Lecture 11: Code Review and SIMD Instructions

Sam McCauley

October 21, 2021

Williams College

Admin

- Still working on grading
- Code/slides from today posted after class
- Questions?

Assignment 2 Optimizations

Assignment 2

- Lots of cool ideas!
- Some seem very nice but don't speed things up much.

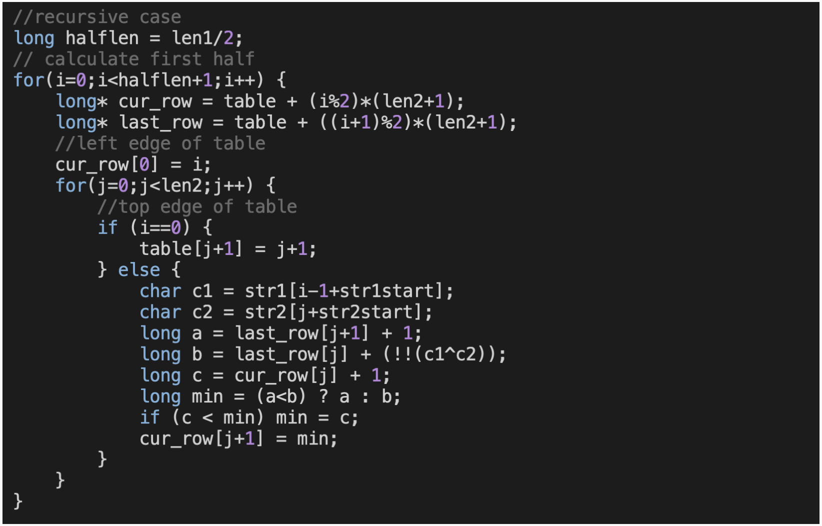
Assignment 2: Some ideas that seemed to work

- Large base case
    - Why is this good? (Hint: why was Hirshberg's a good idea in the first place?)
    - 300, 2048 both used
    - Both small enough that `len1 * len2` fits in cache
- Iterative version!
    - Recursive calls have overhead; can skip them
    - To be honest this seems like it should be a lower-order term to me
- Don't reverse strings?
    - Just doing the DP backwards might be faster(?)

Cool ideas that didn't speed things up much

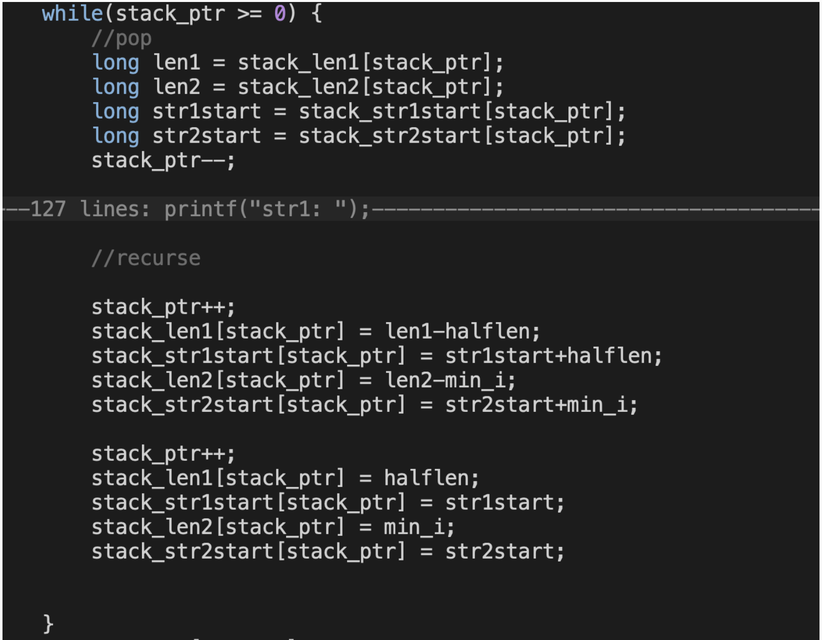
- Using 1 array instead of 2 when calculating the edit distance value
- Getting fancy with calculating the min
    - Two if statements seems fastest
    - I think: compiler is really good at optimizing that.
- Easiest way of passing around solution seemed to be continuously appending each piece as it's found
- Idea: the way the recursive calls work, the solution for the second half is always after the solution for the first half

Code from best last year (calculating costs)



```
//recursive case
long halflen = len1/2;
// calculate first half
for(i=0;i<halflen+1;i++) {
    long* cur_row = table + (i%2)*(len2+1);
    long* last_row = table + ((i+1)%2)*(len2+1);
    //left edge of table
    cur_row[0] = i;
    for(j=0;j<len2;j++) {
        //top edge of table
        if (i==0) {
            table[j+1] = j+1;
        } else {
            char c1 = str1[i-1+str1start];
            char c2 = str2[j+str2start];
            long a = last_row[j+1] + 1;
            long b = last_row[j] + (!!(c1^c2));
            long c = cur_row[j] + 1;
            long min = (a<b) ? a : b;
            if (c < min) min = c;
            cur_row[j+1] = min;
        }
    }
}
```

Code from best last year (iterative Hirschberg's)



```
while(stack_ptr >= 0) {
    //pop
    long len1 = stack_len1[stack_ptr];
    long len2 = stack_len2[stack_ptr];
    long str1start = stack_str1start[stack_ptr];
    long str2start = stack_str2start[stack_ptr];
    stack_ptr--;

---127 lines: printf("str1: ");-----------------------------------

    //recurse

    stack_ptr++;
    stack_len1[stack_ptr] = len1-halflen;
    stack_str1start[stack_ptr] = str1start+halflen;
    stack_len2[stack_ptr] = len2-min_i;
    stack_str2start[stack_ptr] = str2start+min_i;

    stack_ptr++;
    stack_len1[stack_ptr] = halflen;
    stack_str1start[stack_ptr] = str1start;
    stack_len2[stack_ptr] = min_i;
    stack_str2start[stack_ptr] = str2start;


}
```

A note on memory usage

- Efficient memory usage depends on *reusing* memory
- (Like the space-efficient ED DP just reused two arrays)
- If using malloc/free to reuse space: need to be careful how you free!
- Making Hirshberg's space-efficient was not meant to be a gotcha question. I'm going to go back and make sure my comments are correct and give back some points.
- Let's go over how this works together.

Hirshberg's

```
hirshbergs(char* userSolution, char* str1, int len1,
    char* str2, int len2){
   char* solArray = malloc(len1);
   hirshbergs(solArray, str1, len1/2, str2, minItem);
   hirshbergs(solArray, str1 + len1/2, len1 - len1/2,
       str2 + minItem, len2 - minItem);
   strcat(userSolution, solArray);
   free(solArray);
}
```

How much space does this solution require?

Answer: space allocated over whole function, plus max(space used during recursive calls but freed by the end), plus sum(space used during recursive calls and not freed)

Hirshberg's

```
hirshbergs(char* userSolution, char* str1, int len1,
    char* str2, int len2){
   char* solArray = malloc(len1 + len2);
   hirshbergs(solArray, str1, len1/2, str2, minItem);
   hirshbergs(solArray, str1 + len1/2, len1 - len1/2,
       str2 + minItem, len2 - minItem);
   strcat(userSolution, solArray);
   free(solArray);
```

Let x, y be size of strings in recursive call.

Allocates $O(x + y)$ space. Maximum space used (and freed) by any recursive call is $O(x/2 + y - 1)$. No space used and not freed by recursive call.

Level ℓ of the recursion uses $\Omega(n - \ell)$ space. Number of levels is $\log_2 m$. Total: $\Omega(n \log_2 m)$.

Hirshberg's

```
hirshbergs(char* userSolution, char* str1, int len1,
    char* str2, int len2){
   char* solArray = malloc(len1);
   hirshbergs(solArray, str1, len1/2, str2, minItem);
   hirshbergs(solArray, str1 + len1/2, len1 - len1/2,
       str2 + minItem, len2 - minItem);
   strcat(userSolution, solArray);
   free(solArray);
```

Allocates $O(x)$ space. (No, this is not enough to store a solution; this is just an example.) Maximum space used (and freed) by any recursive call is $O(x/2)$. No space used and not freed by recursive call.

Level ℓ of the recursion uses $\Omega(m/2^\ell)$ space. Total space: $\Omega(m)$.

Hirshberg's

```
hirshbergs(char* userSolution, char* str1, int len1,
    char* str2, int len2){
   char* solArray = malloc(len1);
   hirshbergs(solArray, str1, len1/2, str2, minItem);
   hirshbergs(solArray, str1 + len1/2, len1 - len1/2,
       str2 + minItem, len2 - minItem);
   strcat(userSolution, solArray);}
```

Allocates $O(x)$ space. $O(x/2)$ space used and not freed by each recursive call (for $O(x)$ total)

Level ℓ of the recursion uses $O(m)$ space. Total space: $O(m \log_2 m)$.

Hirshberg's

```
hirshbergs(char* tempSolution, char* str1, int len1,
    char* str2, int len2){
   char* solArray1, solArray2;
   hirshbergs(solArray, str1, len1/2, str2, minItem);
   hirshbergs(solArray2, str1 + len1/2, len1 - len1/2,
       str2 + minItem, len2 - minItem);
   tempSolution = malloc(strlen(solArray) +
       strlen(solArray2) + 1);
   tempSolution[0] = '\0';
   strcat(tempSolution, solArray);
   strcat(tempSolution, solArray2);
   free(solArray1);
   free(solArray2); }
```

Only allocate space for each solution character once. Space: $O(n+m)$.

Hirshberg's

```
hirshbergs(char* userSolution, char* str1, int len1,
    char* str2, int len2){
    //any space allocated to find minItem is freed by
        now
    //base case appends to userSolution directly
   hirshbergs(userSolution, str1, len1/2, str2,
       minItem);
   hirshbergs(userSolution, str1 + len1/2, len1 -
       len1/2, str2 + minItem, len2 - minItem);
}
```

Space: $O(n + m)$.

Assignment 2 Question Comments

Bookshelf Problem

- Idea: very similar setup to edit distance
- Need to be careful when gluing the two subproblems together
- In particular: unlike edit distance, need to be careful when reversing the problem
    - First item needs to be on first shelf—but last item doesn't need to be on last shelf

Hash Functions in Practice

What do we want out of a hash function?

Of course, we want consistency (each time we hash an item we get the same result back). What else might we want?

- Fast
- Low space requirements (i.e. may need to store a seed; don't want that to be too big)
- Good collision avoidance
- Bear in mind: different hashes work on different types of elements. We'll focus on integers and strings (especially strings)

Ideal Hash Functions (Independent, Uniform Hashing)

- Perfect collision avoidance
- But: require extremely large space usage unless universe of possible elements is extremely small
- You did use one of these...
    - For h on Assignment 3! Those values were all chosen independently, completely at random

Hashing in Java

- Anyone know how Java hashes a 64 bit `Long`?
- `return x ^ (x >> 32);`
- Advantages of this?

Is this good for:

- In cuckoo filter: h_1, h, f?
  - h_1 and f: might work if elements are fairly well-spread (we take mod)
  - h: probably won't work (output too small)
- CMS? HLL?
  - CMS might be OK; prob not (same as above)
  - HLL likely useless unless elements very uniformly spread

Multiply-Shift Hashing

```
uint64_t hash3(uint64_t value){
   return (uint64_t)(value * 0x765a3cc864bd9779) >> (64 -
       SHIFT);
}
```

- Hash from Mini-Midterm 1
- Seed is a large prime number to multiply by; can also add a large random prime
- Advantages?
    - Fast! (And easy.)

Multiply-Shift Hashing

```
uint64_t hash3(uint64_t value){
   return (uint64_t)(value * 0x765a3cc864bd9779) >> (64 -
       SHIFT);
}
```

- How good is it?
  - Pretty good! For any x, y, $\Pr(h(x) = h(y)) = 1/n$.
  - But unfortunately behavior doesn't extend to larger numbers of elements.
- Let's say we use this for a hash table with chaining (n items, n chains). What is the expected number of elements we find during a query q?
- $X_i = 1$ if $h(x_i) = h(q)$. Then $\mathrm{E}[X_i] = 1/n$. By linearity of expectation, total number of items is $\sum_{i=1}^{n} 1/n = 1$.

Multiply-Shift Hashing

```
uint64_t hash3(uint64_t value){
   return (uint64_t)(value * 0x765a3cc864bd9779) >> (64 -
      SHIFT);
}
```

- How good is it?
- What is the size of the largest chain?
  - Can't say anything given the above. (If assuming ideal hashes then we'd get $O(\log n)$–multiply-shift hashing doesn't guarantee this)
- This hash is fast and good on average, but no concentration bounds on worst-case data.

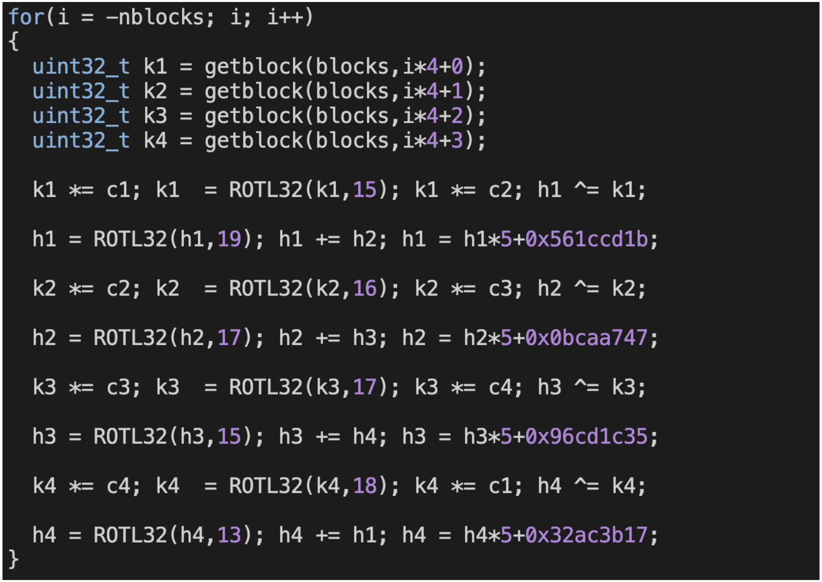
Multiply-Shift hashing for other data structures

- Is this going to work well for a filter?
  - Probably not. Would have to try it.
- Count-min sketch?
  - Average performance would work! But concentration bounds may not—the number of rows may not improve performance as much as we'd think
- Hyperloglog?
  - Would have to try but I doubt it.

Murmurhash

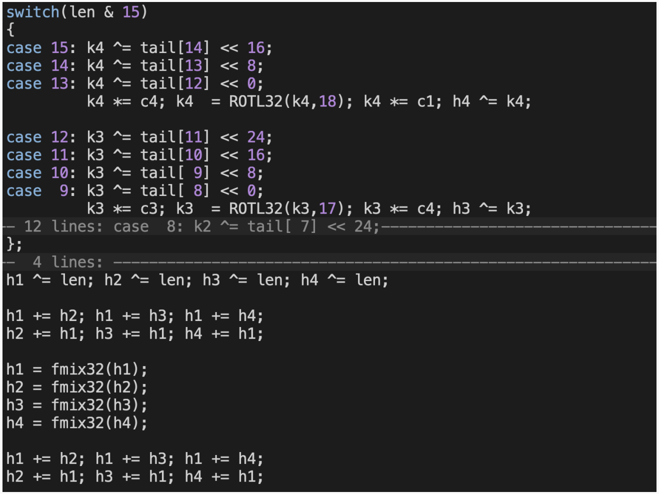
- Popular practical hash function
- Uses repeated MUltiply and Rotate operations
    - Rotate is like shift, but bits that “fall off” are replaced on other side
    - Can be implemented with two shifts and an OR
- Code isn’t exactly short; 50 operations to hash a number

Murmurhash Code



```
for(i = -nblocks; i; i++)
{
  uint32_t k1 = getblock(blocks,i*4+0);
  uint32_t k2 = getblock(blocks,i*4+1);
  uint32_t k3 = getblock(blocks,i*4+2);
  uint32_t k4 = getblock(blocks,i*4+3);

  k1 *= c1; k1  = ROTL32(k1,15); k1 *= c2; h1 ^= k1;

  h1 = ROTL32(h1,19); h1 += h2; h1 = h1*5+0x561ccd1b;

  k2 *= c2; k2  = ROTL32(k2,16); k2 *= c3; h2 ^= k2;

  h2 = ROTL32(h2,17); h2 += h3; h2 = h2*5+0x0bcaa747;

  k3 *= c3; k3  = ROTL32(k3,17); k3 *= c4; h3 ^= k3;

  h3 = ROTL32(h3,15); h3 += h4; h3 = h3*5+0x96cd1c35;

  k4 *= c4; k4  = ROTL32(k4,18); k4 *= c1; h4 ^= k4;

  h4 = ROTL32(h4,13); h4 += h1; h4 = h4*5+0x32ac3b17;
}
```

Murmurhash Code



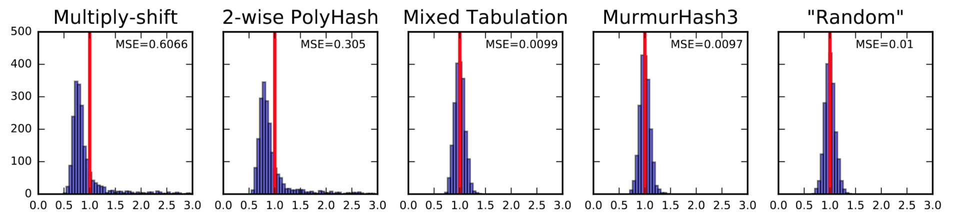
```
switch(len & 15)
{
case 15: k4 ^= tail[14] << 16;
case 14: k4 ^= tail[13] << 8;
case 13: k4 ^= tail[12] << 0;
         k4 *= c4; k4  = ROTL32(k4,18); k4 *= c1; h4 ^= k4;

case 12: k3 ^= tail[11] << 24;
case 11: k3 ^= tail[10] << 16;
case 10: k3 ^= tail[ 9] << 8;
case  9: k3 ^= tail[ 8] << 0;
         k3 *= c3; k3  = ROTL32(k3,17); k3 *= c4; h3 ^= k3;
-- 12 lines: case  8: k2 ^= tail[ 7] << 24;-------------------------------
};
--  4 lines: ----------------------------------------------------------------
h1 ^= len; h2 ^= len; h3 ^= len; h4 ^= len;

h1 += h2; h1 += h3; h1 += h4;
h2 += h1; h3 += h1; h4 += h1;

h1 = fmix32(h1);
h2 = fmix32(h2);
h3 = fmix32(h3);
h4 = fmix32(h4);

h1 += h2; h1 += h3; h1 += h4;
h2 += h1; h3 += h1; h4 += h1;
```

(The light grey lines skip pieces of code.)

Murmurhash3 Performance

- No known worst-case guarantees (not even $\Pr(h(x) = h(y)) = O(1/n)$)
- Someday may discover: might not work well in some circumstances (this is what happened to Murmurhash2)
    - "Will this flaw cause your program to fail? Probably not - what this means in real-world terms is that if your keys contain repeated 4-byte values AND they differ only in those repeated values AND the repetitions fall on a 4-byte boundary, then your keys will collide with a probability of about 1 in $2^{27.4}$ instead of 2^{32}. Due to the birthday paradox, you should have a better than 50% chance of finding a collision in a group of 13115 bad keys instead of 65536."
    - `https://sites.google.com/site/murmurhash/murmurhash2flaw`

Murmurhash3 Performance



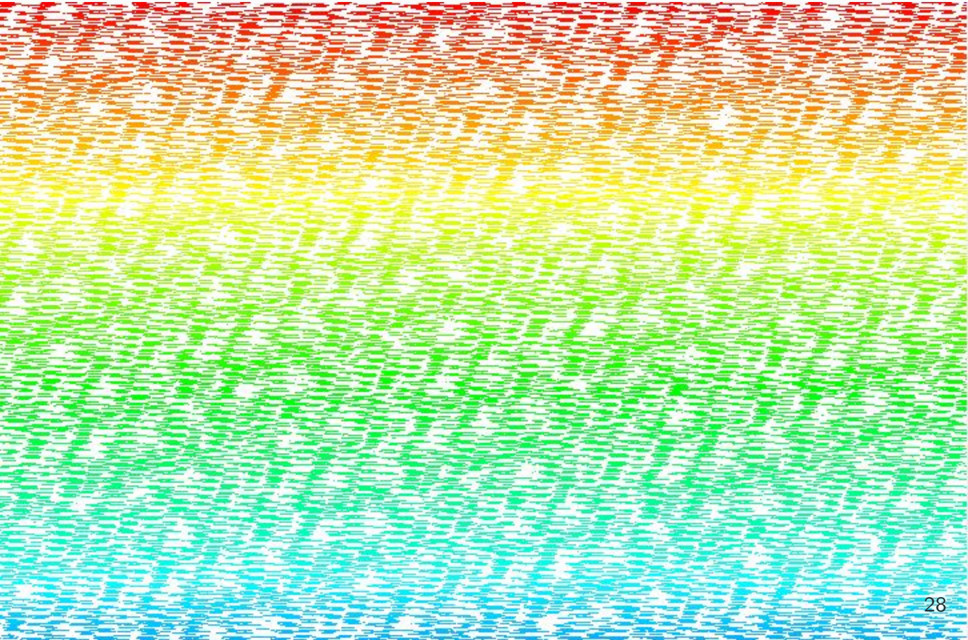
Average of square of bucket sizes. Data is an intentionally bad (albeit reasonable) case

From “Practical Hash Functions for Similarity Estimation and Dimensionality Reduction” by Dahlgaard, Knudsen, Thorup NeurIPS 2017

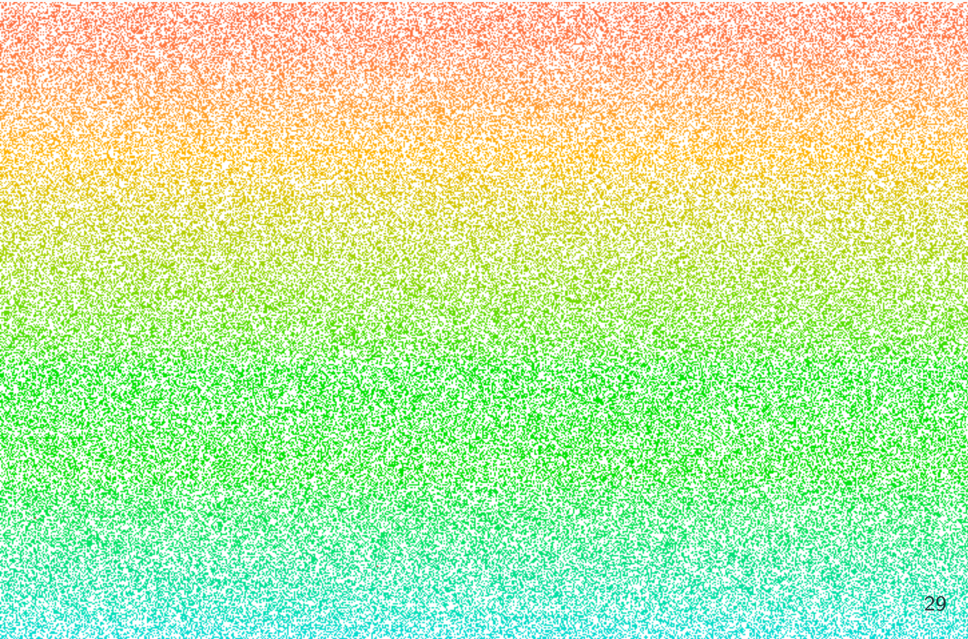
Murmurhash3 Performance in Practice

- Much more resilient than multiply-shift to more-difficult statistical tests (beyond average case)
- Visual example: let's say we hash "number strings": "1", "2", . . . "216553"
- (Cool experiment from `https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed`)
- (I wouldn't normally cite stackexchange but this is really cool)
- Compare SDBM (another popular hash) with Murmurhash2); fill in pixel if corresponding table entry is hashed to

SDBM (lots of chunks of full cells!)



Murmurhash2 (visually: random)



One last murmurhash question

- Murmurhash really just does a bunch of arbitrary multiplies and rotates
- Is there anything special about this specific sequence, or will any such set work pretty well?
- Answer: others might not work. Example: “SuperFastHash” also uses multiplies and rotates

Hash comparison

```
Hash           Lowercase      Random UUID  Numbers
=============  =============  ===========  =============
Murmur            145 ns      259 ns          92 ns
                    6 collis    5 collis       0 collis
SDBM              148 ns      484 ns          90 ns
                    4 collis    6 collis       0 collis
SuperFastHash     164 ns      344 ns         118 ns
                   85 collis    4 collis   18742 collis
```

SuperFastHash has bad performance on lowercase English words, and horrendous performance on numbers-as-strings.

(Also from `https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed`)

Unease with our options

- Murmurhash seems to do well (and is fast), but has few guarantees.
- What do we do if we're OK with a slightly slower hash, but we REALLY want to be sure it does well?
- Answer: cryptographic hashes! Secure even for cryptographic applications; no known statistical weaknesses
- Examples: SHA-3, BLAKE2, many others
- Broken: MD5, SHA-1, many others

Modern Instructions and Intrinsics

Main idea

- Processors aren't getting much faster
- So: modern processors comes with tools that help you do common computations more quickly
- Let's talk about a few of these tools specifically
- Note: need to use lab computers for access to many of these
    - Most processors have some kind of equivalent tools, but I can only guarantee for Intel
    - Need the right kind of processor

Count leading (trailing) zeroes

```
unsigned int v;
unsigned int c = 32;
v &= -signed(v);
if (v) c--;
if (v & 0x0000FFFF) c -= 16;
if (v & 0x00FF00FF) c -= 8;
if (v & 0x0F0F0F0F) c -= 4;
if (v & 0x33333333) c -= 2;
if (v & 0x55555555) c -= 1;
```

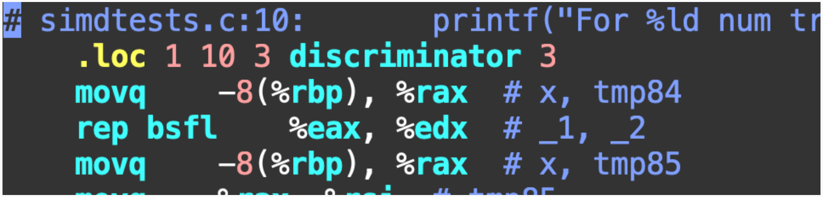
- Count the number of zeroes at the beginning (or end) of a number
- Can do using a few bit tricks
- But nowadays...single CPU operation (usually)

Telling `gcc` to use these operations

- Intrinsics!
- Library functions built into the compiler itself (`gcc` in our case)
- Usually: will use the best compiler option if it exists; will do a very high-quality subroutine if not
  - For example: their version of manually counting the trailing zeroes will almost definitely be faster than your `for` loop
  - (And even a version using bit tricks)
  - And you don't need to worry about debugging it!

Example intrinsic for counting zeroes

- `int __builtin_ctzl (unsigned long)`
- Note that you have to use a type like `unsigned long` (not `uint64_t`)
- If you want to count leading zeroes in an int, use `int __builtin_ctz (unsigned int x)`
- Let's look at some code using this



```
# simdtests.c:10:       printf("For %ld num tr
    .loc 1 10 3 discriminator 3
    movq    -8(%rbp), %rax  # x, tmp84
    rep bsfl    %eax, %edx  # _1, _2
    movq    -8(%rbp), %rax  # x, tmp85
```

Other intrinsics

- Lots and lots of them
- Get num 1s, get parity of num 1s, reverse the bytes in the word, raise number to power
- Not always going to have a CPU instruction

Compiler making decisions for you

- Generally you need to call these manually
- (The compiler doesn't know that you're calculating the number of trailing 0s; so it can't make that substitution)

SIMD instructions

Intro: a touch of parallelism

- Something we've only touched on briefly in this class: word-level parallelism
- Idea: we can do computations on 64 bit numbers very quickly (say 1 clock cycle)
- If our data is much less than 64 bits, can get extra computation done more quickly.

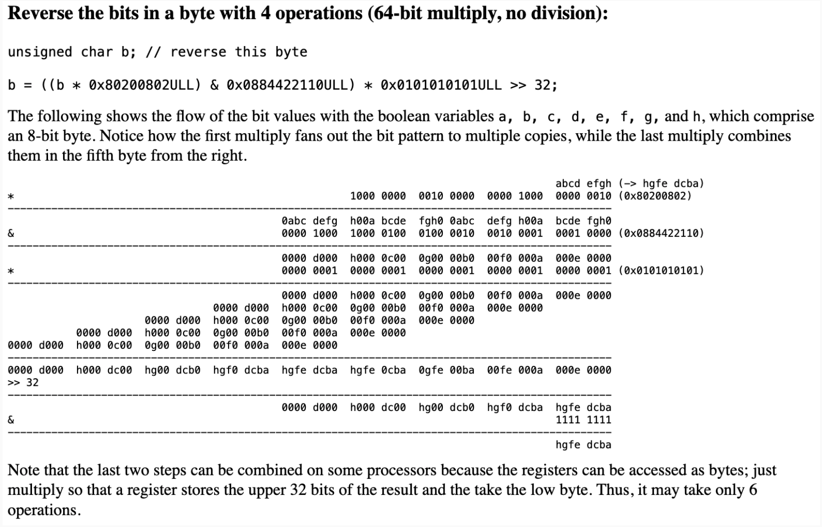
World level parallelism example

- Can you test if a string (array of `chars`) starts with “abcd” in $O(1)$ time?
    - Calculate a `uint64_t` corresponding to the correct integer
    - Cast the string pointer to a `uint64_t` pointer, then compare them
    - Watch for endianness!!!

In general...

- Words of 64 bits allow us to do lots of computations in one (or a few) clock cycles
- Example: taking the bitwise OR of two 64 bit numbers is basically doing 64 computations at once
- This is literally parallelism: the circuits in the chip do these operations simultaneously
- Hard to do things like add (almost impossible to multiply): why?
    - Carries (etc.) mess us up!
    - We'd have to leave "space" between pieces of data; lots of setup means it's probably not worth it

One last fun(?) example



Reverse the bits in a byte with 4 operations (64-bit multiply, no division):

```
unsigned char b; // reverse this byte

b = ((b * 0x80200802ULL) & 0x0884422110ULL) * 0x0101010101ULL >> 32;
```

The following shows the flow of the bit values with the boolean variables `a, b, c, d, e, f, g,` and `h`, which comprise an 8-bit byte. Notice how the first multiply fans out the bit pattern to multiple copies, while the last multiply combines them in the fifth byte from the right.

```
                                                                                        abcd efgh (-> hgfe dcba)
*                                                      1000 0000  0010 0000  0000 1000  0000 0010 (0x80200802)
-------------------------------------------------------------------------------------------------
                                            0abc defg  h00a bcde  fgh0 0abc  defg h00a  bcde fgh0
&                                           0000 1000  1000 0100  0100 0010  0010 0001  0001 0000 (0x0884422110)
-------------------------------------------------------------------------------------------------
                                            0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
*                                           0000 0001  0000 0001  0000 0001  0000 0001  0000 0001 (0x0101010101)
-------------------------------------------------------------------------------------------------
                                            0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
                                 0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
                      0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
           0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
-------------------------------------------------------------------------------------------------
0000 d000  h000 dc00  hg00 dcb0  hgf0 dcba  hgfe dcba  hgfe 0cba  0gfe 00ba  00fe 000a  000e 0000
>> 32
-------------------------------------------------------------------------------------------------
                                            0000 d000  h000 dc00  hg00 dcb0  hgf0 dcba  hgfe dcba
&                                                                                       1111 1111
-------------------------------------------------------------------------------------------------
                                                                                        hgfe dcba
```

Note that the last two steps can be combined on some processors because the registers can be accessed as bytes; just multiply so that a register stores the upper 32 bits of the result and the take the low byte. Thus, it may take only 6 operations.

`https://graphics.stanford.edu/~seander/bithacks.html#`

`ReverseByteWith64Bits`

Word-level paralellism

- Good part: takes advantage of how computers are built to speed up computation
- Bad parts?
    - Only works for a few operations (can't even really add)
    - Only works on really small pieces of data

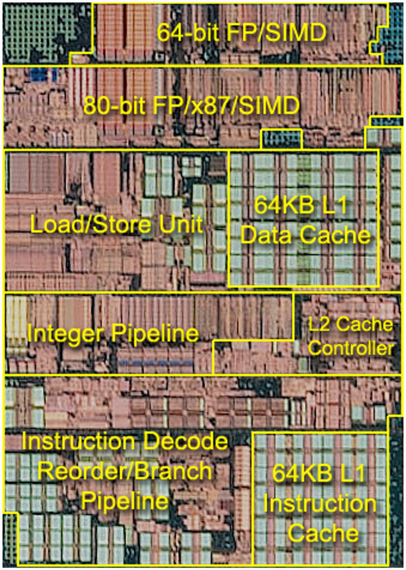
Extending it forward

- Having fast operations on 64 bit data can speed up operations on 1-bit or 8-bit data
- ...but often we want to operate on 32 or 64 bit data. It would be nice if we could do the same!
- Honestly it'd be nice if we could do something better like adding and multiplying rather than just taking OR or doing weird string comparisons...
- This is the purpose of SIMD!

SIMD

- SIMD: **S**ingle **I**nstruction **M**ultiple **D**ata
- A single instruction does an identical operation to multiple pieces of data
- Specialized circuits operate on each piece of data individually
- Can do bitwise operations, adding, multiplying, some others
- Also some operations to help load and read data
- Introduced on Intel processors in 1999, but fairly significantly expanded recently

Other Names



- Sometimes called “vector” instructions
- And/or referred to using instruction sets: SSE, AVX, AVX2, AVX-512 (these are extensions to x86).

SIMD Discussion

- Dipping our toes into parallelism
- Uniprocessor kind of parallelism
- GPU computation uses similar ideas
  - Scaled up significantly (much more speedup potential)
  - More restricted

SIMD on lab computers

```
(gdb) print $ymm0
$1 = {
 v8_float={0,0,0,0,0,0,0,0},
 v4_double={0,0,0,0},
 v32_int8={0 <repeats 32
     times>},
 v16_int16={0 <repeats 16
     times>},
 v8_int32={0,0,0,0,0,0,0,0},
 v4_int64={0,0,0,0},
 v2_int128={0,0}
}
```

- We have SSE, AVX, AVX2 instruction sets (don't have AVX-512)
- 16 "YMM" registers; each 256 bits
- (Older processors may only have 128 bit "XMM" registers.)
- Need to include `#include <immintrin.h>` and compile with `-mavx2`

SIMD Examples

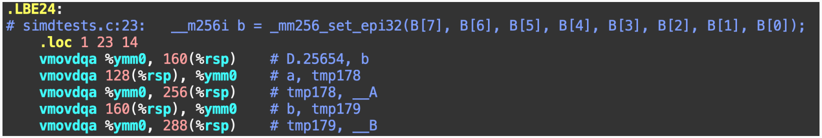
What is SIMD good for?

- Lots of identical operations on a set of elements; these operations are costly
- Elements are in nicely-sized chunks
    - Can always used specialized code to handle other cases

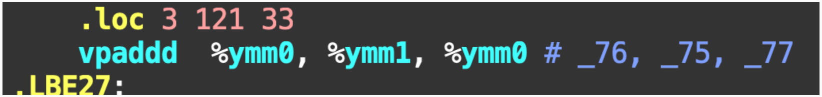
Example 1: Adding two arrays

- Let's add two arrays of 8 32-bit integers with one SIMD operation
- `simdtests.c`

Assembly examples



```
.LBE24:
# simdtests.c:23:   __m256i b = _mm256_set_epi32(B[7], B[6], B[5], B[4], B[3], B[2], B[1], B[0]);
    .loc 1 23 14
    vmovdqa %ymm0, 160(%rsp)    # D.25654, b
    vmovdqa 128(%rsp), %ymm0    # a, tmp178
    vmovdqa %ymm0, 256(%rsp)    # tmp178, __A
    vmovdqa 160(%rsp), %ymm0    # b, tmp179
    vmovdqa %ymm0, 288(%rsp)    # tmp179, __B
```



```
    .loc 3 121 33
    vpaddd  %ymm0, %ymm1, %ymm0 # _76, _75, _77
.LBE27:
```

Example 2: Adding single value to array

- Let's add one value (10) to an array.
- Do we need to declare a new array to do this?

Speed comparison

- How much time does SIMD add (in total in our implementation) take compared to normal add?

- It's a bit faster

Example 3: Searching for Particular Value in Array

- Can do vector comparisons, but get a 256 bit vector out
- Need a way to make that vector into something useful for us. Let's look at the code.
- `int _mm256_movemask_epi8(_mm256 arg)`: returns a 32 bit `int` where the *i*th bit of the `int` is the first bit in the *i*th byte of the argument `arg`

Optimization comparison?

- What happens when we change to O3?
- Everything gets faster!
- In previous tests: for adding, normally suddenly outpaces SIMD; finding the 0 element doesn't
- Guesses as to why? ...Let's take a look at the assembly
    - gcc is vectorizing the operations by itself and doing it very slightly better

SIMD Discussion

Tradeoffs

What are some downsides of using an SIMD instruction?

- SIMD instructions may be a little slower on a per-operation basis (folklore is a factor of ≈ 2 even for the operation itself, but it seems modern implementations are much better)
- Cost to gather items in new location
- SIMD is *not* always faster

How much can we save using SIMD? Let's say we're using 256 bit registers, and operating on 32 bit data.

- Factor of $256/32 = 8$ at absolute best
- Realistically is going to be quite a bit lower in practice

Tradeoffs

- Bear in mind Amdahl’s law when considering SIMD
- Only worth using on the most costly operations, and only when they work very well with SIMD

One Question

- What's a problem we've seen this semester that is particularly suited for SIMD speedup?
    - Hint: I'm not referring to any of the assignment problems

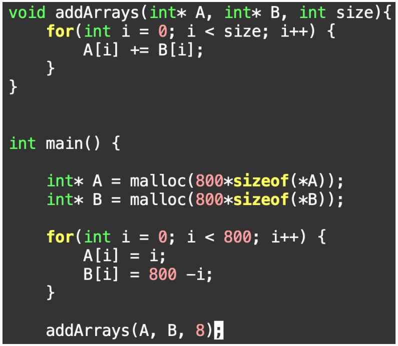
- Matrix multiplication: lots of time doing multiplications on successive matrix elements

- (SIMD works for some other problems too; I just wanted to highlight this as one of the classic examples.)

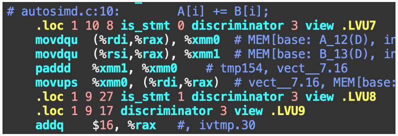
Compiler?

- A lot of the examples we saw were super simple
- Can the compiler use these operations automatically?
- As we just saw: yes it can
    - `--ftree-vectorize`
    - `--ftree-loop-vectorize` (turned on with `O3`)
    - Lots of extra option to tune `gcc` parameters for how it vectorizes
- But, as always, only is going to work in “obvious” situations.

Automatic Vectorization Example



```
void addArrays(int* A, int* B, int size){
    for(int i = 0; i < size; i++) {
        A[i] += B[i];
    }
}


int main() {

    int* A = malloc(800*sizeof(*A));
    int* B = malloc(800*sizeof(*B));

    for(int i = 0; i < 800; i++) {
        A[i] = i;
        B[i] = 800 -i;
    }

    addArrays(A, B, 8);
```



```
# autosimd.c:10:         A[i] += B[i];
    .loc 1 10 8 is_stmt 0 discriminator 3 view .LVU7
    movdqu  (%rdi,%rax), %xmm0  # MEM[base: A_12(D), in
    movdqu  (%rsi,%rax), %xmm1  # MEM[base: B_13(D), in
    paddd   %xmm1, %xmm0    # tmp154, vect__7.16
    movups  %xmm0, (%rdi,%rax)  # vect__7.16, MEM[base:
    .loc 1 9 27 is_stmt 1 discriminator 3 view .LVU8
    .loc 1 9 17 discriminator 3 view .LVU9
    addq    $16, %rax   #, ivtmp.30
```

We can see the `paddd` SIMD instruction (on `xmm1` and `xmm0`) when compiling with `-O3`.