

COMP3320/COMP6464: Loop Optimisation

Alistair Rendell

See: *High Performance Computing*, Dowd and Severance, Chapter 7 and 8

COMP3320 Lecture 11-0 Copyright © 2008 The Australian National University

11.1 The Quest for Speed

Many HPC applications take days or weeks to run. Performance is critical. To improve performance we can:

1. Buy a faster computer
2. Make program changes to improve the algorithm-architecture interaction, eg
 - Eliminate clutter: code that contributes to run-time but not to the result
 - Expose more instruction level ||ism to exploit fully superscalar architectures
3. Seek a more efficient algorithm, eg $O(N \log N)$ rather than $O(N^2)$
4. Use multiple CPUs, ie parallelise our code

COMP3320 Lecture 11-1 Copyright © 2008 The Australian National University

11.2 Procedure Calls: Problems

- Procedures in general are very useful:
 - ✓ improve modularity, code re-use
- but also:
 - × have lots of (startup) overheads
 - eg. parameter passing, register saving, call/return
 - × cause loss of instruction level ||ism within (calling) loop:
 - × cause loss of register variables (when compiler can't assume the call will not overwrite a register)

COMP3320 Lecture 11-2 Copyright © 2008 The Australian National University

Procedure Calls: Problems#2

- For performance procedure calls should be of high granularity, they should not be in the inner loops of your program, ie NOT like this

```
for (i = 0; i < N; i++){
    a[i] = average(b[i],c[i]);
}
--snip--
double average(double x, double y){
    return ( x + y ) * 0.5;
}
```

COMP3320 Lecture 11-3 Copyright © 2008 The Australian National University

11.3 Procedure Calls: Inlining

- Subroutine overhead can be alleviated by inlining the function into the loop. This is similar to (but more elaborate than) what is done using CPP macros, eg.

```
#define average_macro(x,y) ((X+Y)/2)

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

Potential problems with macros and inlining:

- Macros can cause line length problems
- Inlining:
 - Increases binary size
 - May cause register spills
 - Compiler must be able to "see" inlining source code and has to "decide" when should or should not (eg too complex/long) be inlined

11.4 Branches within Loops

- Can create (control) dependencies, ie. reduce instruction level ||ism
- **Conditional assignments:** results in variable assignment, least harmful

(i) 'unnecessary' ⇒ can be removed

```
• for (i = 0; i < N; i++){
    if (abs(x[i]) > 1.0e-16) y[i] = y[i] + a * x[i];
}

• for (i = 0; i < N; i++){
    if (x[i] >= 0.0) {a = a + x[i];}
    else{a = a - x[i];}
}
```

(ii) depend only on the loop index ⇒ 'split' the loop

```
• for (i = 0; i < N; i+=2){
    a = a + abs(x[i]);
    if (i+1 <= N) a = a + ABS(x[i+1]);
}
```

(iii) iteration independent ⇒ can 'unroll'

```
for (i = 0; i < N; i++){
    if (b[i] < 1.0) a[i] = a[i] + b[i]*c;
}
```

Branches within Loops #2

- **Conditional branches:** transfers control based on if test, more costly:

```
for (i = 0; i < N; i++){
    if (b[i] = 0){
        printf(" error \n");
        abort();
    }
    a[i] = a[i] / b[i];
}
```

(In fact IEEE standard requires that the above is trapped so you don't need this conditional at all)

- Instr'n pipeline flushing can be reduced by improving branch prediction
 - If possible organise your data to minimize changes in the conditional

11.5 Subexpression Elimination

- **Common subexpression elimination:** The ability of the compiler to recognise repeated patterns in the code and replace all but one with a temporary variable

```
c = a + b + d
e = q + a + b
Becomes
temp = a + b
c = temp + d
e = q + temp
```

- Simple cases are "usually" recognised by the compiler, but what if written as:

```
c = a + b + d
e = a + q + b
```

- More complicated examples are also not eliminated:

```
x = r*sin(a)*cos(b)
y = r*sin(a)*sin(b)
```

Compiler does not know if value of a is changed in the (sin(a)) function call

- In isolation these savings are minor. But in the inner loop in the kernel of your code they may be significant

11.6 Loop Unrolling and Software Pipelining

- Purposes: $\begin{cases} \checkmark & \text{reduce loop (control) overheads} \\ \checkmark\checkmark & \text{increase potential operation ||ism} \end{cases}$

- eg. daxpy loop:

```

for (i = 0; i < N; i++){           ! Ultra Instr'n Groups
    y[i] = y[i] + a * x[i]         ! Issue          Completes
}                                  //repeat [10]
    x0 = x[i]                     ! [1] ld(x0)
    y0 = y[i]                     ! [2] ld(y0)      ..st(y0)
    x0 = x0 * a                   ! [3] fmul(x0,a)  ld(x0)
    y0 = y0 + x0                 ! [4] -          ld(y0)
    x0 = x0 * a                   ! [5] -
    y0 = y0 + x0                 ! [6] fadd(x0,y0) fmul(x0,a)
    x0 = x0 * a                   ! [7] -
    y0 = y0 + x0                 ! [8] -
    y[i] = y0                    ! [9] st(y0),blt(i,n) fadd(x0,y0)
}

```

- (Load/store latency to level 1 cache is 2 cycles, while fmul/fadd is 3 cycles)
- Loop takes 9 cycles to complete 1 iteration
 - In 4 cycles no instructions are issues!
 - Only once do we use the superscaler capabilities (st(y0),blt(i,n))

Loop Unrolling and Software Pipelining#2

- What if we "unroll" the loop by a factor of 2.

```

for (i = 0; i < N%2; i++){         ! 'preconditioning' loop
    y[i] = y[i] + a * x[i];
}
for (i = N%2; i < N; i+=2){
    y[i] = y[i] + a * x[i];
    y[i+1] = y[i+1] + a * x[i+1];
}

```

- Exposes more possibilities for instruction ||ism
 - We are software pipelining the operations

Loop Unrolling and Software Pipelining#3

```

for (i = N%2; i < N; i+=2){       ! Ultra Instr'n Groups
    x0 = x[i]                     ! Issue          Completes
    x1 = x[i+1]                   ! [1] ld(x0)      ..st(y0)//Repeat[11]
    y0 = y[i]; x0 = x0 * a        ! [2] ld(x1)      ..st(y1)
    y1 = y[i+1]; x1 = x1 * a     ! [3] ld(y0),fmul(x0,a) ld(x0)
    y0 = y0 + x0                 ! [4] ld(y1),fmul(x1,a) ld(x1)
    y1 = y1 + x1                 ! [5] -          ld(y0)
    y0 = y0 + x0                 ! [6] fadd(x0,y0) ld(y1),fmul(x0,a)
    y1 = y1 + x1                 ! [7] fadd(x1,y1) fmul(x1,a)
    y0 = y0 + x0                 ! [8] -
    y1 = y1 + x1                 ! [9] st(y0)      fadd(x0,y0)
}                                  ! [10] st(y1),blt(i,n) fadd(x1,y1)

```

- Now obtain 2 results every 10 cycles, or effectively 1 result every 5 cycles
- Further unrolling will give 1 result every 3 cycles, ie
 - ultimately the loop is load/store dominated.
- **Note:** poor instr'n mix at start of loop

Loop Unrolling and Software Pipelining#4

- Greater unrolling also permits better hiding of L2 cache load/store latencies (8 cycles instead of 2!)
- With moderate levels of optimisation (eg -O3) compilers generally unroll inner loops automatically. But you may need to look at the assembler code to see exactly what is done.
- In general unrolling is inadvisable when loop:
 1. has a low trip count: ie. N is small
 - because extra setup is needed
 2. body is already fat \Rightarrow register spilling
 - generally, the unrolling should match (the register level of) the memory hierarchy
 3. has (unavoidable) procedure calls
- × **note:** unrolling increases code size

11.7 Loops with Inter-Iteration Dependencies: Reductions

- Hard to extract any instruction ||ism at all!
- eg. 'scan' an array:

```

y[1] = 0.0
for( i = 2; i<N; i++){
    y[i+1] = y[i] + x[i];
}

```

- **Reductions:** special case, inter-iteration dependencies is over a scalar variable
- eg. inner product of 2 vectors

```

s = 0.0;          | s = 0.0;
for(i = 0; i < N; i++){ | for(i=0;i<N;i++){   ! Cycle
    s = s + x[i] * y[i]; |     x0 = x[i]      ! [1] //repeat [10]
}                   |     y0 = y[i]      ! [2]
                   |     x0 = x0 * y0   ! [4] wait ld(y0)
                   |     s = s + x0     ! [7] wait mult(x0,y0)
                   |     wait add(s,x0)
                   | }

```

- 9 cycles for 1 iteration

Loop with Inter-Iteration Dependencies: Reductions#2

- Unrolling inner product by 2:

```

...
for(i = N%2; i < N; i+=2){
    x0 = x[i]          ! [1] //repeat [13]
    x1 = x[i+1]        ! [2] //add(s,x1) completes
    y0 = y[i]          ! [3]
    y1 = y[i+1]        ! [4]
    x0 = x0 * y0       ! [5]
    x1 = x1 * y1       ! [6]
    s = s + x0         ! [8] wait mult(x0,y0)
    s = s + x1         ! [11] wait add(s,x0)
                    ! wait add(s,x1)
}                   ! loop book-keeping overlaps

```

- 12 cycles for 2 results (cf 9 cycles for 1 iteration)
- How can performance be further improved?
 1. Change order at start of loop
 2. Remove dependencies of += op'ns

Loop with Inter-Iteration Dependencies: Reductions#3

```

s1=0; s2=0;
for(i = N%2; i < N; i+=2){
    x0 = x[i]          ! [1] //repeat [10] wait mult(x0,y0)
    y0 = y[i]          ! [2] <<NEW order
    x1 = x[i+1]        ! [3]
    y1 = y[i+1]; x0 = x0 * y0 ! [4] <<OVERLAP
    x1 = x1 * y1       ! [5]
    s1 = s1 + x0       ! [7] wait mult(x0,y0)
    s2 = s2 + x1       ! [8] wait mult(x1,y1)
}                   ! loop book-keeping overlaps
s = s1 + s2

```

- 9 cycles for 2 results (cf 12 cycles for 2 results)

11.8 Loop Overheads and Loop Interchange

- For nested loops, put the 'optimal' one innermost: ie. the one having the:
 - Highest trip count
 - Optimal memory reference patterns, eg. smallest stride
 - Best instr'n mix (= operation balance) (this is rarer)
- Loop startup overhead generally consists of setting up registers for loop index and bounds & other (eg. 'induction') variables, eg matrix copy could be written as:

```

if (M <= N) {
    for(i = 0; i < M; i++){
        for(j = 0; j < N; j++)B[i][j] = A[i][j];
    }
} else {
    for(j = 0; j < N; j++){
        for(i = 0; i < M; i++)B[i][j] = A[i][j];
    }
}

```

- Interchange only possible if no iteration dependency exists between i and j loops