Lecture 13: Code Review and SIMD Instructions 2

Sam McCauley

October 25, 2024

Williams College

Admin

- Apply to be a TA! (Especially for Algorithms). Due today
- Start Assignment 2!
- Remember that this is an assignment so you should do the work on your own/with class materials.
- Handout on the website to help you with Assignment 2!!!
- Homework 3/Assignment 1 back. Homework 4 soon
- Questions?

Assignment 1 Discussion

Problems 1–2

- Most people had the right intuition for this problem, but few people got it entirely right
- Let’s review the problems, then go over the solution on the board

Homework 3 Discussion

Homework 3

- Everyone did great! Good job.
- One trick for accessing slots I saw that makes life a little nicer
- Slots are 8 bits. So we can access them as 8-bit variables rather than using bit tricks
- (Similar to what we saw yesterday with casting a chunk of a string as an integer)

Homework 3 Excerpt

```
uint32_t* bin = filter->table + pos;
uint8_t* slot = (uint8_t*)(bin);
for (int i = 0; i < filter->binSize; ++i) {
  if( *(slot + i) == 0){
    *(slot + i) = fingerprint;
    return;
  }
}
```

Functions and Code Organization

- A lot of you did this well, but I want to emphasize
- The best way to write effective, understandable code is by organizing it
- One of the best ways to organize is via functions
- Let’s look at some example student code

Get and set in bin

```
int binGet(Filter* filter, uint64_t h, int binNum) {
    int bin = filter->table[h];
    int res = (bin >> (binNum * 8)) & ((1 << filter->fingerprintLength) - 1);

    // printf("binGet(filter, %ld, %d) = %x, bin = %x\n", h, binNum, res, bin);
    return res;
}

void binSet(Filter* filter, uint64_t h, int binNum, int f) {
    // printf("binSet(filter, %ld, %d, %d)\n", h, binNum, f);

    // resets bits binNum*8..binNum*8+7
    filter->table[h] &= (-1 - (((1 << filter->fingerprintLength) - 1) << (binNum * 8)))
;

    // adds the bits from f to binNum*8..binNum*8+7
    filter->table[h] |= f << (binNum * 8);
}
```

Bin Insert

```
// tries to insert into a bin; returns 0 if fails, 1 if succeeds
int binInsert(Filter* filter, uint64_t h, int f) {
    for (int i = 0; i < filter->binSize; i++) {
        if (!binGet(filter, h, i)) {
            binSet(filter, h, i, f);
            return 1;
        }
    }
    return 0;
}
```

Cuckoo

```c
void cuckoo(Filter* filter, uint64_t h, int f, int depth) {
    if (depth >= filter->maxIter) {
        printf("MAX ITER REACHED. CUCKOO HAS FAILED.\n");
        return;
    }

    int toCuckoo = filter->toCuckoo[h];
    filter->toCuckoo[h]++;
    int evictedF = binGet(filter, h, toCuckoo);
    uint64_t newBin = h ^ (hashFingerprint[evictedF - 1] % (filter->numBins - 1) + 1);

    binSet(filter, h, toCuckoo, f);

    if (!binInsert(filter, newBin, evictedF)) {
        cuckoo(filter, newBin, evictedF, depth + 1);
    }
}
```

One takeaway

- Writing more modular code is often the best way to make your code easier to work with
- Superior to comments; can even be superior to simplifying expressions with intermediate variables.

MinHash Notes

How MinHash Works

First we generate a random permutation P.

- For every element x, take the first k entries of P that are in x (remember that x is a set)
- Concatenate them together to form a *signature*
- We want to compare every pair of elements with the same signature. So for each item, we hash the *signature* to index into a hash table of n bins.
- We all-compare all within each bin. If we find a close item we are done! Otherwise we start over from the beginning (with a new permutation)

MurmurHash

Two functions you can call (either work for this use case):

```
void MurmurHash3_x86_32(const void* key, int len, uint32_t
    seed, void* out);
```

- Hashes `len` bytes starting at `key` using random seed `seed`. Stores the output (32 bits) in `out`

```
void MurmurHash3_x64_128(const void* key, int len, uint32_t
    seed, void* out );
```

- Hashes `len` bytes starting at `key` using random seed `seed`. Stores the output (128 bits) in `out`.
- Make sure you pass 128 bits! Something like `uint64_t out[2] = {0 ,0};` works.

Why is 32 bits enough for Assignment 2?

SIMD instructions

Redux of String Checking

- Students seemed to be a bit uncomfortable with this code excerpt
- Why do these give different results?

```
char* str = "abcd";
uint64_t test =
    *((uint64_t*) str);
```

```
uint64_t test = *str;
```

SIMD

- SIMD: **S**ingle **I**nstruction **M**ultiple **D**ata
- A single CPU 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

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 16 32-bit integers with one SIMD operation
- `simdtests512.c`

Assembly of the add operation

```
        vpaddd  %zmm0, %zmm1, %zmm0 # _46, _45, _47
# simdtests512.c:19:     __m512i c = _mm512_add_epi32(a, b);
        vmovdqa64   %zmm0, 256(%rsp)    # D.26749, c
```

- In assembly, the data is moved around, and there is a single (special) add operation

Usual Breakdown of a SIMD Function

- Can get these functions from the Intel website; I'll give you all functions you need for Assignment 2 between here, the assignment, and `simdtests512.c`
- (Don't need *entire execution* to be SIMD on Assgn. 2! I want you to get *some* parallelism using the functions I've given you.)
- Example: `__m512i _mm512_add_epi32 (__m512i a, __m512i b)`:
  - `__m512i` is the return type (a 512 bit variable)
  - `_mm512` means that this is a 512 bit operation
  - `add` is the operation
  - `epi32` means that we are operating on 32-bit words (as opposed to packing, say, 64 8-bit words into the 512 bits)

Example 2: Adding single value to array

- Let's add one value (10) to each entry of an array.
- Do we need to declare a new array to do this?
  - No! SIMD operations give us a single function that fills a variable with copies of a single value

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 512-bit vector out
- Need a way to make that vector into something useful for us. Let's look at the code.
- `__mmask16 _mm512_cmp_epi32_mask(_mm512i arg1, _mm512i arg2, __MM_COMPINT_ENUM type)`: does 16 comparisons at once, stores results of all (bit by bit) in a 16 bit integer
- type is one of: `_MM_CMPINT_EQ` `_MM_CMPINT_LT` `_MM_CMPINT_LE` `_MM_CMPINT_FALSE` `_MM_CMPINT_NE` `_MM_CMPINT_NLT` `_MM_CMPINT_NLE` `_MM_CMPINT_TRUE`

Optimization comparison?

- What happens when we change to `-O3`?
- Everything gets faster!
- In previous tests: for adding and popcount, SIMD is suddenly slightly slower than without; SIMD is still best at finding 0 element
- 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
  - `gcc` only uses 256 bit operations by default, even if larger ones are available

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 512 bit registers, and operating on 32 bit data.

- Factor of $512/32 = 16$ 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

```c
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`.