Optimization of C Programs

C Programming and Software Tools

N.C. State Department of Computer Science

with material from R. Bryant and D. O'Halloran "Computer Systems: A Programmer's Perspective" and Jon Louis Bentley "Writing Efficient Programs"

Computer Science
NC STATE UNIVERSITY

Optimization

- Performance depends on...
  1. algorithm / data structure choices
  2. coding style
  3. compiler + options
  4. programming language (C is a good choice ☺)

Compilers

- Most compilers offer a variety of optimization choices
- **`gcc`**: **`-O`** or **`-O1`** or **`-O2`** or **`-O3`** (in order of increasing optimization)
- How much can you expect this to help?
- Does it ever hurt?

Compilers... (cont'd)

All the gcc choices(!) :

-falign-functions=*n* -falign-jumps=*n* -falign-labels=*n* -falign-loops=*n* -fbounds-check -fmudflap -fmudflapth -fmudflapir -fbranch-probabilities -fprofile-values -fvpt -fbranch-target-load-optimize -fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves -fcprop-registers -fcse-follow-jumps -fcse-skip-blocks -fcx-limited-range -fdata-sections -fdelayed-branch -fdelete-null-pointer-checks -fearly-inlining -fexpensive-optimizations -ffast-math -ffloat-store -fforce-addr -ffunction-sections -fgcse -fgcse-lm -fgcse-sm -fgcse-las -fgcse-after-reload -floop-optimize -fcrossjumping -fif-conversion -fif-conversion2 -finline-functions -finline-functions-called-once -finline-limit=*n* -fkeep-inline-functions -fkeep-static-consts -fmerge-constants -fmerge-all-constants -fmodulo-sched -fno-branch-count-reg -fno-default-inline -fno-defer-pop -floop-optimize2 -fmove-loop-invariants -fno-function-cse -fno-guess-branch-probability -fno-inline -fno-math-errno -fno-peephole -fno-peephole2 -funsafe-math-optimizations -funsafe-loop-optimizations -ffinite-math-only -fno-trapping-math -fno-zero-initialized-in-bss -fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls -fprefetch-loop-arrays -fprofile-generate -fprofile-use -fregmove -frename-registers -freorder-blocks -freorder-blocks-and-partition -freorder-functions -frerun-cse-after-loop -frerun-loop-opt -frounding-math -fschedule-insns -fschedule-insns2 -fno-sched-interblock -fno-sched-spec -fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns=*n* -fsched-stalled-insns-dep=*n* -fsched2-use-superblocks -fsched2-use-traces -freschedule-modulo-scheduled-loops -fsignaling-nans -fsingle-precision-constant -fstack-protector -fstack-protector-all -fstrength-reduce -fstrict-aliasing -ftracer -fthread-jumps -funroll-all-loops -funroll-loops -fpeel-loops -fsplit-ivs-in-unroller -funswitch-loops -fvariable-expansion-in-unroller -ftree-pre -ftree-ccp -ftree-dce -ftree-loop-optimize -ftree-loop-linear -ftree-loop-im -ftree-loop-ivcanon -fivopts -ftree-dominator-opts -ftree-dse -ftree-copyrename -ftree-sink -ftree-ch -ftree-sra -ftree-ter -ftree-lrs -ftree-fre -ftree-vectorize -ftree-vect-loop-version -ftree-salias -fweb -ftree-copy-prop -ftree-store-ccp -ftree-store-copy-prop -fwhole-program --param *name*=*value* -O -O0 -O1 -O2 -O3 -Os

Limitations on Optimizing

- Must not change program outputs or results
- May increase code length
- May decrease code readability
- C features that complicate optimization...
  - pointers
  - functions with side-effects

Code Profiling

- To speed up a program, you have to know where it spends the most time
- To measure execution time, use **`time`** utility

  `time ./program [command line args]`

- **`gprof`** : a tool for profiling program execution
  - counts number of times each function is called
  - + how much time spent in each function
  - Time values only useful for relative, not absolute, performance measurement

...Profiling (cont'd)

- To add cycle counting to your program, compile with **`-pg`** flag, e.g.,

```
gcc -pg pgm.c -o pgm
```

- When you run **`pgm`**, it produces normal output, but also generates a file called **`gmon.out`**
- Execute **`gprof`** after running the program, , e.g.,

```
gprof ./pgm
```

gprof Example

```
  %    cumulative    self
time      seconds  seconds    calls
-----    -------- --------   -------
86.60        8.21    8.21          1  sort_words
 5.80        8.76    0.55     946596  lower1
 4.75        9.21    0.45     946596  find_ele_rec
 1.27        9.33    0.12     946596  h_add
```

- Shows number of calls and cumulative time for each function
- Where would you try to optimize the above program?

Code Motion

- move an expression evaluation outside of a loop (i.e., execute it fewer times)

Example

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

Before optimization

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

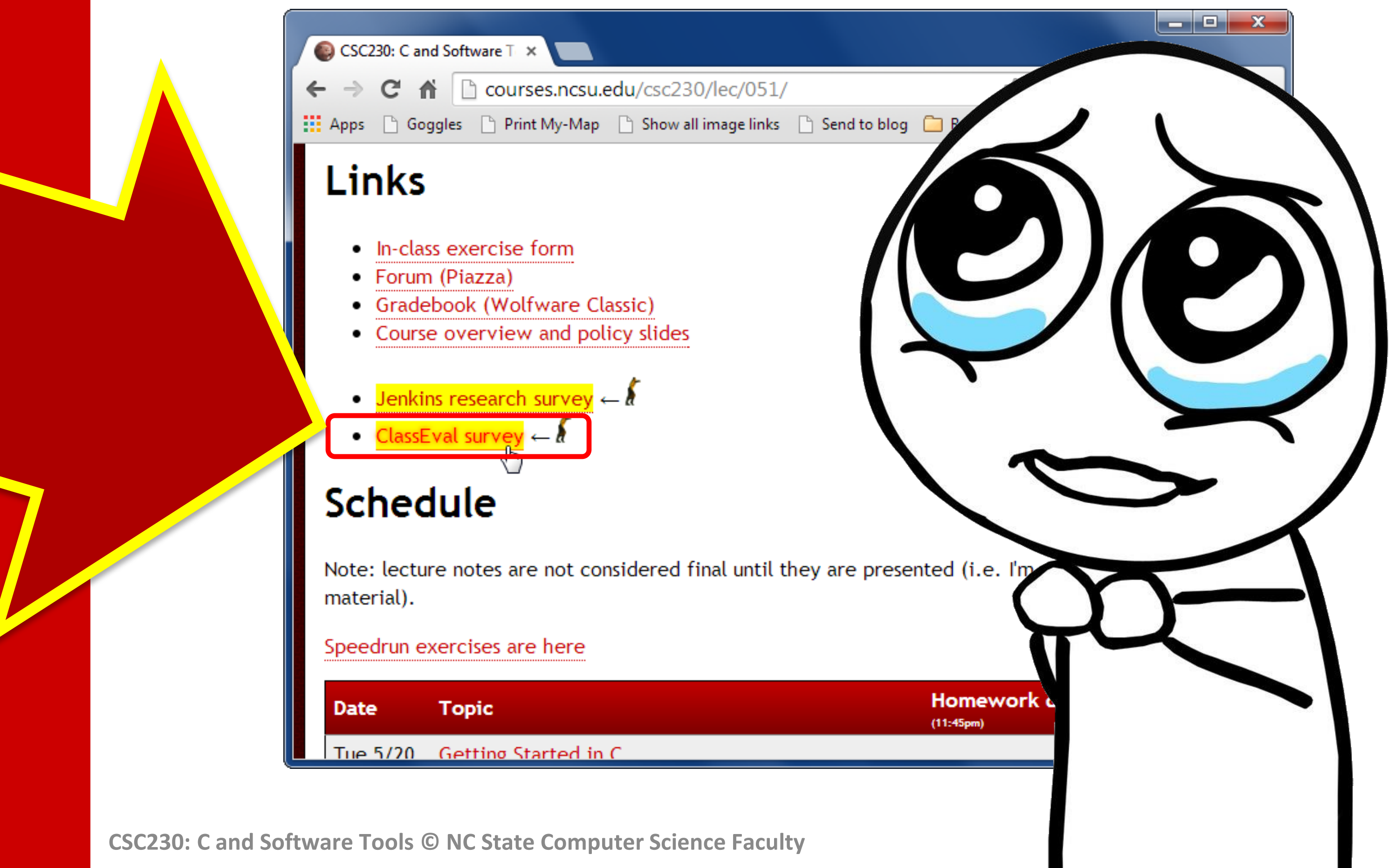
After optimization

Optimization?

```
/* 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;
```

DRAMATIC PAUSE

Please fill out the course survey, linked on the course webpage



Computer Science NC STATE UNIVERSITY

Share (Reuse) Expression Results

- "Compute once, use twice", ex.:

```
/* 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 different multiplications: `i*n, (i–1)*n, (i+1)*n`

Before optimization

1 multiplication: `i*n`

```
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;
```

After optimization

Inlining Function Calls

- Replace a function call with equivalent inline

```
int
prod(int i, int j, int n, int b[n][n], int c[n][n])
{
  int sum = 0;
  for (k = 0; k < n; k++)
    sum += b[i][k] * c[k][j];
  return sum;
}


…
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[i][j] = prod(i, j, n, b, c);
```

Before optimization

M-I Optimization (cont'd)

After optimization

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    sum = 0;
    for (k = 0; k < n; k++)
      sum += b[i][k] * c[k][j];
    a[i][j] = sum;
  }
```

Reordering Tests

- Place frequent **`case`** labels or **`if`** conditions first
  - reduces the average number of comparisons

```
if (height > 84)    /* extremely rare */
    f1();
else if (height > 72)     /* uncommon */
    f2();
else                  /* usually the case */
    f3();
```

Before optimization

Reordering Tests… (cont'd)

After optimization

```
if (height <= 72) /* usually the case */
    f3();
else if (height <= 84)    /* uncommon */
    f2();
else                      /* extremely rare */
    f1();
```

Pass Large Parameters by Reference

- Avoid passing large structs as arguments to functions.

```
struct mystruct {
    … many members, incl. array(s)…
} bigstruct;
…
int r = f(bigstruct);
…
int f(struct mystruct bigstruct) {
…
}
```

Before optimization

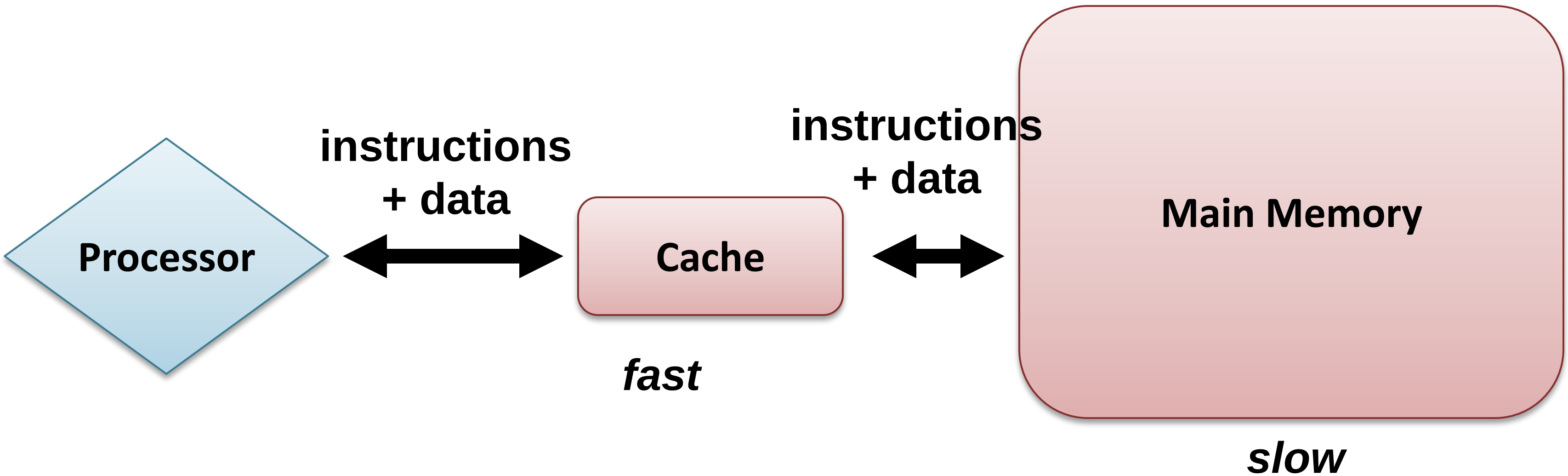
Pass Large … (cont'd)

After optimization

```
…
int r = f(&bigstruct);
…
int f(const struct mystruct *sp) {
…
}
```

Cache Optimization

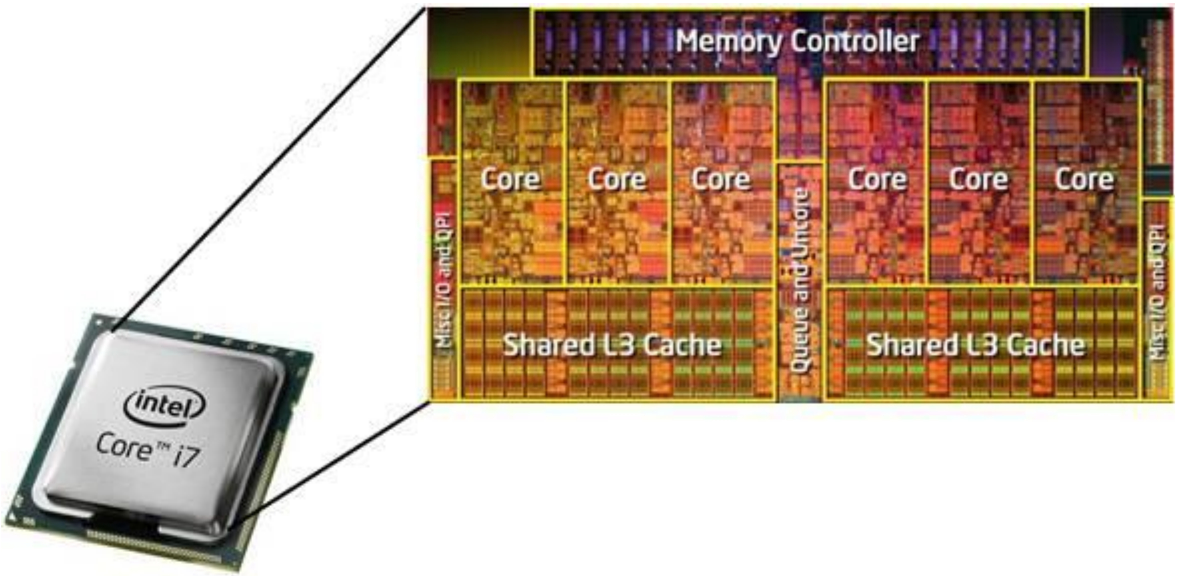
- Caching speeds up memory access
  - store in the (small, expensive) cache the data/instructions that are accessed most frequently



The program design / data layout can improve cache performance substantially in some cases

Multi-Core

- Getting optimal performance from multi-core processors also requires careful attention to coding
  - current tools don't help that much



Recommendations from GNOME Project

- “If you want to optimize your program, the first step is to profile the program running with real life data and collect profiling information.”
- “Do not write code that is hard to read and maintain if it is only to make the code faster.”

Bentley's Fundamental Rules for Optimization

- Code Simplification
  - Fast programs are typically simple programs
- Problem Simplification
  - Example: simplify loop by moving some work outside of the loop
- Relentless Suspicion
  - Question every part of the data structure and algorithm bottleneck areas
- Early Binding
  - Do some work as early as possible and only once

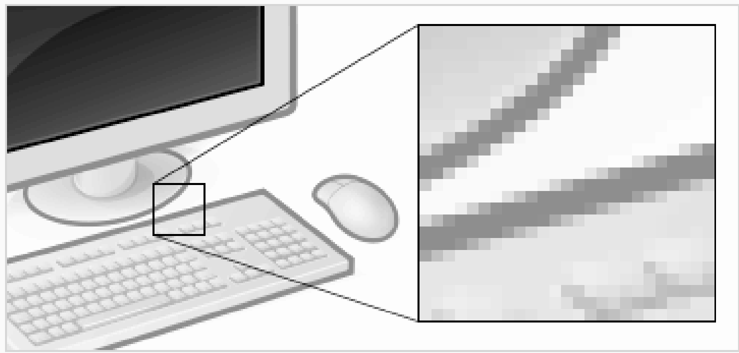
Test!!!

- Optimizations should NEVER change functionality
  - Test your program to ensure no regression in behavior!!!
  - Test after each optimization

EXAMPLE (TIME PERMITTING)

An Exercise

- Test case: an image processing program
- Digital images are composed of pixels
  - each is an integer value, representing brightness
  - 0 = black, 255 = white (grayscale picture)



This example shows an image with a portion greatly enlarged, in which the individual pixels are rendered as little squares and can easily be seen.

- How many pixels in an image?

Exercise… (cont'd)

- Image filtering: blurring, edge detection, ...

example of smoothing (blurring)

- How is (FIR) filtering done?
  - image convolution with a kernel

I_{11}	I_{12}	I_{13}	I_{14}	I_{15}	I_{16}	I_{17}	I_{18}	I_{19}
I_{21}	I_{22}	I_{23}	I_{24}	I_{25}	I_{26}	I_{27}	I_{28}	I_{29}
I_{31}	I_{32}	I_{33}	I_{34}	I_{35}	I_{36}	I_{37}	I_{38}	I_{39}
I_{41}	I_{42}	I_{43}	I_{44}	I_{45}	I_{46}	I_{47}	I_{48}	I_{49}
I_{51}	I_{52}	I_{53}	I_{54}	I_{55}	I_{56}	I_{57}	I_{58}	I_{59}
I_{61}	I_{62}	I_{63}	I_{64}	I_{65}	I_{66}	I_{67}	I_{68}	I_{69}

K_{11}	K_{12}	K_{13}
K_{21}	K_{22}	K_{23}

Exercise … (cont'd)

- Quadruply-nested loop!

```
for each row i of "old" image {
   for each column j of "old" image {
      newpix[i][j] = 0;
      for (k = -n/2; k < n/2; k++)
         for (l = -m/2; l < m/2; l++)
            newpix[i][j] +=
                oldpix[i+k][j+l] *
                kern[k][j];
   }
}
```

Outputs?

original

smoothed

sobel edge filter

Optimizations

1. (Base version)
2. Swap inner and outer loops, better caching
3. Use code motion (pointer arithmetic)
4. Skip processing of boundaries of image
5. Exploit distributivity of multiplication over addition, and specific kernel values
6. fwrite row of pixels instead of putc each pixel
7. Streamline reading of image, less pointer arithmetic
8. Use –O3 optimization in gcc

```
real    0m4.502s
user    0m4.401s
sys     0m0.079s
```

→

```
real    0m1.450s
user    0m0.354s
sys     0m0.119s
```