Algorithms and Computation in Signal Processing

special topic course 18-799B
spring 2005
20^{th} Lecture Mar. 24, 2005

Instructor: Markus Pueschel
TA: Srinivas Chellappa

Assignment 3 - Feedback

Peak Performance Calculation

- Operations considered depend on the application considered
  - For numerical algorithms typically operations = floating point adds and mults (floating point operations)
  - (If algorithm needs only adds, then operations = adds)

- Peak performance: The maximum number of operations per second the computer can complete.
  Usually needs the manual.

- For operations = floating point ops, peak performance is measured in FLOPS (floating point ops/second).

- Loads and stores are not counted (and if, it would change the peak performance)

Performance Measurement

- Performance = number of operations / second
- For operations = floating point ops also measured in FLOPS
- Needs:
  - Runtime
  - Number of operations
- Number of operations
  - Either measure (using a tool like PAPI)
  - Or count ops executed in code. But also examine assembly code since compiler may optimize ops away.
- Comparing to peak performance gives an idea how far away from a theoretical optimum

Submitted code - feedback

```
for (i = 0; i < 10000000; i++) {
        temp1 += 0.5;
        temp2 *= 0.5;
        temp3 += 0.5;
        temp4 *= 0.5;
}
```

- 266/1700 MFLOPS, gcc -02, P4
- Good:
  - Instruction parallelism; adds and mults
- Bad:
  - Loop body too short; constant may not be reused

Submitted code - feedback

```
for (i=1 to N) {
    a = a+num;
    b = b+num;
    ..
    f = f+num;
}
```

- Does not state processor, compiler
- 1449/800 MFLOPS for add only. 1919/1600 for Add+multiply

- Pentium 4 allows 1 add/cycle

- Incorrect determination performance

Submitted code - feedback

```
for (i=0; i<N; i+=2)
  for (j=0; j<N; j+=4) {
    s1 = x[i][j] + x[i][j+1];
    s11 = x[i][j+2] + x[i][j+3];

    s2 = x[i+1][j] + x[i+1][j+1];
    s22 = x[i+1][j+1] + x[i+1][j+2];

    st1 = s1 + s11;
    st2 = s1 + s22;
    s = st1 + st2;

    sum += s;
  }
```

- P4, gcc -02, 1400mhz
- Reported MFLOPS: 92.8%
- Maybe counted index computations
- Can hardly be true (arrays, double loop, short loop body, dependencies)

Submitted code - feedback

- G4 1500mhz, peak 2400mhz (every 5th cycle stall, deep in the manual)
- FMA instructions only
- No dependencies across any 5 cycles
- 99.5% peak
- Loop body (part):

```
f0 = f0 * f1 + f1;
f2 = f2 * f3 + f3;
f4 = f4 * f5 + f5;
f6 = f6 * f7 + f7;
f8 = f8 * f9 + f9;
f10 = f10 * f0 + f0;
f1 = f1 * f2 + f2;
f3 = f3 * f4 + f4;
f5 = f5 * f6 + f6;
f7 = f7 * f8 + f8;
f9 = f9 * f10 + f10;
f0 = f0 * f1 + f1;
…
```

Submitted code - feedback

```
for () {
y1+= inc; y2+=inc; y3+=inc; y4+=inc; y5+=inc; y6+=inc; y7+=inc;
... <1000 times>
}
```

- No machine, no compiler flags
- Reported peak performance: 96.3%
- Exactly 8 variables, instruction parallelism

Submitted code - feedback

```
for (i=0; i < 1000000; i++) {
    a0 += a1; a2 += a3; a4 += a5; a6 += a7; a8 += a9;  a10 += a11;
  a12 += a13; a14 += a0;
  mults
....
}
```

- Sun blade sparc IIi, 500 mhz, 1gflops peak, gcc –O3
- 74% peak performance
- Considered different loop bodies
- Surprisingly small (a14 – a0 dependency?)

Submitted code - feedback

```
for(i = 0; i < 33333333; i++){
   asm("fadd   %st,%st(1)");
   asm("fmul   %st,%st(2)");
   asm("fadd   %st,%st(3)");
…..<80 times>
}
```

- P4 2.4 ghz, gcc -03
- 84% peak performance
- Good part: actual executed code guaranteed
- Asm can break instruction scheduling

Submitted code - feedback

```
for (j=0; j<iteration_num; j++){
    recursive part for multiplication and addition

    a0 = a0*const0_val;
    a1 = a1+const0_val;
    b0 = b0*const0_val;
    b1 = b1+const0_val;
    c0 = c0*const0_val;
    c1 = c1+const0_val;
     ....
}
```

- P4 1.8ghz, gcc -02
- 82% Peak performance

General Feedback

- State computer, compiler and flags
- Discuss what you do
- Explain how you computed performance
- Be suspicious if it was too easy, or results seem strange

Achieving high performance

- Sufficient computation: e.g., loop
- Reduce impact of branching instruction:
(partially) unroll loop, but not so far to get i-cache misses
- Use scalar variables (so compiler does proper analysis and register allocation)
- Avoid loads:
  - reuse variables
  - make sure variable set fit into register
- Keep all units busy
  - Use adds and mults (exceptions: e.g., P4)
  - Sufficient instruction-level parallelism
- Use good compiler and flags

Things to remember

- Understand what FLOPS performance is, and why it is important in numerical computing
  - How is it computed
  - Allows to compare to an upper bound
  - Careful: FLOPS performance is not runtime; an algorithm with higher FLOPS rate may still be slower because it has more operations
- For algorithm containing more than floating point adds and mults one needs to adjust analysis
  - For example other operations may need to be considered
  - E.g., a comparison a > b usually requires one add

Cost of Cooley-Tukey FFT

- Blackboard
- Example induction pitfall