Announcements

- Program #6
  - is due in a week from Thursday
- Reading
  - Notes

Improving Code Performance

- This lecture is not about algorithms, but rather
  - how to convert algorithms into efficient code
  - how to refine code to make it run faster
- Helpful to know a bit about
  - what a compiler can and can't do
  - what takes time on this specific hardware
- Key Ideas
  - Be systematic and data driven in what you do
    - Easy to get into making random code changes
  - Programs spend their time in
    - loops (or recursion)
      - a sequence of code executed once is fast
    - doing I/O

Know Your Compiler

- Compilers can be asked to "optimize" your code
  - -O flag (and -On) enables the optimizer
  - they are supposed to never break your code
    - only safe changes to the program are permitted
- Limits on compiler optimizations
  - should never alter correct programs
    - may discover latent bugs though
  - limited understanding of the program
    - you are much smarter than the best compiler
  - need to compile program quickly

Understanding Modern Processors

- Pipelined
  - parts of multiple instructions running at once
  - decode instruction, load from memory
- Superscalar processors
  - Can execute 2 or more instructions at once
- Branch prediction
  - Computer guess which way a branch will go
  - Allows pipeline to stay full
- Cache memory
  - keep copies of recently accessed memory in fast storage
    - recently accessed memory is much faster
- Floating point takes much longer than integer
  - typically 20-30x for the same operation

Sample Compiler Optimization

- Original Code

```
int i, j, k;
….
for (i=0; i < 200; i++) {
    a[2*j+i] = j * k + i;
}
```

- Optimized Code

```
int i, j, k;
….
int temp1 temp2;
temp1= 2 * j;
temp2 = j * k;
for (i=0; i < 200; i++) {
    a[temp1+i] = temp2 + i;
}
```

Typical Types of Compiler Optimizations

- Code motion
  - previous example
- Loop Unrolling
  - often faster if a loop body is a bit bigger
    - modern computers can do several instructions at once
  - convert:

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

  - to:

```
for (i=0; i < n; i+=2) {
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] +b[i+1];
}
```

- Dead-Code Elimination
  - if compiler can infer code is never run, remove it

Limits to Compiler Optimization

- Pointer aliases

```
void func1(int *xp, int *yp) {
    *xp += *yp;
    *xp += *yp;
    }
```

  - problem if xp and yp point to same location
- Function Side effects
  - given:
    - return f(x) + f(x) + f(x) + f(x);
  - can't simplify to
    - return 4 * f(x);
- Some transformations can only be made by programmer

Common Code Changes - Loop Bound Checks

- Consider

```
for (i=0; i < strlen(data); i++) {
    if (data[i] > 'A' && data[i] < 'Z') data[i] -= ('A' - 'a');
}
```

- strlen is called each time, but doesn't change in loop
  - results in $O(N^2)$ performance
- Change to

```
int limit = strlen(data);
for (i=0; i < limit; i++) {
    if (data[i] > 'A' && data[i] < 'Z') data[i] -= ('A' - 'a');
}
```

- Compiler could not figure this one out

Common Code Changes - Use Pointers

- Recall

```
int limit = strlen(data);
for (i=0; i < limit; i++) {
    if (data[i] > 'A' && data[i] < 'Z') data[i] -= ('A' - 'a');
}
```

- With a function call

```
int limit = strlen(data);
for (i=0; i < limit; i++) {
     makelower(&data[i]);
}
```

- Can eliminate strlen call

```
for (ptr = data; *ptr; ++ptr)  {
    if (*ptr > 'A' && *ptr < 'Z') *ptr -= ('A' - 'a');
}
```

- Can move one subtraction outside

```
int diff = ‘A’ –’a’;
for (ptr = data; *ptr; ++ptr)  {
    if (*ptr > 'A' && *ptr < 'Z') *ptr -= diff;
}
```

Common Code Transformation - Don't write small functions

- Functions calls take time
  - Recall last week's code to setup calls stacks
- 1-2 line functions
  - take more time for the call than work
  - provide relatively little isolation of ideas
- Solution:
  - Define abstractions so work is larger than a few lines

Approach To Applying These Techniques

- For a big program
  - Hard to know where to start to optimize
- Start with gprof data
  - concentrate in parts of the program that take the most time
- Ahmdals Law
  - if tuning a function that takes α share of the time
  - and make it k times faster
  - $T_{new} = (1 - \alpha) T_{old} + (\alpha T_{old})/k$
  - Basically are limited by how big α is