Applied Algorithms Lec 7: Mini-Midterm and Optimization/Code Review

Sam McCauley

October 21, 2021

Williams College

Admin

- Assignment 1 graded
- Assignment 2 almost done!
- One of the harder assignments in the course
- Valgrind is your friend! Use it if you have seg faults, or unusual errors, or just to double-check that your program doesn't have memory issues
  - `make clean; make debug`
  - `valgrind ./test.out testData.txt timeData.txt`
- More than half the advice I've given in office hours this week was "let's run valgrind and see what it says"

Mini-Midterm Reminders

- Released right after class
- No collaboration!
- Deadline is firm unless in exceptional cases
    - Please do still email me if something comes up! But generally looking for exceptional circumstances that would normally delay a midterm. (Serious illness, personal emergency, etc.)
- No automated testing or leaderboard
- No TA office hours
- I'll hold office hours, but really just to help with basic structural things—no hints

Brief Assignment 2 discussion

- Any remaining questions about Assignment 2?

Mini Midterm 1: External Memory 3 SUM

Optimization (And Assignment 1 Review)

Plan for this topic

- First, talk about how various techniques can make code more efficient
    - ...or less efficient
- Focus on loops, and on compiler options
- Then, look back a bit at Assignment 1. Talk about various strategies, and what some final products looked like
- Also review the problem set questions

Taking out expensive operations

```
for(int i = 0; i < strlen(str1); i++){
   str1[i] = 'a';
}
```

- What's wrong with this code? How long does it take?
- Does the compiler optimize this out?

- It can't: we're changing the array, which could change its length. (Of course, we know that we're never setting any values to 0, but the compiler doesn't check for that.)

More subtle issues

```
int len = strlen(str1);
for(int i=0; i < len; i++){
   str1[i] = str1[0];
}
```

```
int len = strlen(str1);
int start = str1[0];
for(int i=0; i < len; i++){
   str1[i] = start;
}
```

- Version on the right runs 2-3x faster (on TCL 312 machines, compiled with gcc) even with optimizations on
- Why is that?
- Don't need to look up value! (Compiler doesn't know it doesn't change after the first iteration)

Theme of user optimizations vs compiler optimizations

- The compiler will do the best optimizations it can that work for *all* code
- Bear in mind: only common optimizations are implemented
- Opportunities for you: what do you know about your data, and about your methodology, that allows for further efficiency?

Loop Unrolling

- Classic technique to improve loop efficiency

- What are the costs of each iteration of a simple for loop?

```
for(int x = 0; x < 1000; x++){
   total += array[x];
}
```

Loop Unrolling

```
for(int x = 0; x < 1000; x++){
   total += array[x];
}
```

- Need to do a branch every loop
- Instruction pointer jump every loop (cost of "jumping back" varies; outside scope of course)
- Need to compare every loop
- Need to increment every loop

Unrolled Loop

```
for(int x = 0; x < 1000; x+=5){
   total += array[x];
   total += array[x+1];
   total += array[x+2];
   total += array[x+3];
   total += array[x+4];
}
```

- In short: repeat body of the loop multiple times. What does this gain us?

Unrolled Loop

```
for(int x = 0; x < 1000;
    x++){
   total += array[x];
}
```

- Branch every loop
- Instruction pointer jump every loop
- Compare every loop
- Increment every loop

```
for(int x = 0; x < 1000;
    x+=5){
   total += array[x];
   total += array[x+1];
   total += array[x+2];
   total += array[x+3];
   total += array[x+4];
}
```

- Branch every 5 loops
- Instruction pointer jump every 5 loops
- Compare every 5 loops
- Increment every 5 loops*

What did we need to know to make this substitution?

- Needed array size to be a multiple of 5

- If not: still (might) give speedup with checks between each substitution (number of branches and compares does not decrease in that case)

Disadvantages of Loop Unrolling?

- Seems like we break even at worst?
- Loop unrolling increases code size
- Can hurt performance if important parts of code no longer fit in cache
- Remember: fetching instructions can require cache misses!

Automatic loop unrolling?

- Why can't gcc unroll our loops?
- It can!
- Need to turn on specifically (not enabled at *any* optimization level)

`-funroll-loops`
Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. `-funroll-loops` implies `-frerun-cse-after-loop`. This option makes code larger, and may or may not make it run faster.

`-funroll-all-loops`
Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. `-funroll-all-loops` implies the same options as `-funroll-loops`,

- -O3 does a specific kind of unrolling of nested loops

Compiler optimizations?

- We've stumbled upon a classic (and thematic) problem in optimization: time vs space of the machine code itself

- Many optimizations of code reduce the number of operations (or their total time), but increase the size of the code itself—potentially leading to cache misses

Revisiting compiler flags

- `-O0`: No optimizations
- `-O1`: Some optimizations; may take longer to compile than `-O0`
- `-O2`: Turns on "nearly all" optimizations *that do not involve a space-time tradeoff*
- `-O3`: More optimizations. May lead to larger final programs
- `-Ofast`: Even more optimizations. Most notable is reordering floating point operations (can lead to correctness issues)

Optimizations and this course

- Our projects generally involve really small programs. This is why the very optimized versions tend to work well for your code.
- Not advised in general
- Example: Gentoo user manual. (Gentoo is a linux distribution in which *all* software is compiled from scratch. So this is advice for people compiling large software like the linux kernel, chromium, libreoffice, etc. (as well as, of course, very small utilities like git))

Gentoo optimization advice

- `-O1`: the most basic optimization level. The compiler will try to produce faster, smaller code without taking much compilation time. It is basic, but it should get the job done all the time.
- `-O2`: A step up from `-O1`. The *recommended* level of optimization unless the system has special needs. `-O2` will activate a few more flags in addition to the ones activated by `-O1`. With `-O2`, the compiler will attempt to increase code performance without compromising on size, and without taking too much compilation time. SSE or AVX may be utilized at this level but no YMM registers will be used unless `-ftree-vectorize` is also enabled.
- `-O3`: the highest level of optimization possible. It enables optimizations that are expensive in terms of compile time and memory usage. Compiling with `-O3` is not a guaranteed way to improve performance, and in fact, in many cases, can slow down a system due to larger binaries and increased memory usage. `-O3` is also known to break several packages. Using `-O3` is not recommended. However, it also enables `-ftree-vectorize` so that loops in the code get vectorized and will use AVX YMM registers.

One more common cost for time-space tradeoff

- We've talked about how costly it is to call a function
- Well, most of the time, we don't really *need* function calls at all, do we? If the function doesn't call another function, can just put the code for the function directly into the code
- Called *function inlining*
- Tradeoff?

Function Inlining

- Can do it yourself. May not be a good idea. (Makes code harder to read.)
- `gcc` will judge each function for you and inline it if `gcc` thinks it's a good idea (flag to get `gcc` to do this is `--finline-functions`; it is turned on with `-O2`)
- Can use `inline` keyword. `gcc` will try particularly hard to inline it for you, and if it can't will tell you if you have `-Winline` flag on
  - Can use `__inline__`; does the same thing. Some compilers may like this better
  - Probably want to always use `static inline`
- Can also use `__attribute__((always_inline))` which really forces it to inline even if optimizations are turned off

One more optimization flag

- `--march=native`
- tells `gcc` to use instructions specific to this processor. May increase speed
- Only disadvantage: your compiled binary may not run on other computers unless they have an identical processor

Looking Back at Assignment 1

Some comments

- Lots of great submissions!
- It seems that algorithmic improvements are more important than engineering improvements for Assignment 1
- (I believe the reverse is the case for Assignment 2)

Leaderboard at the end

Assignment1

Last Updated Sep 22 22:02

1	85cc	0.227501
2	Best Last Year	0.512625
3	590c	1.420481
4	d46e	2.071631
5	005c	2.505946
6	6977	2.99559
7	Sam	4.566402

- Most students were in the ballpark of 4-8 seconds

Where do our costs come from in Assignment 1?

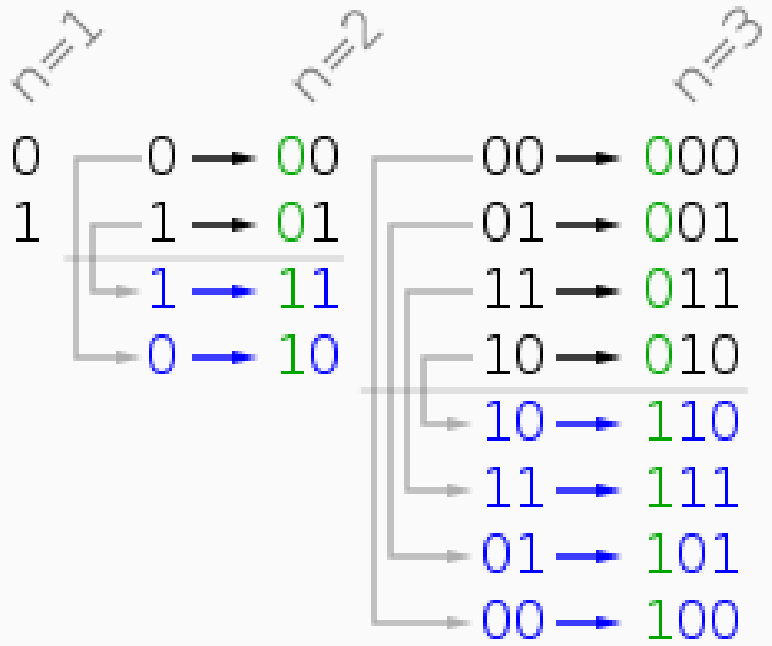
- Three $O(n2^{n/2})$ terms:
- Calculating the height of all subsets
- Sorting the table
- Performing a binary search for each first-half-subset

Let's improve all of these to $O(2^{n/2})$. (The fastest submission is a very clean implementation of all three of these.)

Calculating the height in constant time

- What's faster than calculating the height from scratch each time?
- Only adding on "new" items
- That's $O(1)$ on average, but it's a pain to implement
- Can we change our ordering to get improved performance?
- First idea: Gray codes

Gray codes



- (Named after Frank Gray)
- Reorder all subsets such that each differs by 1 bit
- Always possible; lots of clean implementations that can iterate through
- So far: doesn't do *too* much to make two towers faster

Calculating the height in constant time

- What's faster than calculating the height from scratch each time?
- Only adding on "new" items
- That's $O(1)$ on average, but it's a pain to implement
- Can we change our ordering to get improved performance?
- Next idea: fill in set *one item at a time*

Calculating the height in constant time

```
for(int64_t i = 0; i < f_half_length; i ++) {
        int64_t two_p_i = 1LL << i;
        for(int64_t j = 0; j < two_p_i; j ++){
                s2_heights[j + two_p_i].height = s2_heights[j].height + heights[i];
                s2_heights[j + two_p_i].mask = j + two_p_i;
        }
}
```

Sorting in linear time??

- Not possible in general, but our data has special structure

- Remember how we could more efficiently build up the heights. What would happen if we sorted the array at the same time?

Sorting in linear time (from best-performing solution)

```
for(int i=0; i<(inputSize-inputSize/2); i++){


        int64_t currID = s2t[i].id;
        double currH = sqrt(s2t[i].value);

        int maxq= pow(2,i);
        struct SubsetItem* newArr = (struct SubsetItem*)calloc(maxq,sizeof(struct SubsetItem));

        for(int q=0; q<maxq; q++){
                newArr[q].height=pArr[q].height+currH;
                newArr[q].key=pArr[q].key+currID;
        }

        sortedMerge(pArr,newArr,currPsize,maxq);
        currPsize+=maxq;
        free(newArr);
}
```

Binary search

- Really really costly
- Let's look at two attempts to engineer a more efficient binary search

Inlined, Unrolled binary search

```
 * Binary search for the predecessor in a sorted table
 * The loop is unrolled to enable slightly better performance
 */
__inline__ int unrolled_bin_search(entry* table, uint32_t high, double target) {
  uint32_t low = 0, mid;
  double mid_val;

  while (low < high) {
    mid = (low + high + 1) / 2;
    mid_val = table[mid].val;

    if (target > mid_val) {
      if (mid >= high) return mid;
      mid = (mid + high + 1) / 2;
      mid_val = table[mid].val;
      if (target > mid_val) {
        low = mid;
      } else if (target < mid_val) {
        high = mid - 1;
      } else {
        return mid;
      }

    } else if (target < mid_val) {
      if (low >= mid - 1) return low;
      mid = (low + mid) / 2;
      mid_val = table[mid].val;
      if (target > mid_val) {
        low = mid;
      } else if (target < mid_val) {
        high = mid - 1;
      } else {
        return mid;
      }
    } else {
      return mid;
    }
  }

  return low;
}
```

- From “best last year”
- Did much better than one would think—we’ll talk about why

Why is binary search really slow?

- Cache efficiency! Lots of cache misses
- Also branch mispredictions
- Can we avoid these?

Code that avoids these

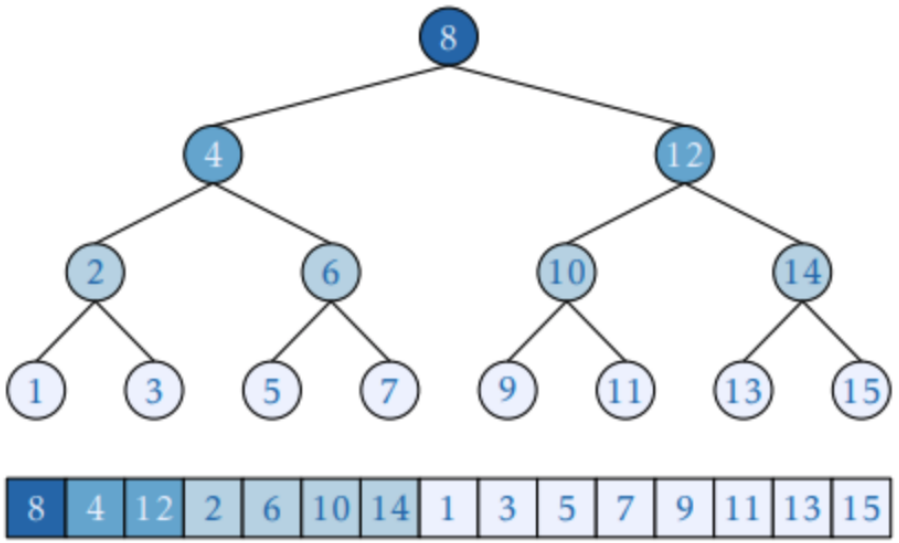
```
int64_t eytzinger(int64_t i, int64_t k, double* a, double* b) {
  if (k <= n) {
      i = eytzinger(i, 2 * k + 1,a,b);
      b[k] = a[i++];
      i = eytzinger(i, 2 * k,a,b);
  }
  return i;
}

__inline__ __attribute__((always_inline)) int64_t search(double x, double* b) {
    int64_t k = 1;
    while (k <= n) {
      __builtin_prefetch(b + k * block_size);
      k = 2 * k + (b[k] > x);
    }
    k >>= __builtin_ffs(~k);
    return k;
}
```

- What the heck?

Eytzinger layout

- Comment in code led to github that referenced a paper about a way to rearrange arrays to lead to better binary search performance

This is how this layout will look when applied to binary search:



Analysis

- Recall: normal binary search is $O(\log_2 N/B)$ cache misses
- What about with this layout?
- Answer: $O(\log_2 N/B)$ cache misses!
- What gives?
- By the way: there IS a layout that gives $O(\log_B N/B)$ cache misses. But it has issues with constants.

Analysis

- Let's say I'm at a given branch in my binary search. What can I say about where I'll be 2 branches from now?

- Idea: with this layout, we know where we're going ahead of time. While doing previous operations, can *prefetch* future cache accesses so that they're already available by the time we get there.

- Prefetching is very rare as an optimization technique. But this is a cool example of it

Getting rid of the binary search

- That said, even these highly optimized binary searches are costly. Can we avoid them entirely?

- Hint: the high cost of binary search is that we're jumping all over the table. Can we group entries so that we don't need to jump all over the table?

- Stronger hint: if I have two similar heights for the first half of my blocks, I'm going to be doing very similar searches in the second half...

Use two tables

- Idea: make a table for *both* halves of the input; sort each. $O(2^{n/2})$ time with the optimizations from before.
  - This does double our space usage!
- Now, can do a merge-like operation to determine, for each set in the first half, find the optimal set in the second half
- Same idea as 3SUM
- $O(2^{n/2})$ total time.
- Cache efficiency? $O(2^{n/2}/B)$.

(Pretty much) rest of best solution

```
int currPindex = pSize-1;
for(int i=0; i<qSize; i++){

        while(qArr[i].height+pArr[currPindex].height>totalHeight/2){
                currPindex--;
        }
        if(maxSubset.height<qArr[i].height+pArr[currPindex].height){
                maxSubset.height=qArr[i].height+pArr[currPindex].height;
                maxSubset.key=qArr[i].key+pArr[currPindex].key;
        }
}
```

Lessons

- Cache efficiency is king
- In this case, most optimizations depended on the problem itself. gcc can't help with that
    - I think Assignment 2 is more optimization-heavy once it fits in cache.
- Looks like I need to increase the input size a bit next time to make sure the final times are macroscopic

Problem Set Questions On Board