  - Capacity (cache not big enough)
  - Conflicts (cache not associative enough)

How can we arrange for the data to be in the cache already?

Blocking

The idea of "blocking" (a.k.a. "tiling") is to reorder the execution of the loop nest so that data is not evicted from the cache before it is needed again. E.g.

```
for (kk = 0; kk < 500; kk += BLKSZ)
 for (jj = 0; jj < 500; jj += BLKSZ)
  for (i = 0; i < 500; i++)
   for (k = kk; k < min(kk+BLKSZ,500); k++){
    r = A[i][k];
     for (j = jj; j < min(jj+BLKSZ, 500); j++)
      C[i][j] += r * B[k][j];
    }
```

- The inner i,k,j loops perform a multiplication of a pair of $BLKSZ \times BLKSZ$ partial matrices.
- $BLKSZ$ is chosen so that a $BLKSZ \times BLKSZ$ submatrix of `B` and a row of length $BLKSZ$ of `C` can fit in the cache.

- Total memory traffic reduced from $2N^3 + N^2$ to $2N^3/BLKSZ + N^2$.
- What is the right value for $BLKSZ$?

Performance of blocked version

Using software-pipelined inner loop:

Blocking factor	Execn time	MFLOPS
8	34.7 seconds	7.2 MFLOPS
16	25.9	9.7
24	23.5	10.6
32	22.9	10.9
40	23.2	10.8
48	25.7	9.7
56	27.7	9.0
64	28.6	8.7
72	27.9	9.0
80	28.4	8.8

Summary

- Original version: 118 seconds (2.1 MFLOPS).
- That was using a "good" optimising compiler!
- Factor of five performance improvement.
- No reduction in amount of arithmetic performed.

Conclusions

- "Sophisticated" architectures are commonplace.
- Conventional optimisation must be augmented with techniques to deal with complex architectures. Examples include
  - Instruction scheduling.
  - Software pipelining.
  - Loop interchange.
  - Blocking.
- The compiler writer must solve two distinct problems:
  - Analysis: under what precise conditions can each transformation be applied?
  - Transformation: given the semantic constraints, what is the best code to generate?