EC527 – High Performance Programming with Multicore and GPUs

# Lab 0 — Getting Started

Learn about and practice using tools and techniques for performance evaluation of computer programs.

**Readings** (all are on Blackboard):
**lab_orientation** PDF (in Assignments section)
**B&O Chapter 5 intro** and **5.2** (in Course Documents > Readings, BO_chapter5_1.pdf and BO_chapter5_2.pdf)
**P&H Chapter 6.10** and **6.11** (in Course Documents > Readings, PH_6_p2.pdf)

**Administration**

*This section repeats some of the things in the Lab Orientation document, but you still need to read that.*

**Deliverables:** The ZIP file you submit should contain a PDF that is the write-up (answers to questions, graphs, etc.), and code for each program you write and/or modify (as specified below). As always, changes in the code should be well and clearly commented. Combine all of these files into a single archive as described in the Lab Orientation and submit it using the Blackboard Assignment function.

**Code for validation by the graders:** Code should be compilable and executable on the "Grid" workstations (SIGNALS and VLSI machines in the 3rd floor Linux labs, PHO 307 and 305 respectively). Be sure to include compilation instructions (or Makefile) and tell which machine(s) you tested your code on. It's OK if you develop code on your own computer, but it is also your responsibility to make sure that everything works on the common machines. Graders do not have access to your machine!

**Prerequisites for this assignment**

Basic programming in C and use of Linux programming environments. More advanced C skills (including how to debug) will be essential in later weeks (see "Debugging Your Program.txt" in Blackboard > Assignments).

---

## Setting up

---

**Accessing the Machines.** The "Lab Orientation" PDF describes remote access and in-person access.

---

## Assignment

---

### Part 1. Find machine characteristics.

1a. Use the following commands to find the basic machine characteristics:

```
less /proc/cpuinfo
lscpu
```

- What CPU are you using?
- What is the operating frequency of the cores?
- How many cores are there?

1b. Search the web to find out more machine details:

- How many levels of cache are there and what are the characteristics of each one?
- What is the microarchitecture of the processor core?
- Are all of the cores real?
- Can you find information online related to the machine's memory bandwidth? If so, what did you find?

***Turn in answers to the questions.***

### Part 2. Timers.

Accurate timing is essential to performance evaluation. It is also a (perhaps) surprisingly subtle topic, and getting more so all the time. There are several ways to time program execution which you should explore as follows.

Throughout this part, use the `-O0` optimization setting. Also note the `-lrt` switch is needed to compile `test_clock_gettime.c` on the lab machines.

2a. Read **`notes_timers.txt`** and **`test_timers.c`**. The source file `test_timers.c` contains three different timing mechanisms. Compile and execute. Observe each method's effectiveness and accuracy.

- How can you tell whether the timer is accurate? In its deepest form, this is a fundamental question in physics; no need to go that far! First answer this question generally: how do you determine accuracy? Then consider the resolution to which you can easily determine accuracy. What is it?

2b. The timer that uses RDTSC has some basic problems. Do an internet search to find out what they are.

- What problems do RDTSC-based methods have? Is it still useful? Note: you can solve these problems yourself, but no need to do so for this lab!

2c. Each timer requires some calibration which may or may not have been done correctly in the source file.

- For each timer, what (potentially) needs to be done?
- As necessary, adjust the constants to give correct timing. (Note that all are trying to print values in units of seconds.)
- Describe how you did this and the modification (if any) you made to `test_timers.c`

`2d.` Perhaps the best way to do timing nowadays (on our systems) is by using `clock_gettime()`. The source file **`test_clock_gettime.c`** has a partially written testbench for this function. Use the `-lrt` option when compiling this.

2e. Notice the function "`diff()`" which takes two `timespec` arguments and returns a `timespec` result. You need to add two calls to this function, near the end of `main()`, as indicated by the comments. This should be simple, but will also test your basic C knowledge.

2f. Write dummy code that takes time close to 1 second to execute. It won't always take exactly the same amount of time to run, but how close can you get it? Both accuracy and precision play a role here: What is the resolution of the measurements given by `clock_gettime`, and (approximately) what is the standard deviation when you make several repeated attempts?
***Turn in your new test_clock_gettime.c and the answers to the questions.***

**Part 3. Plotting/Graphing Data.**

Presenting data *accurately* and *concisely* is essential in this course. Most of the time simple methods will do, once you figure out what you need to display. This is a plotting/graphing practice exercise, and it is also described in the "Lab Orientation" PDF which shows what the output might look like.

- Compile **`plotting.c`**
- Run **plotting** and output data to plotting.csv ( `./plotting | tee plotting.csv` )
- Start OpenOffice Calc ( `soffice plotting.csv &` ). Click on "OK" button.

You will notice that there are three separate data collections. Graph them as follows:

1. X-Y plot
2. Scatter plot
3. Scatter plot with X-Axis as log axis

The legend should come up automatically.
**Note:** Instead of using a csv file, you could copy-and-paste directly from your terminal into the spreadsheet.
***Turn in your plot (a screen shot / clipping is fine). You do not need to turn in anything else from Part 3.***

**Part 4. Performance evaluation using Cycles-per-Element (CPE).**

The book ***Computer Systems: A Programmer's Perspective*** by Bryant and O'Halloran is incredibly rich and insightful. Much of the material here is derived from Chapters 5&6. As described in B&O Chapter 5, evaluating performance requires establishing a deep understanding of how your code interacts with the CPU and memory hierarchy. But even with a deep understanding, noise is really hard to avoid. Please note that "noise" in this sense is not random, but rather caused by interactions that are often related to parameters in the code under study. Therefore simply running the same identical experiment multiple times will not help!

B&O suggest an excellent way to run performance experiments: find the number of cycles it takes to compute each marginal (additional) element in a program. In effect, instead of using single points, you plot a line and use its slope.

Benefits are as follows:

- Multiple experiments that are more independent than if the same problem size is used
- Exposure of quirky data points, e.g., caused by interactions with cache associativity
- Ability to apply various slope finding algorithms, e.g., standard regression or robust methods that eliminate outliers.

Use the `-O1` optimization setting for this part.

Note: Sometimes the first point in each graph is obviously garbage. Feel free to ignore it/them, either by skipping it in your output, or by not using it in curve fitting in your spreadsheet.

4a. Read **B&O Chapter 5 intro** and section **5.2**. (in BO_chapter5_1.pdf and BO_chapter5_2.pdf)

4b. The example from Chapter 5.2, figure 5.1 (functions `psum1` and `psum2`) is provided in **`test_psum.c`**
Add timing using `clock_gettime()` and `diff()` as in `test_clock_gettime.c`; and plot your results as discussed in the previous sections.

- Make sure to test large enough sizes so that your timing measurements are meaningful. Remember what you observed about accuracy and precision in part 2. The textbook only measures sizes up to 200, you will need to go a lot bigger.

4c. You may notice some anomalies (making it impossible to draw a straight line through all the data points). What's a good way to get rid of them?

4d. As in the textbook example, estimate the cycles per element (CPE) by finding the approximate slope of a line fitting the data points. How many cycles per element does it take on your computer for `psum1` and `psum2`? Is it the same as in B&O's Figure 5.2? If not, why might it be different?
(A similar figure is in a slide titled "Cycles Per Element (CPE)" in the lecture notes, probably "L00b_PerformanceEssentials.pdf")

***Turn in the modified test_psum.c and your plot.***

**Part 5. Interacting with the compiler.**

One of the difficulties in developing an understanding of how programs interact with hardware is making sure that your program actually does what you intend! In particular, the compiler will, by default, try to make your code as efficient as possible. Usually that's a good thing, but not when it obfuscates your attempts to understand machine behavior. For example, if your program is not actually doing anything (e.g., it has only reads with no writes as in `test_timers.c`), then the compiler may *get rid of all of your code* leaving nothing to measure. This is why you compiled `test_timers.c` with the `-O0` optimization setting: this turns off all optimization and leaves your code as is (however silly your code may be!). But the way to *really* make sure that what you expect to happen has actually happened is to read the assembly language code. That is what you will now do. Some basics:

- `-O0` compiler directive turns off all optimizations. `-O1` turns on basic optimization. `-O2` and `-O3` turn on more complex optimizations. More about all of these later in the semester.
- `-S` compiler directive generates an assembly language file. While compiler-generated assembly language can be opaque, looking at it is essential to confirming what code is being executed.
  To do: Observe what happens when `-O1` (as opposed to `-O0`) is applied to the `test_timers.c` kernel code, i.e., the code that we use as a delay loop.

5a. Compile and run **`test_O_level.c`** as follows:

```
gcc -O0 test_O_level.c -o test_O_level
time ./test_O_level
```

What gets printed?

5b. Now compile and run `test_O_level.c` as follows:

```
gcc -O1 test_O_level.c -o test_O_level
time ./test_O_level
```

What gets printed? You should see that the execution time has gone down nearly to 0 seconds.

5c. This time compile test_O_level.c as follows to generate the assembly language code:

```
gcc -O0 -S test_O_level.c
```

Look at `test_O_level.s` and find the loop. Don't panic! Look for the two "`call puts`" or "`callq _printf`" lines which are the `printf()` statements. Also, the variables "`steps`" and "`i`" are at offsets from `rbp`, like `-8(%rbp)` or `-32(%rbp)`. The "`$`" means "treat this number as an immediate (constant)" so `$3` means 3. Rename the file to save it.

5d. Now generate the assembly language code of `test_O_level.c` with basic optimization:

```
gcc -O1 -S test_O_level.c
```

Look at `test_O_level.s` again and find the loop. What do you see? How does it compare with the `-O0` version?

5e. You should have found something strange. Perhaps using the variable "`steps`" will solve the problem (how?).

Try this: modify `test_O_level.c` so that "`steps`" gets printed out after the loop is done.
Compile again with "`-O1`". How much time does the code take to execute now?
Compile again with "`-O1`" and "`-S`" and look at `test_O_level.s` again. What is happening? That is, how has the code been optimized?

***Turn in answers to the questions.***

**Part 6: Generating roofline plots**

Recall that roofline plots are a nice way of displaying the bottleneck inhibiting performance of a particular program/computer combination. Good roofline analysis tells the programmer where to put optimization effort. For example, is the program compute bound or memory bound?

6a. Read the section in the textbook called "The Roofline Model" (Patterson & Hennesey section 6.10, probably in "PH_6_p2.pdf"). In it they talk about the "***stream"*** benchmark (for example in the caption to figure 6.18). The "stream" benchmark is the subject of this part of the lab. Think about how such a benchmark might operate (what work does "Kernel 1" and "Kernel 2" try to do? How might they differ?).

6b. Read the **`stream.c`** code. There is a lot of detail having to do with reproducibility which you can skip, but you should find the "payload" loops, i.e., the inner loops where the data transfers are being generated. Are they what you imagined while you were reading the P&H section? In particular, note that finding the performance of even something as simple as the memory bandwidth is quite complex. The authors of STREAM use four different methods and a variety of parameters. The standard way to interpret the results is to average the highest values for the four methods.

6c. Compile and run. What is the memory bandwidth? How does it compare to any memory bandwidth information you found on-line (Part 1)?

Use the **`stream_simple.c`** code for the following parts of this task. Start by reading and understanding the code.

6d. Adjust the Arithmetic Intensity by modifying *all of the lines marked "`//Modify`"*, then recompile and rerun.

For example, the given code has 1.0 floating-point operation (FLOP) per loop and 8.0 Unique Reads per loop (there are also some integer additions but those are not "FLOPs"). The following corresponds to 4.0 floating-point operations (FLOPs) per loop and 2.0 Unique Reads per loop:

```
#define FLOPs_per_Loop 4.0
#define Unique_Reads_per_Loop 2.0

b[j] = a[j] * a[j] + a[j] * a[j] + (float)(integer[0][j]);
```

When changing the third line (the one with the code) you must also change the first two lines to agree with whatever change you made. Otherwise the program will just calculate incorrectly and print a useless result.

Repeat this to get data points covering at least the range [1/8, 2].

6e. Plot your results. You are likely to find regions with different slopes. Your scales and data points should enable you to determine them.

6f. What are your conclusions? What have you found out about Arithmetic Intensity and measured memory bandwidth? What does this say about achieved GFLOPS/s (billions of floating-point operations per second)?

Note: it may be hard to get a good roofline plot. There are several causes for this, but the most likely is that your added complexity gets optimized away by the compiler (thinking it is doing you a favor!).

Try making an example that is multiplying nine copies of `a[j]`, and compile `stream_simple.c` with:

```
gcc -O1 -S stream_simple.c
```

By compiling the code with the `-S` flag, you will get an assembly file `stream_simple.s`. Try searching for "`mul`" inside the file. You should see 8 multiplications in a row somewhere.
***Turn in your plots and answers to all questions.***

**Part 7: Quality Control**

7a. Are you missing skills needed to carry out this assignment?
7b. How long did this take (hours)?
7c. Did any part take an "unreasonable" amount of time for what it is trying to accomplish?
7d. Are there problems with the lab?
***Turn in answers to questions.***