# EC527: High Performance Programming with Multicore and GPUs
## Programming Assignment 3

**Objectives**
Learn about and practice using: small-scale vector programing 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 is 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. Write code for a 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: **`test_intrinsics.c`**

Read the reading and the code (`test_intrinsics.c`). The work functions 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.** 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?

**2b.** 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?

**2c.** 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). Plot results and get CPE. Compare the results you got this time (using intrinsics) to the results in parts 1d and 1e above using `test_dot8.c`, which used `__attribute__ ((vector_size(VBYTES)))` declaration.

**2d.** 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.***