

Assignment 7: Basic aligned SDOT using assembly and SSE

Due: Wednesday 03/16/05, Multiplication Factor: 2.0

This assignment is to get you comfortable with writing SSE x86 assembly, including addressing the required alignment concerns. This assignment will not be ranked by performance, but I will count off for bone-headed optimization decisions (eg., unnecessary loads in loops, unnecessary jumps, bad best-case code positioning, etc.). You can get credit back for mistakes of this sort by correcting the code.

I will also be grading on readability, as assembly needs to be made as clear as possible if it is to be maintained. Use the cpp tricks we discussed in class, and provide good (though not effusive) comments, particularly when you are doing something non-straightforward. You may assume the vector length is at least zero.

This routine will contain three loops:

1. A loop that assumes both X and Y are aligned to 16-bytes, and does a `sdot` using the vector SSE instructions (i.e., there must be an implicit loop unrolling of 4).
2. A loop that assumes X is 16-byte-aligned and Y is not, using the vector instructions.
3. A loop that uses scalar SSE instructions to guarantee alignment and to cleanup after the invocation of one of the vector loops.

On entry, you should check if X and Y are aligned, and if so, fall into the aligned loop. Otherwise, jump to code that calculates how to align X , sets things up, and jumps to the scalar loop to do the alignment. After X is aligned, if Y is aligned as well jump to the both-aligned loop, otherwise, jump to the X-only aligned loop. After the appropriate loop, you will again (possibly) jump to the scalar loop to handle any unrolling (implicit, from the vector instructions) cleanup before reducing `sdot` and returning.

I suggest tackling this problem in the following way:

1. First, write the scalar cleanup loop, and get basic `sdot` working.
2. Next, write the aligned loop, and have the tester always pass aligned X and Y (see below) with $N \bmod 4 = 0$, and get the basic vector `sdot` working. At this point, never call scalar loop you debugged in step 1. Remember to reduce the vector dot to scalar.
3. Add the logic for jumping to the scalar loop (using a jump-to-register to return) in order to support $N \bmod 4 \neq 0$. You'll want to have a register which is known to have the remainder as well, so that you can use this same loop later for alignment purposes.
4. Use your aligned loop to create your Y-misaligned loop, and add the logic to force alignment using the scalar loop, and choose which vector loop to use.
5. With everything coded, use `ddd` to trace execution to ensure the correct loops are being used (i.e., make sure you aren't always doing everything in the cleanup loop, etc).

To get the files you need, perform:

```
cp ~whaley/teach/cis5930/ASG7/Makefile .
make archdirs
```

You should create the file `sdot_sse.S` in the source directory, and this is the file you should e-mail to `whaley@cs.fsu.edu` by 10AM on the due date.

I have provided a tester for this routine which allows you to control the alignment of your input arrays. You build the tester with `make xsdotst`, and invoke it with `./xsdottst`. This gives you vectors aligned to whatever `malloc` returns. You can force particular alignments through the `-F` flag. This flag is suffixed by the vector to be forced to a particular alignment (eg., `y` or `x`, and lower case is required). The flag also takes a number, and if this number is positive, the vector is aligned to *at least* that byte alignment, and if it is negative, the vector will *not* be aligned to that alignment. For instance `-Fx 16 -Fy 8 -Fy -16` forces `X` to be 16-byte aligned, while `Y` is 8-byte aligned, but not allowed to be 16-byte aligned. Therefore, note that when a particular array has a negative and positive constraint, the negative number must be greater than the positive in absolute value.

Note that it would be more efficient to implement our vector cleanup loop as a switch statement, since we can be sure that the amount is between 1 and 3 (why is that?). Is there any advantage for later optimization if we keep it a loop?