CS528
High Performance Computing

Serial Code Optimization

A Sahu
Dept of CSE, IIT Guwahati

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  --> x << 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 == > 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

CSO: Loop Jamming

```
for(i=0;i<10000;i++){
     Dostuff(i);//Small Independent work
}
for(i=0;i<10000;i++){
     DoMorestuff(i); //Small Independent work
}
```

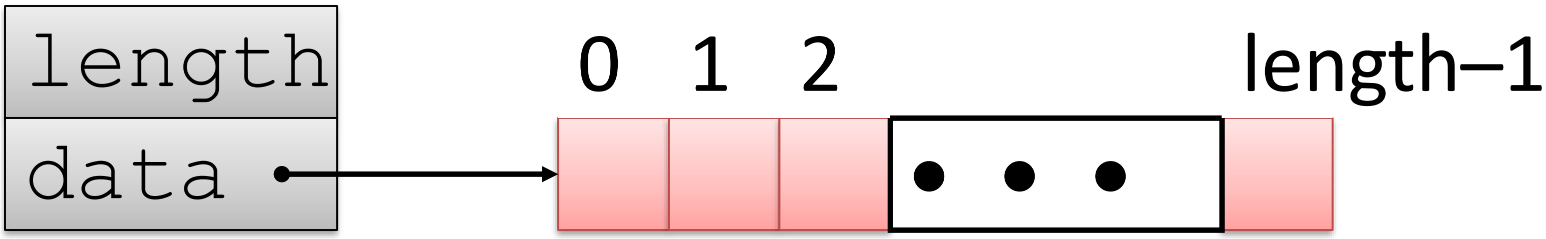
```
for(i=0;i<10000;i++){
     Dostuff(i);
     DoMorestuff(i);
}
```

CSO: Function Looping

```
for(i=0;i<10000;i++){
     Func(t,i);
}
Fun (int w, d){//do lots of stuff}
```

```
funn(t);
void funn(w){
 for(i=0;i<10000;i++){//do lots stuffs
   }
}
```

CSO: Example: Vector ADT




```
vec_ptr new_vec(int len)
```

- Create vector of specified length

```
int get_vec_element(vec_ptr v, int
  index, int *dest)
```

- Retrieve vector element, store at *dest
- Return 0 if out of bounds, 1 if successful

```
int *get_vec_start(vec_ptr v)
```

- Return pointer to start of vector data

Optimization Example

- Procedure
  - Compute sum of all elements of vector
  - Store result at destination location
  - What’s the Big-O of this code?

```
void combine1(vec_ptr v, int *dest){
  int i;
  *dest = 0;
  for (i=0; i<vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

Move `vec_length` Call Out of Loop

- Value does not change from one iteration to next
- Code motion, `vec_length` requires only constant time, but significant overhead

```
void combine2(vec_ptr v, int *dest){
   int i;
   int length = vec_length(v);
   *dest = 0;
   for (i = 0; i < length; i++) {
      int val;
      get_vec_element(v, i, &val);
      *dest += val;
   }
}
```

Reduction in Strength

```
void combine2(vec_ptr v, int *dest){
 int length = vec_length(v);
 *dest = 0;
  for (i = 0; i < length; i++) {
     int val;
     get_vec_element(v, i, &val);
     *dest += val;
  }
}
```

Reduction in Strength

```
void combine3(vec_ptr v, int *dest){
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

Eliminate Unneeded Memory Refs

```
void combine4(vec_ptr v, int *dest){
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

Code Motion Example #2

- Procedure to Convert String to Lowercase
  - Extracted from many beginners' C programs
  - (Note: only works for ASCII, not extended characters)

```
void toLower(char *s) {
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i]>='A' && s[i]<='Z')
      s[i] -= ('A' - 'a');
}
```

Optimization Blocker: Procedure Calls

- *Why couldn't the compiler move* `vec_len` *or* `strlen` *out of the inner loop?*
  - Procedure might have side effects
    - Alters global state each time called
  - Function might not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`

Optimization Blocker: Procedure Calls

- *Why doesn't compiler look at code for* `vec_len` *or* `strlen`*?*
  - Linker may overload with different version
    - Unless declared static
  - Interprocedural optimization is not extensively used, due to cost
- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations in and around them