Lecture 13: SIMD Instructions and Code Review

Sam McCauley

October 25, 2021

Williams College

Admin

- Assignment 5 due Wednesday
- Mini-midterm 2 next week; experimental focus

Plan for today

- Go over some of the “implementation” Minhash slides again
    - Hopefully you started already!
    - But may help reinforce some things you’ve only seen quickly, or may help you get a piece of it working more effectively.
- SIMD instructions with examples
- Assignment 3 code review

MinHash

128 in the assignment

- The hash consists of an *permutation* of all possible items in the universe

- To hash a set A: find the first item of A in the order given by the permutation. That item is the hash value!

- Concatenate k hashes to lower collision probability. If far points have similarity j_2, best when $k \approx \log_{1/j_2} n$.

- Repeat until finding the close pair (requires $O(R)$ hashes in expectation)

Practical MinHash Considerations

So many Permutations!

- OK, so kR repetitions is a LOT of preprocessing, and a lot of random number generation
- And most of this won't ever be used! Most of the time, when we hash, we don't make it more than a few indices into the permutation.
- Idea: Instead of taking just the first hash item that appears in the permutation, take the first (say) 3. Concatenate them together. Then we just need $k/3$ permutations per hash table to get similar bounds.
- So let's say we have $A = \{$black, red, green, blue, orange$\}$, and we're looking at a permutation $P = \{$purple, red, white, orange, yellow, blue, green, black$\}$.
- Then A hashes to redorangeblue

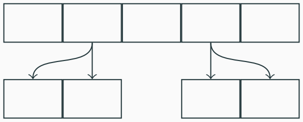
Reducing Permutations

- If you take the $\hat{k}$ first items when hashing, rather than just taking the first one, we only need $kR/\hat{k}$ total permutations.
- Does this affect the analysis?
  - Yes; the k we're concatenating for each hash table are no longer independent!
  - But this works fine in practice (and is used all the time)

Problems with Expectation

- We chose parameters so that buckets are small in expectation (i.e. on average)
- But: time to process a bucket is *quadratic*.
- So getting unlucky is super costly!
- What can we do if we happen to get a big bucket?

Handling Big Buckets



- One option: recurse!
- Take all items in any really large bucket, rehash them into subbuckets
- Might need to repeat
- This option can shave off small but significant running time
- (Not required; just one optimization suggestion.)

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the element we're hashing.
- How can we do this?
- Most efficient way I know is not clever. Just go through each index, and check to see if that bit is set (say by calculating `x & (1 << index)` —but remember that these are 128 bits)
  - Method to help with this was given in the starter code.

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from 0 to 127
- How can these get concatenated together?
- Option 1: convert to strings, call `strcat`
- Note: need to make sure to convert to *three-digit* strings! Otherwise hashing to 12 and then 1 will look the same as hashing to 1 and then 21. (012 and 001 instead)
- Option 2: Treat as bits. 0 to 127 can be stored in 7 bits. Store the hash as a sequence of k 8-bit chunks.

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.
- Why? Lots of buckets and lots of repetitions have bad constants.
- Smaller k means fewer buckets, fewer repetitions (but bigger buckets and more comparisons)
- Start with $k \approx \log_3 n$, but experiment with slightly smaller values.

Repetitions?

- You're guaranteed that there exists a close pair in the dataset
- My implementation just keeps repeating until the pair is found (no maximum number of repetitions)
- The discussion of repetitions in the lecture is for two reasons: 1. analysis, 2. give intuition for the tradeoff by varying k

How to Deal with Buckets?

- Each time we hash, (i.e. build a new “hash table”) need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we’re not hashing to a number from (say) 0 to $n-1$. We’re instead concatenating indices
- How to keep track of buckets?

How to Deal with Buckets?

- How to keep track of buckets?
    - This is a dictionary problem, where the “key” is the hash value of the bucket, and the “value” is the bucket itself. Can use your favorite dictionary solution to solve. (Bear in mind there are $\Theta(n)$ buckets.)
    - Can use murmurhash to convert a hash string value to a number; this means that you can keep the buckets in an array and index into them directly. (This will result in some buckets being combined, which hurts efficiency—but probably not too much.)
    - Can also do it in-place using sorting—don't need to store explicit buckets

Storing a Hash

- Just need a permutation on $\{0, \ldots, 127\}$
- How can we store that?
- First key observation: we (basically) *never* make it through the whole permutation (we'll always see at least one 1 first)
- Taking that a bit further: we only really need the first few indices. If we're using $\hat{k}$ indices from one ordering, something like $8\hat{k}$ or $16\hat{k}$ will almost certainly suffice.
- What about elements that hash further? Answer: just give them the value of the last index in the ordering.

Truncating Hash Example

- Let's say our permutation is $\{47, 11, 85, 64, 13, 74, 70, 107, 112, 103, 7, 95, 3, \ldots\}$ and $\hat{k} = 2$.
- I only store $\{47, 11, 85, 64, 13, 74, 107, 112\}$. If we go past 112 for some x, and we have not seen $\hat{k}$ indices that are a 1 in x, I just write 112 until I get $\hat{k}$ numbers.

Takeaway from Truncating Hashes

- This means we can store fewer bits, fewer random numbers
- Is this important for us? It may be nice (array is smaller) but probably doesn't change much
- But, is very important if number of dimensions is very high (i.e. Netflix movies watched)
- I want to be clear: don't spend time implementing this in your code. Just something to be aware of. (And I ask you to analyze one specific case of this in Problem 4.)

Back to SIMD

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

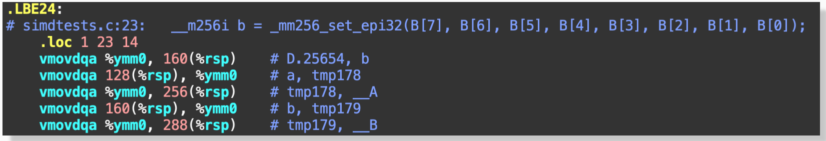
What is SIMD good for?

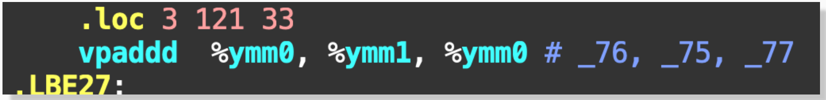
- Lots of identical operations on a set of elements; these operations are costly
- Elements are in nicely-sized chunks
  - Can always used specialized code to handle other cases

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:
```

How AVX2 instructions work

- Need to invoke operation
- Also need to state the type of underlying data. Even if you have 256-bit SIMD items, you need to specify:
    - Are you adding 4 64-bit longs?
    - Or 8 32-bit floats?
    - Or 32 8-bit unsigned integers?
- Each of these involves a different function
- Ex: `_mm256_add_epi32(first, second);` vs `_mm256_add_pd(first,second)`

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? Or can we make a vector of 10s manually?

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 (i.e. a SIMD add may take 2 times as long as a 64-bit add), but it seems modern implementations are much better)
- Cost to gather items into SIMD register
- 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

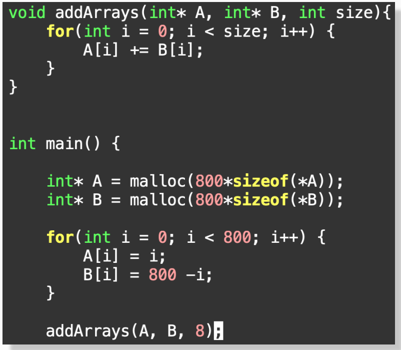
One Question

- 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



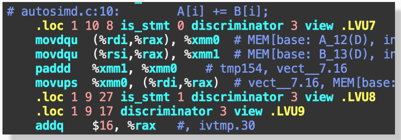
```c
void addArrays(int* A, int* B, int size){
    for(int i = 0; i < size; i++) {
        A[i] += B[i];
    }
}


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), i
    movdqu  (%rsi,%rax), %xmm1  # MEM[base: B_13(D), i
    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.

Assignment 3 Discussion

Assignment 3

- Looked good!
- Most common mistake: need to check both bins if it's empty when inserting before entering cuckoo loop.
- Perhaps biggest code challenge: extracting specific fingerprint from bin

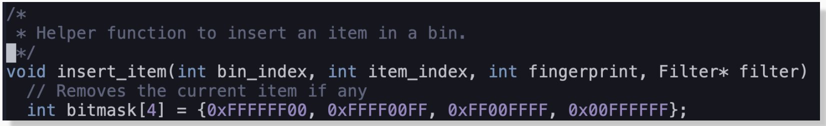
Using loop for mask

```
uint32_t bin;
uint32_t fingerprint;
//bin stores the whole bin we're looking at
//fingerprint stores the fingerprint we're looking for
for(int slot = 0; slot < numSlots; slot++) {
   if( (bin >> slot) & 255 == fingerprint) {
      return 1;
   }
}
```

255 masks out all but the last 8 bits of the bin (sets the rest to 0). This method works fine, but gets annoying for inserts.

- Need to first set slot to 0, then store fingerprint in correct slot

Using array for mask

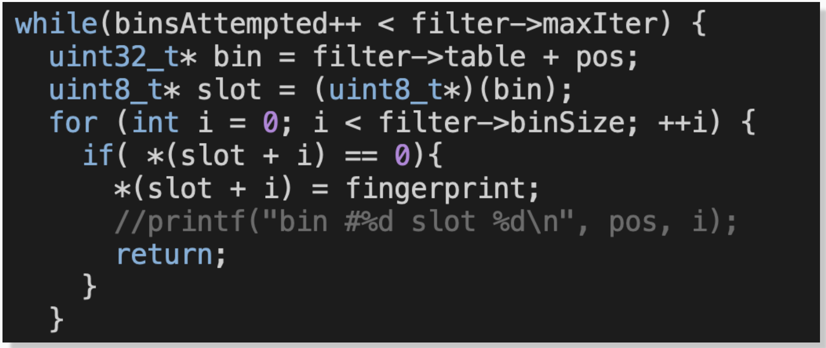


```
/*
 * Helper function to insert an item in a bin.
 */
void insert_item(int bin_index, int item_index, int fingerprint, Filter* filter)
  // Removes the current item if any
  int bitmask[4] = {0xFFFFFF00, 0xFFFF00FF, 0xFF00FFFF, 0x00FFFFFF};
```

This allows us to use something like

```
//store fingerprint in bin
bin = (bin & bitmask[slot]) + (fingerprint << slot);
```

Using 8-bit pointers



```
while(binsAttempted++ < filter->maxIter) {
  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;
      //printf("bin #%d slot %d\n", pos, i);
      return;
    }
  }
```

- Advantage: don't need to deal with masks at all!
- Disadvantage: do need to deal with potential pointer issues. (Not casting the pointer type correctly will lead to odd, subtle bugs.)

Separating out into functions

- No optimization criteria for Assignment 3
- So: cleanest solution is best solution. (Best for me/to maintain. But also, easiest for you to debug.)
- This is *always true*. But it is true that sometimes to write fast code, you should take clean code and make it less clean (e.g. by manually inlining functions)
- Let's look at one good example of a very very clean Assignment 3

Get and set in bin

```c
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.

One note on coding style

- Classic computer science problem (especially in C): how to gracefully exit a loop
- More difficult when you want to exit *two* loops (because `break` doesn't work).
- There are several ways to do this, and people have strong opinions about the benefits of each.
    - I don't think there's one right way to do things...
    - But I do think there are some specific suboptimal things to watch out for

Nested loop

```
void cuckoo() {
   for(int iteration = 0; iteration < maxIter;
       iteration++){
      for(int slot = 0; slot < 4; slot++) {
         if(bin[slot] == 0) {
           /* what do we want to do here? */
         }
      }
   }
   /*cuckoo failed if never hit the above*/
}
```

- Many of you had code that looks like this
- When find fingerprint, want to:
    - Update bin with new fingerprint
    - Get out of the whole function

Nested loop: 1st solution

```
void cuckoo() {
   for(int iteration = 0; iteration < maxIter;
       iteration++){
      for(int slot = 0; slot < 4; slot++) {
         if(bin[slot] == 0) {
           bin[slot] = fingerprint;
           return;
         }
      }
   }
   printf("failed!")
}
```

- In my opinion the best solution by far
- Only downside: multiple return points for function

Nested loop: 2nd solution (avoid return)

```
void cuckoo() {
   int flag = 1;
   for(int iteration = 0; iteration < maxIter && flag;
       iteration++){
      for(int slot = 0; slot < 4 && flag; slot++) {
         if(bin[slot] == 0) {
           bin[slot] = fingerprint;
           flag = 1;
         }
      }
   }
   if(!flag)
      printf("failed!")
}
```

- Also OK (IMO inferior, but not problematic)

Nested loop: 3rd solution (too much IMO)

```
void cuckoo() {
int flag = 1;
for( int iteration = 0; iteration < maxIter; iteration ++){
    for( int slot = 0; slot < 4; slot++) {
        if(bin[slot] == 0) { flag = 1; break;
        }
    }
    if( flag ) break;
}
if( flag )
    printf(" failed !")
else
  bin[slot] = fingerprint ;
}
```

- Messy; extra ifs; unclear when last line runs. (not ideal)

Point to bring home

- Sometimes getting out of a loop requires messy control flow
- May require things you usually want to avoid: `break`, flag variables, early returns, even `goto`
- Rule of thumb: use what's simplest; do the least possible
- Multiple exit points:
  - Definitely don't want to have unnecessary exit points
  - In my opinion, not worth writing more complex code to avoid multiple exit points (though I won't penalize for that alone)
  - More important: not worth doing extra work to avoid multiple exit points; execute code immediately when you know you can