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. Use the information from 1a to search the web to find out more machine details:

- How many levels of cache are there and what are the characteristics of each one? (Self study: do you know what these characteristics all mean?)
- What is the microarchitecture of the processor cores?
- Are all of the cores real? (what does this mean?)
- Can you find information online related to the machine's memory bandwidth? If so, what did you find? (You will use this in part 6.)
- What is this processor's "Base Frequency" and "Turbo Frequency" in GHz? (You will use this in part 2.)

Turn in answers to the questions.

Part 2. The computer's self-measurement of time

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 automatically time program execution, which we explore here. 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.

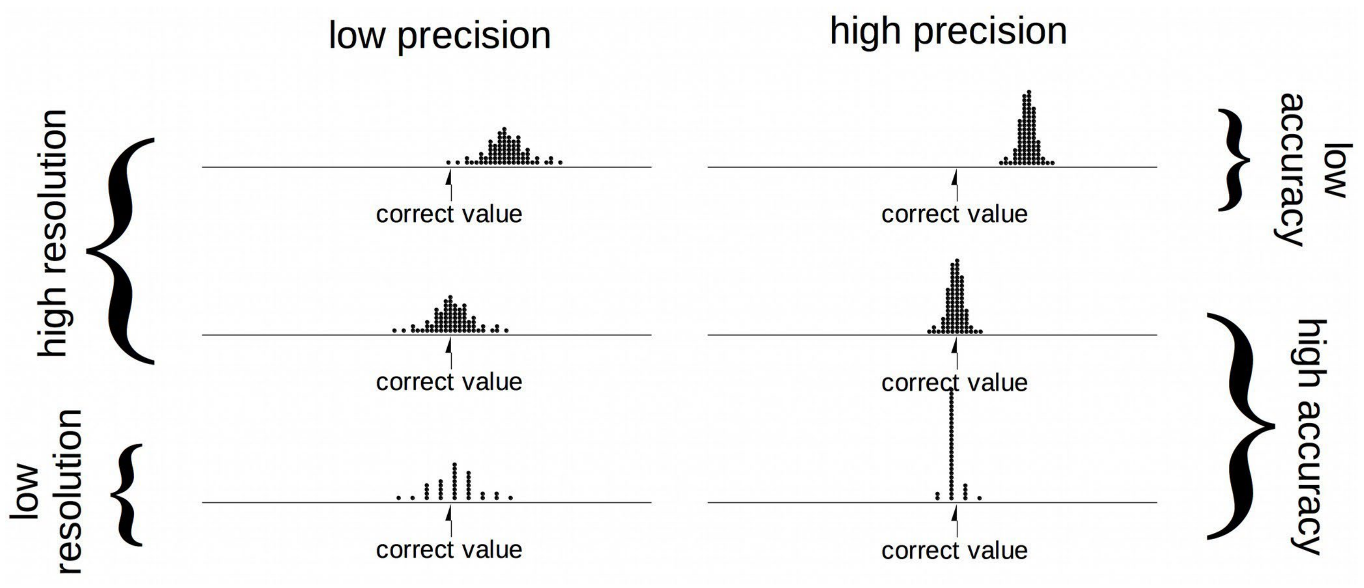

Figure 1: Accuracy, precision, and resolution. (Each • represents a single measurement)

2a. Read notes_timers.txtand test_timers.c. The source file test_timers.c contains three different timing mechanisms. Compile and execute. Observe each method's resolution, precision, 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 scientists determine accuracy?

Then consider how well *you*, without the aid of the un-trusted lab computer, can determine accuracy of something it is printing out. What is the accuracy with which *you* can determine a time measurement?

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 *could* solve some of these problems yourself, but no need to do so for this lab!

• You may wonder what CPU clock frequency to use when converting an RDTSC measurement to seconds. Referring back to part 1b, an Intel processor runs at a really low speed (like 800 MHz) to save power until a program starts running; then if it is cool enough it runs at the "Turbo Frequency" — but only for about 1/10 to 1/4 of a second; and then after that it slows down to the "Base Frequency" or something close to that.

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 to calibrate the timer?

• 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 timespecarguments and returns a timespec result. You need to add a call to this function, near the end of main(), get the information it returns and print it out, as indicated by the comments. This should be simple, but will also test your basic C knowledge.

2f. Write dummy code that takes 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? Resolution, accuracy, and precision all 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? If you do five runs in a row really quickly, remember (as explained in part 2b) that it will probably run a little faster the first time.
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 basics and 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.

"Noise" and how to deal with it

But even with a deep understanding, noise is really hard to avoid. Please note that "noise" in this sense is not random (and so we are really not using the term "noise" correctly), but rather hard-to-explain data points caused by interactions that are often related to parameters in the code under study and how they interact with some property of the machine (e.g., size of L1 cache). Therefore simply running the same identical experiment multiple times will not help – you will keep getting the same "weird" points!

So let's define "noise" in this context as the small number of points that are not on an otherwise well behaved curve. Sometimes the first point in each graph is obvious 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. Sometimes there are other points that are systematically far off an otherwise well-behave curve. There are generally only a few such points, which means they will be mosly ignored with standard curve fitting techniques. If there are enough bad points to seriously affect your curve fitting, then altering the points you are looking at may be necessary. For example, you should always avoid powers of 2!

Creating experiments that cover depth and breadth

Most functions are useful over a very wide range of data. For example, it makes sense to perform matrix-matrix multiply (MMM) or Fast Fourier Transform (FFT) on data sizes from single digits to the millions. (Multiplying E6XE6 matrices? Yes!) So what's the problem? Aren't computers fast? Can't we just measure all sizes from 10 to 1,000,000? If that's too many, how about all sizes in that range by 10s? 100s? 1000s?
Problem 1: Execution time can get ***much*** longer with size. MMM for nxn matrices takes $\mathbf{n^3}$ time. So small sizes take microseconds, 4Kx4K can take minutes, larger can take over night (or never finish on your workstation).
Problem 2a: You always want the widest range of data. That is you would like to see the behavior of your function until **n** gets so big that it would take hours to get a data point. Why? (i) These are important cases and (ii) you never know (for sure) when behavior will change significantly.

Problem 2b: Sometimes you need high resolution. For example, you might like to see whether a behavior changes when $\mathbf{n^3}$ is around the size of L1 or L2 cache. So you want lots of data points.
Fallacy: You can get *both* the widest range and the highest resolution. All it takes is creating a clever script and letting your job run over night.
Solution: Running the entire range at high resolution usually doesn't work. At the high end it may only be possible to get a few points. At the low end, high resolution is practically free. Q: So what to do? A: Run some small test experiments and observe! For example, you might want to get a few points each in a few ranges – small, medium, large. Or generate a few points on a log scale (10,100,1000, etc.). And then zoom in on regions of interest, say, around 1000.
Why Ax^2+Bx+C? Why not just "X"? In a few of the labs the loops are indexed by a function of X rather than X. The reason we do this is to give you options with minimal recoding. For example, for each iteration, let X increase by 1. Then
A = 0, B = 10, C = 0 → each iteration the problem size increases linearly (by 10)
A = 0, B = 10, C = 1000 → As the previous, but you have started the range at 1000 rather than 0.
A = 100, B = 0, C = 0 → This gives you a quick way to cover a large range: 100, 400, 900, 1600, etc.
A = 2, B = 10, C = 100 → Starts the experiment at 100 (perhaps to avoid anomolies at small sizes), gets large faster than linearly, but also generates some intermediate data points at the low end: 100, 112, 128, 148, 176, …

When to plot on linear axes, when log axes?
In general, getting a straight line is good. So if execution time is nonlinear with problem size (say x = matrix dimension size in MMM) then a log Y axis is called for.
Also, if you are covering a wide range (100, 1000, 10000) then you should use a log X axis. And if you use a log X axis then you should also use a log Y axis.
Review: On linear axes the slope is the coefficient. On log axes the slope is the exponent and the intercept is the coefficient.

Evaluation of code using Cycles per Element (CPE)
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 (noise isn't random!!)
- 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.

Your tasks for part 4

Use -O1 optimization for this part.

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 psum1and psum2) is provided in **test_psum.c.** Add timing using clock_gettime() and diff() as in **test_clock_gettime.c**; plot your results as discussed in the previous sections.

The main loops select a variety of sizes to test. You should understand this thoroughly since we will be using similar mechanisms through the semester. Here it does so by evaluating the polynomial Ax^2+Bx+C for values of x from 0 to NUM_TESTS-1. Adjust the coefficients A, B and C to make it cover a greater range of sizes. Also, you can increase NUM_TESTSto get more data points.

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.

4b. 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?

4c. 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 psum1and 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)
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
```

Here we are using the shell's **time** command to measure program run time, so the program doesn't need to measure itself. What gets printed? Pay most attention to the "user:" time.

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 "user:" 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. Variables like "quasi_random" and "i" are at offsets from **rbp**, like **-8(%rbp)** or **-24(%rbp)**. There should be some kind of "mul" and "sub" instruction inside the loop. 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 "quasi_random" will solve the problem (how?).

- Try this: modify **test_O_level.c** so that "quasi_random" gets printed out after the loop is done (it is a double, so use the %f format specifier).

- 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 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)? We will re-visit this question in the next assignment.

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 2.0 floating-point operations (FLOPs) per loop and 8.0 Unique Reads per loop. There are also some integer additions, and a type conversion from integer to float — *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] = quasi_random * in[j] + in[j] * in[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.

This example is a ratio of 4.0/2.0 (FLOPs per Unique Read). Repeat this to get data points for ratios covering the range [1/8, 8].

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 does not use quasi_random or any **integer[][]** values, but is multiplying together nine copies of **in[j]** (which takes eight multiplications, one unique read). 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.