

## EC527: High Performance Programming with Multicore and GPUs

### Programming Assignment 3

#### Objectives

Learn about and practice using: small-scale vector programming using SSE.

#### Prerequisites (to be covered in class or through examples in on-line documentation)

**HW** – Vector processing

**SW** – SSE instructions and their use

**Programming** – alignment, use of intrinsics, mapping C to vector instructions (e.g., with struct/union)

Note: For some intrinsics you need to enable AVX. Use the '-mavx' option to make it work.

---

### Assignment

---

#### Part 1 -- SSE extensions using C structs and union

Reading: B&O web extension "Achieving Greater Parallelism with SIMD Instructions."

Given: [test\\_combine8.c](#), [test\\_dot8.c](#)

Read the reading and the code. You will notice that solutions to B&O (web extension) practice problems 1, 3, and 4 have been implemented in the two .c files.

**1a.** Modify [test\\_combine8.c](#) to use `float` data type, set `IDENT` and `OP` for addition. Note that `VSIZE` is the number of data elements that fit in `VBYTES` bytes. Choose `A`, `B` and `C` so that  $Ax^2+Bx+C$  is always a multiple of `VSIZE`, and when `x=NUM_TESTS` it should be 10000 or so.

(When allocating memory for use with vectors it is important for the allocated memory to be "aligned"; read [notes\\_align.txt](#) to learn more about this. There is also a program [test\\_align.c](#) that accesses scalar data (one double at a time) with different alignments. On older machines it showed the performance penalty of using misaligned data; but on modern machines there is little observable effect. [notes\\_align.txt](#) shows sample output from older machines so you can see how much things have changed.).

Compile [test\\_combine8.c](#) and run. Plot the results and get the CPE. Justify the vector results (also comparing with the scalar results).

**1b.** Currently [test\\_combine8.c](#) has a function that does vector unrolling using 4 accumulators. Write code for two more functions, with 2 and 8 accumulators, respectively. Notice you need to also change the `OPTIONS` constant, add blocks of code in `main()` to test these new functions, change the first `printf` in the output section, etc.. Plot the results, get the CPEs. Discuss how and why the CPEs are different.

**1c.** Recompile using `double` rather than `float`. Does having 8 accumulators still help?

**1d.** Compile and run [test\\_dot8.c](#) using `float`. Plot the results and get the CPE. Justify the vector results (also comparing with the scalar results).

**1e.** Currently [test\\_dot8.c](#) has vector unrolling using 2 accumulators. However, there is a problem with it -- it is computing the wrong answer! Look at the "Computed dot products" and notice that the numbers in the `dot2_8` column are different from the other columns.

A test of this is provided. Near the beginning of `main()`, change the `"if (0)"` to `"if (1)"`, compile and run, and see the results of `dot4()` and `dot2_8()` using vectors of length 10.

Debug and fix `dot2_8`. Use the appropriate hints from the "Debugging Your Program.txt" file provided on Blackboard. For example, print out the contents of the input vectors and compute the correct answer yourself. Another hint: How big is the error (much does the `dot2_8` answer differ from the correct answer), and can you see the relationship between the vector size and this error?

**1f.** Now that you have `dot2_8` working, write code for new functions with 4 and 8 accumulators. As before you have to add to `main()` in a few places. Plot the results, get the CPEs, and justify.

**Hand in: results, code, and explanations of results. Explain why the CPEs are different for dot and combine and the various unrollings.**

## **Part 2 -- SSE extensions using intrinsics.**

Reading: Alex Fr "Introduction to SSE Programming"

Given: `avx_align.c` and `test_intrinsics.c`

Read the reading and the code (`avx_align.c` and `test_intrinsics.c`). The second of these has work functions that scan through two input arrays, performing an element-wise calculation  $1/2 + \sqrt{a^2+b^2}$  and writing the results to an output array.

**2a.** The programs in part 1 were using the `"__attribute__((vector_size(VBYTES)))"` method to create and manipulate AVX vector data. They also have a `new_array()` routine that allocates space for an array of data and returns a pointer that is aligned to a multiple of 32 bytes.

Now however we are going to use the "intrinsic" data types like `__m256` and "intrinsic functions" like `_mm256_sqrt_ps` which introduce some new issues for alignment when loading and storing data.

Compile `avx_align.c` and run. This program tries to do the same thing four different ways: Starting with an array in memory, read part of it into a `__m256` vector, compute square roots, and write the results back into the array in memory. The functions are called `unalign_local_alloc`, `unalign_heap_naive`, `align_heap_1`, and `unalign_storeu_ps`.

Do all four methods work? Which ones do not work? Do a search online like "AVX store Segmentation fault" to find out why, then explain the problem. If a function is not working, modify `main()` so it no longer calls the broken function(s), then compile and run again.

**2b.** In `test_intrinsics.c`, set A, B and C as before. Compile and run. Plot the results and get the CPEs. Is this what you expected?

**2c.** Create two simple functions to get execution time baselines: element-wise add and multiply (float only). What is the CPE? Can you vectorize these functions to make the throughput optimal?

**2d.** Create a vectorized dot product function using intrinsics, in particular, using the dot product primitive `_mm_dp_ps` (see the handwritten section near the end of the class notes "L03b\_SSE.pdf", or find a description online).

When you think you have it working, compare the computed result of the dot product calculation to a known working dot product like `dot4()` from `test_dot.c`. Until it is computing the correct answer, there is no point in trying to finish this question.

Plot time measurement results and get CPE. Compare the results you got this time (using intrinsics) to the results from `test_dot8.c`, which used `__attribute__((vector_size(VBYTES)))` declaration.

**2e.** Answer the following question: How does this approach compare with `test_dot8.c`? What are the specific differences (performance, programmability, etc.) and when do they matter? (Please note – unlike most other questions in these assignments, this one is mostly qualitative.)

**Hand in: modified code, description, results, answers to questions.**

## **Part 3 -- A simple SSE application from scratch: Transpose**

Given: `test_transpose.c` (similar to what you would have made in Lab 1 part 5)

**3a.** Create the fastest transpose you can. Try using the SSE transpose intrinsic `_MM_TRANSPOSE4_PS`. Try combining the transpose intrinsic with blocking (like you did in Lab 1).

**3b.** Compile `test_transpose.c` with `-O2` and `-O3` options and compare with your version.

**Extra Credit:** Create a function that performs a 4x4 transpose of *double* values using AVX `_mm_256_*` operations, and use it to make `test_transpose.c` work with `data_t` defined as `double`.

**Hand in:** *modified code, description, results, and analysis.*

**Part 4: QC**

- 4a. How long did this take?
- 4b. Did any part take an “unreasonable” amount of time for what it is trying to accomplish?
- 4c. Are you missing skills needed to carry out this assignment?
- 4d. Are there problems with the lab?