Lab 9: Data-Level Parallelism

Deadline: Friday, November 12, 04:00:00 PM PT

Goals

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

Setup

Warning: We strongly recommend working on the hive machines for this lab. Many older processors don't support SSE intrinsics, and your local machine may perform differently due to having different hardware (CPU cache size, memory speed, etc.).

In your `labs` directory, pull the files for this lab with:

```
$ git pull starter main
```

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 Intrinsics Naming and Usage documentation. Then, visit the Intel Intrinsics Guide. This site might also give a helpful overview. The hive machines support SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, and AVX2, so you can check those boxes in the filters list. Some of the other instruction sets are also supported, but we can ignore those for the purposes of this lab.

There isn't an autograded task for this exercise, but try looking through the various instructions in the guide and familiarize yourself with the syntax used. As an example, `__m128 _mm_add_ps (__m128 a, __m128 b)` **add**s **128**-bit vectors of `float`s (`ps` suffix), processing `128 / 32 = 4` floats per operation.

Exercise 2: Loop Unrolling Example

The `sum()` function in `simd.c` is an un-optimized implementation of the sum the elements of a really big array (roughly `2^16` elements). We use an outer loop to repeat the sum OUTER_ITERATIONS (roughly `2^14`) times to increase the code runtime so we can take decent speedup measurements. We time the execution of the code by finding the difference between the start and end timestamps (using `clock()`). The file `test_simd.c` is the one which will have a `main` function to run the various `sum` functions.

Let's look at `sum_unrolled()`. The inner loop processes 4 elements per iteration, whereas the inner loop in `sum()` processes 1 element per iteration. By performing more operations per iteration of the for loop, we have to loop less and not have to waste as many cycles -- remember the effects and hazards related to pipelining and branching! Note the extra loop after the primary loop -- since the primary loop advances through the array in groups of 4 elements, we need a tail case loop to handle arrays with lengths that are not multiples of 4.

▶ **Code Example**: RISC-V visualization of unrolling a sum function

Try compiling and running the code:

```
$ make simd
$ ./simd
```

The unrolled function should be slightly faster, although not by much. But faster programs are always nice to have!

Question: if loop unrolling helps, why don't we unroll everything?

- Loop unrolling means more instructions, which means larger programs and potentially worse caching behavior!
- Our simplified examples in `simd.c` use a known array size. If you don't know the size of the array you're working on, your unrolled loop might not be a good fit for the array!
- The unrolled code is harder to read and write. Unless you plan to never look at the code again, code readability may outweigh the benefits of loop unrolling!
- Sometimes, the compiler will automatically unroll your naive loops for you! Emphasis on *sometimes* -- it can be difficult to figure out what magic tricks a modern compiler performs (see

Godbolt in the next paragraph). For demonstration purposes, we've disabled compiler optimizations in this lab.

Optional: 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 Godbolt Compiler Explorer project. It can also provide a lot of insights when you need to optimize any code in the future.

General SIMD Advice

Some general advice on working with SIMD instructions:

- Be cautious 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, if you have control over memory allocation, is almost always desireable to keep your data aligned (can be achieved using special memory allocation APIs). 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 3: Writing SIMD Code

Common Bugs

Below are common bugs that the staff have noticed in implementations for this exercise. Some of it may not make sense yet, but feel free to refer back as you're working on the code:

- **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.
- **Re-initializing your sum vector**: make sure **you are not** creating a new sum vector for every iteration of the inner loop!

- **Trying to store your sum vector into a `long long int` array**: use an int array. 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. `long long int` and `int` have different bit widths, so storing an `int` array into a `long long int` will produce different numbers!

Action Item

Now, let's implement `sum_simd()`, a vectorized version of the naive `sum()` implementation!

You only need to vectorize the inner loop with SIMD. Implementation can be done with 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

▶ **Code Example**: vectorizing the sum of an 8-element array

Let's use `sum_unrolled()` as a reference, and use SSE intrinsics to re-implement the inner loop. Recall that the vector instructions 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 performing one operation on one piece of data per iteration.

In `sum_unrolled()`, we process 4 array elements in each iteration. Similarly, the vector functions listed above allow you to perform one operation on 4 integers at once, so the 4 sets of operations can be merged into 1 set of SIMD operations! When implementing `sum_simd()`, 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.

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

```
$ make simd
$ ./simd
```

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 4: Unrolling Loops

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 difference between `sum()` and `sum_unrolled()`.

Action Item

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:

```
$ make simd
$ ./simd
```

Exercise 5: Feedback Form

We are working to improve the labs for next semester, so please fill out this survey to tell us about your experience with Lab 9. The survey will be collecting your email to verify that you have submitted it, but your responses will be anonymized before the data is analyzed. Thank you!

Submission

Save, commit, and push your work, then submit to the **Lab 9** assignment on Gradescope. The autograder tests are similar to those in `test_simd.c`, but with potentially different constants (`NUM_ELEMS` and `OUTER_ITERATIONS`) and reduced speedup requirements (to compensate for more variability in autograder resources).

Checkoff
