Lab 9

Lab 09

Objectives:

- TSW learn about and use various SIMD functions to perform data level parallelism
- TSW write code to SIMD-ize certain functions
- TSW learn about loop-unrolling and why it works

Setup

Pull Lab 09 files from the lab starter repository with

Disclaimer

NOTE THAT ALL CODE USING SSE INSTRUCTIONS IS GUARANTEED TO WORK ON THE HIVE MACHINES AND IT MAY NOT WORK ELSEWHERE

Many newer processors support SSE intrinsics, so it is certainly possible that your machine will be sufficient, but you may not see accurate speedups. Ideally, you should ssh into one of the hive machines to run this lab. Additionally, many of the performance characteristics asked about later on this lab are more likely to show up on the Hive.

Exercise 1 - Familiarize Yourself with the SIMD Functions

Given the large number of available SIMD intrinsics we want you to learn how to find the ones that you'll need in your application.

For this mini-exercise, we ask you to look at the Intel Intrinsics Guide. Open this page and once there, click the checkboxes for everything that begins with "SSE".

Look through the possible instructions and syntax structures, then try to find the 128-bit intrinsics for the following operations:

- Four floating point divisions in single precision (i.e. float)
- Sixteen max operations over signed 8-bit integers (i.e. char)
- Arithmetic shift right of eight signed 16-bit integers (i.e. short)

Hint: Things that say "epi" or "pi" deal with integers, and those that say "**ps**" or "**pd**" deal with **s** ingle **p** recision and **d** ouble **p** recision floats.

You can visualize how the vectors and the different functions work together by inputting your code into the code environment at this link! Another interesting tool that might help you understand the behavior of SIMD instructions is the Compiler Explorer project. It can also provide a lot of insights when you need to optimize any code in the future.

General advice on working with SIMD instructions:

- Be ware of memory alignment. For example, `_m256d _mm256_load_pd (double const * mem_addr)` would not work with unaligned data – you would need `_m256d _mm256_loadu_pd`. Meanwhile, it is almost always desireable to keep your data aligned (can be achieved using special memory allocation APIs). In fact, when the data is aligned, aligned load/store will give identical performance to an aligned store. Aligned loads can be folded into other operations as a memory operand which reduces code size and throughput slightly. Modern CPUs have very good support for unaligned loads, but there's still a significant performance hit when a load crosses a cache-line boundary.
- Recall various CPU pipeline hazards you have learned earlier this semester. Data hazards can drastically hurt performance. That being said, you may want to check data dependencies in adjacent SIMD operations if not getting the desired performance.

Exercise 2 - Writing SIMD Code

Common Mistakes

The following are bugs that the staff have noticed which were preventing students from passing the tests (bold text is what you should not do):

- **Trying to store your sum vector into a `long long int` array**. Use an int array. **Side note: why??** The return value of this function is indeed a `long long int`, but that's because an `int` isn't big enough to hold the sum of all the values across all iterations of the outer loop. However, it is big enough to hold the sum of all the values across a single iteration of the outer loop. This means you'll want to store your sum vector into an `int` array after every iteration of the outer loop and add the total sum to the final result `result`.
- **Re-initializing your sum vector**. Make sure when you add to your running sum vector; **you are not** declaring a new sum vector!!
- **Forgetting the CONDITIONAL in the tail case.** What condition have we been checking before adding something to the sum?
- **Adding to an UNINITIALIZED array.** If you add stuff to your result array without initializing it, you are adding stuff to garbage, which makes the array still garbage! Using `storeu` before adding stuff is okay though.

We've got one file `simd.c` that has some code to sum the elements of a really big array. It's a minor detail that it randomly does this `1 << 16` times... but you don't need to worry about that. We also pince the execution of the code between two timestamps (that's what the `clock()` function does) to measure how fast it runs! The file `test_simd.c` is the one which will have a `main` function to run the `sum` functions.

Task

We ask you to vectorize/SIMDize the code in `simd.c` to speed up the naive implementation of `sum()`.

You only need to vectorize the inner loop with SIMD! You will also need to use the following intrinsics:

- `__m128i _mm_setzero_si128()` - returns a 128-bit zero vector
- `__m128i _mm_loadu_si128(__m128i *p)` - returns 128-bit vector stored at pointer p
- `__m128i _mm_add_epi32(__m128i a, __m128i b)` - returns vector (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)
- `void _mm_storeu_si128(__m128i *p, __m128i a)` - stores 128-bit vector a into pointer p
- `__m128i _mm_cmpgt_epi32(__m128i a, __m128i b)` - returns the vector (a_i > b_i ? `0xffffffff : 0x0` for `i` from 0 to 3). AKA a 32-bit all-1s mask if a_i > b_i and a 32-bit all-0s mask otherwise
- `__m128i _mm_and_si128(__m128i a, __m128i b)` - returns vector (a_0 & b_0, a_1 & b_1, a_2 & b_2, a_3 & b_3), where & represents the bit-wise and operator

Start with the code in `sum()` and use SSE intrinsics to implement the `sum_simd()` function.

How do I do this?

Recall that the SSE intrinsics are basically functions which perform operations on multiple pieces of data in a vector in parallel. This turns out to be faster than running through a for loop and applying the operation once for each element in the vector.

In our sum function, we've got a basic structure of iterating through an array. On every iteration, we add an array element to a running sum. To vectorize, you should add a few array elements to a sum vector in parallel and then consolidate the individual values of the sum vector into our desired sum at the end.

- Hint 1: `__m128i` is the data type for Intel's special 128-bit vector. We'll be using them to encode 4 (four) 32-bit ints.
- Hint 2: We've left you a vector called `_127` which contains four copies of the number 127. You should use this to compare with some stuff when you implement the condition within the sum loop.
- Hint 3: DON'T use the store function (`_mm_storeu_si128`) until after completing the inner loop! It turns out that storing is very costly and performing a store in every iteration will actually cause your code to slow down. However, if you wait until after the outer loop completes you may have overflow issues.
- Hint 4: It's bad practice to index into the `__m128i` vector like they are arrays. You should store them into arrays first with the `storeu` function, and then access the integers elementwise by indexing into the array.
- Hint 5: READ the function declarations in the above table carefully! You'll notice that the loadu and storeu take `__m128i*` type arguments. You can just cast an int array to a `__m128i` pointer. Alternatively, you could skip the typecast at the cost of a bunch of compiler warnings.

To compile and run your code, run the following commands:

Sanity check: The naive version runs at about 7 seconds on the hive machines, and your SIMDized version should run in about 1-2 seconds.

Exercise 3 - Loop Unrolling

Concept Time! Another tactic used to increase performance is to unroll our for loops! By performing more operations per iteration of the for loop, we have to loop less and not have to waste as many cycles (think about why we would have to waste some cycles?). Theoretically, code would be faster if we didn't create loops and just copy pasted the loop `n` times, but that's not a very pretty function.

For example, consider this very simple example that adds together the first n elements of an array `arr`:

```
int total = 0;
for (int i = 0; i < n; i++) {
    total += arr[i];
}
```

The corresponding assembly code might look something like this:

```
        add t0, x0, x0
        add t1, x0, x0 // Initialize loop counter
loop:   beq t0, a1, end // Assume register a1 contains the size n of the array
        slli t2, t1, 2
        add t2, t1, a0 // Assume register a0 contains a pointer to the beginning of the array
        lw t3, 0(t2) // Load arr[i] into t3
        add t0, t3, t0 // total += arr[i]
        addi t1, t1, 1 // Increment the loop counter
        jal x0, loop
end:    ...
```

If we unroll the loop 4 times, this would be our equivalent code, with a tail case for the situations where n is not a multiple of 4:

```
int total = 0;
for (int i = 0; i < n / 4 * 4; i+=4) {
    total += arr[i];
    total += arr[i + 1];
    total += arr[i + 2];
    total += arr[i + 3];
}

for (i = n / 4 * 4; i < n; i++) {
    total += arr[i];
}
```

For the unrolled code, the corresponding assembly code might look something like this:

```
      add t0, x0, x0
      add t1, a1, x0 // Assume register a1 contains the size n of the array
      srli t1, t1, 2
      slli t1, t1, 2 // Find largest of multiple 4 <= n
      add t2, x0, x0 // Initialize loop counter
loop: beq t2, t1, tail
      slli t3, t2, 2
      add t3, t3, a0 // Assume register a0 contains a pointer to the beginning of the array
      lw t4, 0(t3) // Load arr[i] into t3
      add t0, t4, t0 // total += arr[i]
      lw t4, 4(t3) // Load arr[i + 1] into t3
      add t0, t4, t0
      lw t4, 8(t3), t0 // Load arr[i + 2] into t3
      add t0, t4, t0
      lw t4, 12(t3), // Load arr[i + 3] into t3
      add t0, t4, t0
      addi t2, t2, 4 // Increment the loop counter
      jal x0, loop
tail: beq t2, a1, end
      slli t3, t2, 2
      lw t4, 0(t3)
      add t0, t4, t0
      addi t2, t2, 1
end: ...
```

To obtain even more performance improvement, carefully unroll the SIMD vector sum code that you created in the previous exercise to create `sum_simd_unrolled()`. This should get you a little more increase in performance from `sum_simd` (a few fractions of a second). As an example of loop unrolling, consider the supplied function `sum_unrolled()`

Task

Within `simd.c`, copy your `sum_simd()` code into `sum_simd_unrolled()` and unroll it 4 (four) times. Don't forget about your tail case!

To compile and run your code, run the following commands:

Checkoff

Please submit to the **Lab Autorgrader** assignment.

In your check-in, feel free to explain your implementation of `sum_simd()` and `sum_simd_unrolled()`. How much faster did the SIMD code run over the naive implementation? How much of a performance boost did unrolling provide (and why did it increase performance)?