Lecture 12: Code Review and SIMD Instructions

Sam McCauley

October 22, 2024

Williams College

Admin

- Apply to be a TA! (Especially for Algorithms)
- Assignment 2 out; get started early!
- Remember that this is an assignment so you should do the work on your own/with class materials.
- Questions?

Wrapping up Minhash Discussion

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 $A = \{\text{black, red, green, blue, orange}\}$, and we're looking at a permutation $P = \{\text{purple, red, white, orange, yellow, blue, green, black}\}$.
- 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)
- We will do this on the Assignment; in fact I recommend using $\hat{k} = k$. That means that each repetition has only one permutation.
- I think it makes life very significantly easier. In the real world you want a smaller value of $\hat{k}$
- I do think this hurts performance a little if you're trying to optimize

Assignment Parameters

- 128 bit integers (stored as a `struct` of two unsigned 64 bit ints; called an `Item`)
- It may be possible to store these as 128-bit SIMD variables; I haven't experimented with this
- Universe: $\{0, \ldots, 127\}$. (You can pretend that these are images, each of which is labelled with a subset of 128 possible tags.)
- Each bit is a 0 or 1 at random
- (Not realistic case, but hard case!)

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the `Item` we're hashing.
- How can we do this?
- Most efficient way I know is not clever. Just go through each index in order of the permutation, and check to see if that bit is set (say by calculating `x & (1 << index)` —but remember that these are 128 bits. An implementation of this function is included 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)
- This option work well but is slow
- 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? Having a large number of buckets and/or repetitions leads to bad constants.
  - Repetitions mean *hashing* which is expensive ($\approx$ 50 cycles). But comparing to a few more items just takes a few extra comparisons
- 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 if you want better performance

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?
- Similar to Assignment 1. Cache efficiency is still important if you care about performance (but is not required this time)

Homework 2 Optimizations

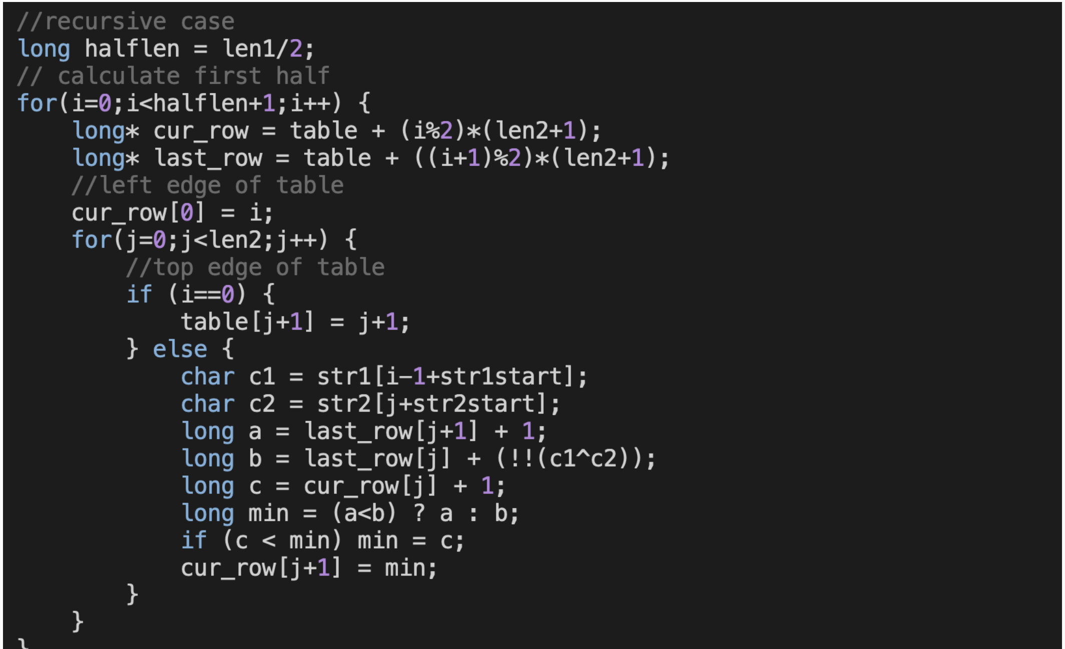
Homework 2

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

Assignment 2: Some ideas that seemed to work

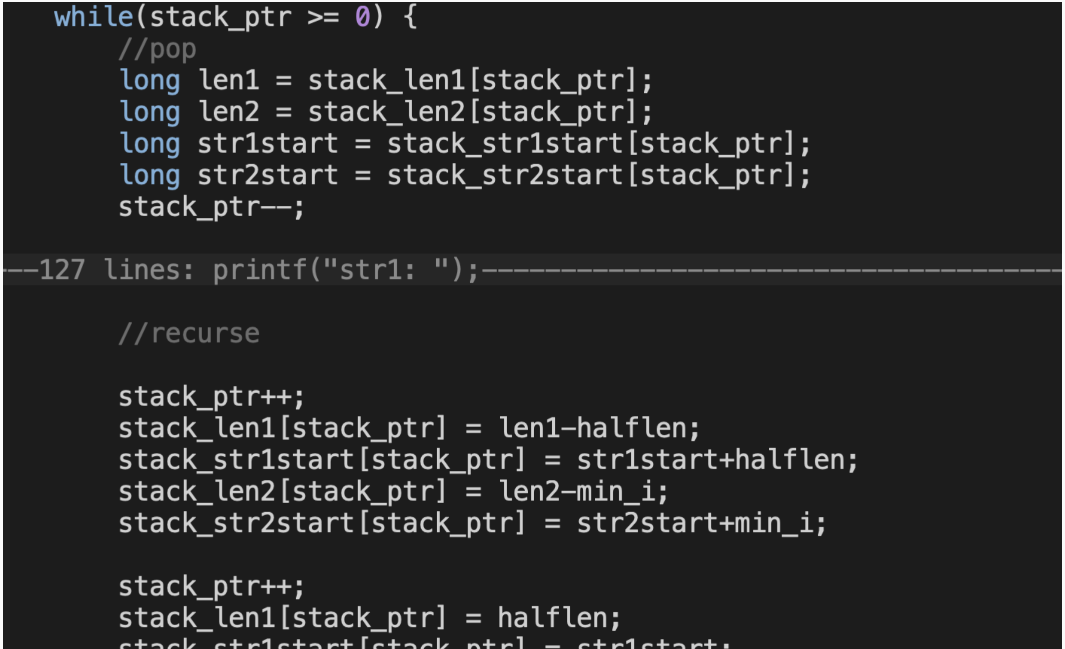
- Large base case
    - Why is this good? (Hint: why was Hirshberg's a good idea in the first place?)
    - 300, 2048 both used
    - Both small enough that `len1 * len2` fits in cache
- Iterative version!
    - Recursive calls have overhead; can skip them
    - To be honest this seems like it should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

Code from best last time (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 time (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;
```

Fastest Code This Time

- Simple backtracking implementation
- Nothing fancy/obviously fast
- Did use `-O3` flag (helped a lot!)
- Let’s take a look at it

Fastest Code This Time

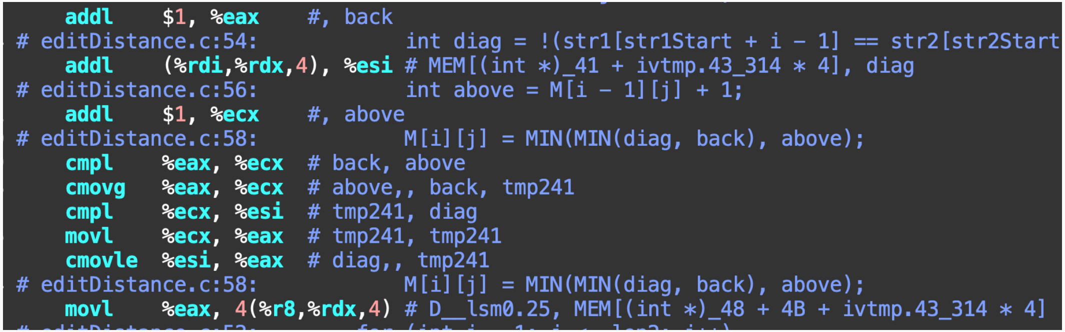
```c
for (int i = 1; i <= len1; i++)
{
    for (int j = 1; j <= len2; j++)
    {
        int diag = !(str1[str1Start + i - 1] == str2[str2Start + j - 1]) + M[i - 1][j - 1];
        int back = M[i][j - 1] + 1;
        int above = M[i - 1][j] + 1;

        M[i][j] = MIN(MIN(diag, back), above);
    }
}
```

```c
#define MIN(x, y) ((x < y) ? (x) : (y))
```

- Min does use an if, but it’s very obviously a min
- No “if” to figure out if adding 1

Fastest Code Assembly



```
    addl    $1, %eax     #, back
 # editDistance.c:54:            int diag = !(str1[str1Start + i - 1] == str2[str2Start
    addl    (%rdi,%rdx,4), %esi # MEM[(int *)_41 + ivtmp.43_314 * 4], diag
 # editDistance.c:56:            int above = M[i - 1][j] + 1;
    addl    $1, %ecx     #, above
 # editDistance.c:58:            M[i][j] = MIN(MIN(diag, back), above);
    cmpl    %eax, %ecx   # back, above
    cmovg   %eax, %ecx   # above,, back, tmp241
    cmpl    %ecx, %esi   # tmp241, diag
    movl    %ecx, %eax   # tmp241, tmp241
    cmovle  %esi, %eax   # diag,, tmp241
 # editDistance.c:58:            M[i][j] = MIN(MIN(diag, back), above);
    movl    %eax, 4(%r8,%rdx,4) # D__lsm0.25, MEM[(int *)_48 + 4B + ivtmp.43_314 * 4]
```

No jumps or branches! Just “compare-and-move” operations. (Thanks gcc!)

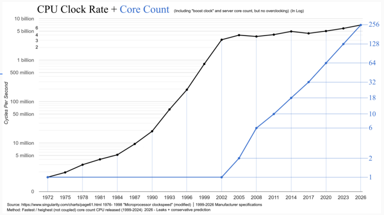
Assignment 1 Discussion

Hard part: Buckets

- Each item hashes to a bucket
- Want to make sure that buckets can be scanned efficiently when calling with naive 3SUM method
  - Giant arrays
  - Vectors
  - Linked list and copy to an array
    - Does this hurt cache-efficiency?
    - Not asymptotically! Need a cache miss to hash anyway
  - Just sort; scan for bucket size! (On board)

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!
- Need to compile with `-march=native` to use

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`, instead use `int __builtin_ctz (unsigned int x)`
- Let's look at some simple code using this: `trailingZeroes.c`

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
- There are limits; 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)
- But, of course, it will do its best when it can.
- The compiler does do good work for you: when you call (say) `int __builtin_ctzl (unsigned long)`, it chooses the best way to do the operation to count the number of leading zeroes on your processor

SIMD instructions

Intro: a touch of parallelism

- Something we've occasionally touched on in this class: *word-level parallelism*
- Idea: we can do computations on single 64 bit numbers very quickly (say 1 clock cycle)
- So: 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 `char`s) starts with “abcd” in $O(1)$ time?
  - Calculate a `uint64_t` corresponding to the correct integer
  - Eight bytes, where the first byte is the character `‘a’`, second is `‘b’`, then `‘c’` and `‘d’`
  - Cast the data to a `uint64_t`, then can do a single comparison

Watch for endianness

- I'd strongly recommend avoiding tricky back-end issues by storing data the same way you access it rather than manually
- *Both* actually work on any machine in the following example; C requires strings be stored in-order. But it's nice to not have to know that...

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

```c
uint64_t test = 'a';
test = test << 8 + 'b';
test = test << 8 + 'c';
test = test << 8 + 'd';
```

Let's look at simultaneously testing first four characters of a string

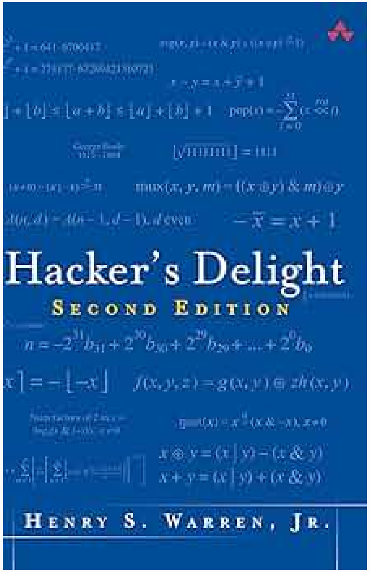
```c
bool starts_with_abcd(char* str){
    char* beginning = "abcd";
    uint64_t test = *((uint64_t*) beginning);
    uint64_t str_beg = *((uint64_t*) str);
    return test == str_beg;
}
```

- Is this faster? Sometimes...we'd have to run some tests.

In general...

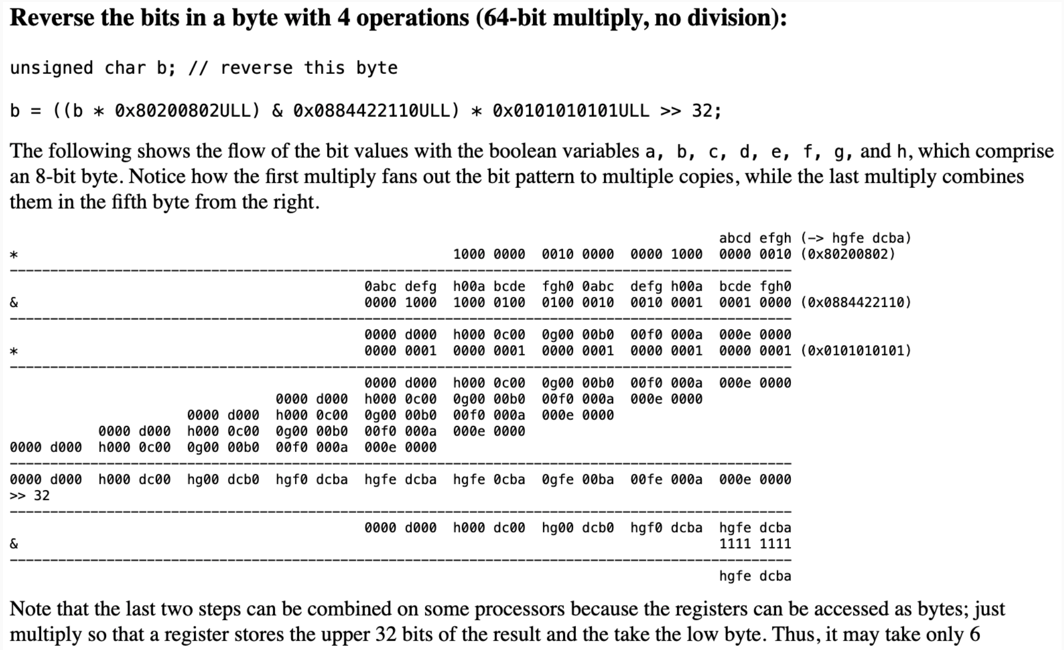
- Words of 64 bits allow us to do lots of computations in one (or a few) clock cycles
- Example: taking the bitwise OR of two 64 bit numbers is basically doing 64 computations at once
- This is literally parallelism: the circuits in the chip do these operations simultaneously
- Harder to do simultaneous operations like add (*really* hard 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

Lots of known tricks



- Whole books on the topic
- Very important for the *last level* of optimization
- Writing clear code is almost always better until the very end of optimizing; always test any tricks you use to make sure they're actually faster

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

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

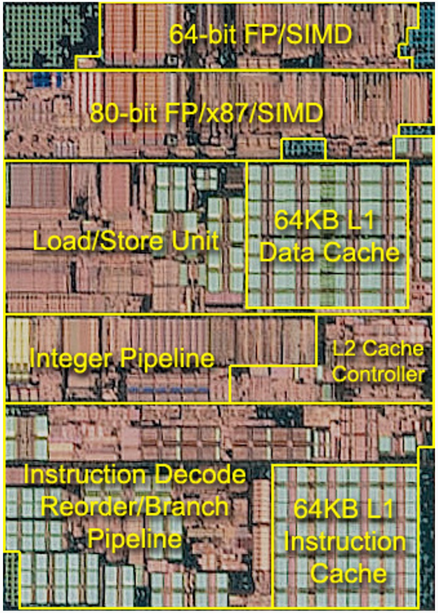
Extending it forward

- 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 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

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

- We have SSE, AVX, AVX2, AVX-512 instruction sets
- 32(?) “ZMM” registers; each 512 bits
- (Older processors may only have 128 bit “XMM” or 256 bit “YMM” registers.)
- Need to include `#include <immintrin.h>` and compile with `-march=native`

SIMD Examples

What is SIMD good for?

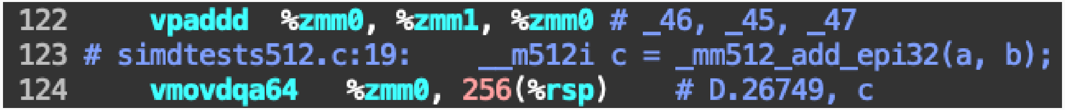
- Lots of identical operations on a set of elements; these operations are costly

- 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
- `__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

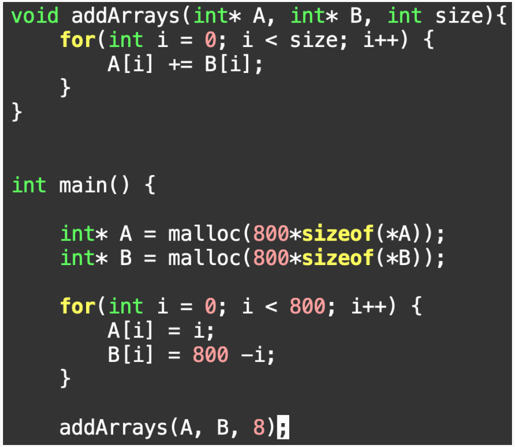
- Matrix multiplication: lots of time doing multiplications on successive matrix elements

- (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



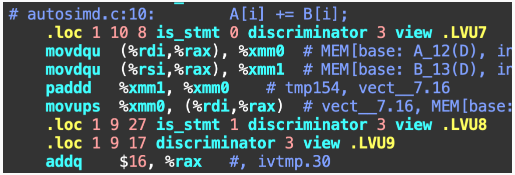
```c
void addArrays(int* A, int* B, int size){
    for(int i = 0; i < size; i++) {
        A[i] += B[i];
    }
}


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), 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`.

Memory Alignment

Storing Items in Memory

- We've mentioned that our computer moves data around in cache lines of 64 bytes
- Sometimes the hardware does better at taking in chunks of memory when they are lined up properly
- SIMD has different operations depending on if you're aligned or unaligned; affects performance
  - `load` vs `loadu` (u for "unaligned")
- Can use special commands when allocating to make sure the beginning of the allocated memory is aligned with a certain boundary
- Performance gain is quite small, but it's also really easy to set up