

CS528

Serial Code Optimization

A Sahu

Dept of CSE, IIT Guwahati

Ref Books : Text

- Hager G and Wellein G . ***Introduction to High Performance Computing for Scientists and Engineers*** (1st ed.). CRC Press,, India, 2010.

Outline

- Intro to Code Optimization
- Machine independent/dependent optimization
- Common sense of Optimization
 - Do less work, avoid expensive Ops, shrink working set
- Simple measure Large impact : simd, branch, comm sub expre
- C++ Optimization
- Scalar Profiling
 - Manual Instrumentation (get_wall_time, clock_t)
 - Function and line based profiling (gprof, gcov)
 - Memory Profiling (valgrind, callgraph)
 - Hardware Performance Counter (oprofile,likwid)

Code Optimization & Performance

- Machine-independent opt
 - Code motion, Reduction in strength, Common sub-expression Elimination
- Machine-dependent opt
 - Pointer code, Loop unrolling, Enabling instruction-level parallelism
- Tuning: Identifying perf bottlenecks
- Understanding processor Opt
 - Translation of INS into Ops, OOO
- Branches
- Caches and Blocking
- Advice

Speed and optimization

- Programmer
 - Choice of algorithm, Intelligent coding
- Compiler
 - Choice of instructions, Moving code, Reordering code, Strength reduction, *Must be faithful to original program*
- Processor
 - Pipelining, Multiple FU, Memory accesses, Branches, Caches
- Rest of system : Uncontrollable, OS, Load

Great Reality

There is more to performance than asymptotic complexity

- Constant factors matter too!
 - Easily see 10:1 performance range depending on how code is written
 - Must optimize at multiple levels:
 - Algorithm, data representations, procedures, and loops

Great Reality

There is more to performance than asymptotic complexity

- Must understand system to optimize performance : 3 things
 - How programs are compiled and executed
 - How to measure program performance and identify bottlenecks
 - How to improve performance without destroying code modularity, generality, readability

Optimizing Compilers

- Provide efficient mapping of program to machine
 - Register allocation
 - Code selection and ordering
 - Eliminating minor inefficiencies
 - Don't (usually) improve asymptotic efficiency
 - Up to programmer to select best overall algorithm
 - Big-O savings are (often) more important than constant factors
- But constant factors also matter

Optimizing Compilers

- Have difficulty overcoming “optimization blockers”
 - Potential memory aliasing
 - Potential procedure side effects

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraint
 - Must not cause any change in program behavior under any possible condition
 - Often prevents making optimizations that would only affect behavior under pathological conditions
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
 - E.g., data ranges may be more limited than variable types suggest

Limitations of Optimizing Compilers

- Most analysis is performed only within procedures
 - Whole-program analysis is too expensive in most cases
- Most analysis is based only on *static* information
 - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Need to Do

- Do not rely on compiler completely
- You try to write good code
- Optimize your own code

Machine-Independent Optimizations

- Optimizations you should do regardless of processor / compiler

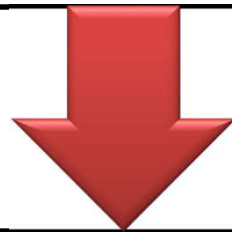
Common sense of Optimizations

Common sense of Optimizations

- Do less work
- Shrink the working set
- Avoid expensive operations
 - Strength reduction: Convert costly to cheaper OPS
 - LUT
- Subexpression elimination

CSO: Do less work

```
bool  Flag=false;
for  (i=0; i<N; i++) {
    if (complex_func(A[i]) < THRESHOLD )
        Flag=true;
}
```

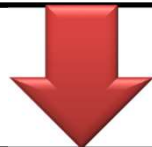


```
bool  Flag=false;
for  (i=0; i<N; i++) {
    if (complex_func(A[i]) < THRESHOLD )
        { Flag=true; break; }
}
```


CSO: Shrink the working set

- Working set of a memory is amount of memory it uses
- Use less memory, it may fit into cache, less misses
- Example Histogram Equalization of X-ray Image (generally a 8-bit/pixel Gray image)

```
// Large M,N  
unsigned int Image[M][N];
```



```
// Large M,N  
unsigned char Image[M][N];
```

CSO: Code Motion

- Reduce frequency with which computation performed
 - If it will always produce same result
 - Especially moving code out of loop

```
for (i=0; i<n; i++)  
    for (j=0; j<n; j++)  
        a[n*i+j] = b[j];
```



```
for (i = 0; i < n; i++) {  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
}
```

Most compilers do
a good job with
array code +
simple loop
structures

Strength Reduction

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

$$16 * x \quad \longrightarrow \quad x \ll 4$$

- Utility is machine-dependent
- Depends on cost of multiply or divide instruction
- On Pentium II or III, integer multiply only requires 4 CPU cycles. **In Corei3/i5: one cycle**
- **Power consumption of MultOp > ShiftOp**
- Recognize sequence of products

CSO: Avoid expensive operations

```
int L, R, U, O, S, N; //can be +1 or -1
double tt=0.83;
for (i=0; i<ALargeN; i++){
    GetNxtValOfSpin(i, &L, &R, &U, &O, &S, &N);
    BF[i]=0.5*(1+tanh((L+R+U+O+S+N)/tt)); //Costly
}
```



```
int L, R, U, O, S, N; //can be +1 or -1
double tt=0.83, BTanhLUT[14];
for(i=-6;i<=6;i++)
    BTanhLUT[i+6]=0.5*(1+tanh(i/tt));
for (i=0; i<ALargeN; i++){
    GetNxtValOfSpin(i, &L, &R, &U, &O, &S, &N);
    BF[i]=BTanhLUT[L+R+U+O+S+N+6]; //Cheaper
}
```

CSO: Strength Reduction

```
for (i = 0; i < n; i++) {  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
}
```



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

CSO: Strength Reduction

- FP division is **Slow** as compared to add/mul
 - $X = x / 3.0 \implies x = x * 0.33333$
- Avoid transcendental functions
 - Sin, cos, tan : take huge amount of time
 - Use Look-up Table (LUT) : pre-compute if possible and use them

```
for(i=0;i<10000;i++){  
    if(i%2==0) S=S+sin(Pi/4);  
    else S=S+sin(Pi/8);  
}
```

```
LUT[0]=sin(Pi/4); LUT[1]=sin(Pi/8);  
for(i=0;i<10000;i++){  
    if(i%2==0) S=S+LUT[0];  
    else S=S+LUT[1];  
}
```

CSO: Share Common Sub-expressions

- Reuse portions of expressions
- Compilers often unsophisticated about exploiting arithmetic properties

```
/* Sum neighbors of i,j */  
  
up=val[(i-1)*n+j]; down=val[(i+1)*n+j];  
  
Left=val[i*n+j-1]; right=val[i*n+j+1];  
  
sum = up+down+left+right;
```

3 multiplications: $i*n$, $(i-1)*n$, $(i+1)*n$

CSO: Share Common Sub-expressions

- Reuse portions of expressions
- Compilers often unsophisticated about exploiting arithmetic properties

```
/* Sum neighbors of i,j */  
  
up=val[ (i-1)*n+j ]; down=val[ (i+1)*n+j ];  
  
Left=val[ i*n+j-1 ]; right=val[ i*n+j+1 ];  
  
sum = up+down+left+right;
```

3 multiplications: $i*n$, $(i-1)*n$, $(i+1)*n$

CSO: Share Common Sub-expressions

- Reuse portions of expressions
- Compilers often unsophisticated about exploiting arithmetic properties

```
int inj = i*n + j;  
Up= val[inj - n]; down = val[inj + n];  
Left=val[inj -1]; right= val[inj + 1];  
sum = up+down+left+right;
```

1 multiplication: i*n