Lab 2 -- Memory Optimization
Due Tuesday, February 3rd

Objectives
Learn about and practice using basic memory oriented optimizations in compute kernels. In particular:
- Locality effects in real programs: spatial (neighbor fetches) and temporal (proximate reuse and working set size)
- Two fundamental methods of code optimization: loop interchange and blocking

Reading
B&O Chapters 5.3 and 6.5

Prerequisites (to be covered in class or through examples in on-line documentation)
HW – Basic knowledge cache and memory.
SW – How to program Matrix-Matrix Multiplication (MMM). How to do two basic code transformations: loop interchange and blocking. Effects of these optimizations on memory operation.

Assignment

Preliminaries:
Overall, report (plot) your results and explain them in as much detail as you can especially with respect to the machine you are running on. If something does not work as expected, give a reason (or at least a hypothesis).

Hint: For parts 1 and 4 your runs should be short, just a few seconds. Parts 2 and 3, however, may end up taking a few minutes per data point as you exercise the memory hierarchy with matrices bigger than the cache. For these it is probably a good idea to break your experiments into separate runs always leaving ITERS less than, say, 30. For example, for any trial, start with BASE = 0 and large DELTA so that you try out the whole range. Then "zoom in" on interesting parts by setting BASE to the starting point of where you want to look and also reducing DELTA.

Note: Compilation instructions are in the .c files. You should use –O1 optimization throughout. Please see B&O Chapter 5 for reasoning.

Part 1: Testing code transformations -- Task: Optimize combining data from 2D arrays
Reading: B&O Section 5.3. The code in Part 1 is based on the code example that runs through Chapter 5 and that we will cover in depth next week (basic operations on a 1D array of data). For now, study Section 5.3 and thoroughly understand the code there.
Reading: B&O Section 6.5. Especially be sure that you thoroughly understand the practice problems.
Test code: **test_combine2d.c**. The core is a 2D version of the combine4 function on page 493 (a simplified version of the version in Chapter 5.3). Be sure you understand what is going on before you get started. In the test_combine2d.c file you will find:
- two versions of combine2d with the indices reversed in the second version. Note that the data type (data_t) and operation (OP) are parameterized (as in B&O). You should try out a few different types (int, long int, float, double) and ops (+ and *) to see what happens, but this is not a central part of this lab.
- initialization code. Having real numbers allows the code to be verified. More about this in a future lab.

- calling sequences that step through array sizes. At the top of the code there are three #define'd constants: BASE, DELTA, and ITERS. These give the sizes of the array dimensions to be tried. Note that these are 1D. To get the total number of elements you need to square the length.
- timing capture
- prints/displays

In these experiments you will often be testing a wide range of number, i.e., over many orders of magnitude. You will need to use log and log-log axes to properly interpret your data.

To do -- Overall: run code, capture data, and interpret using a spread sheet. Measure various relationships, try different scales of array sizes, zoom in on interesting transitions in the data.
Data to be plotted versus TOTAL number of elements in the array (log scale) in a scatter plot:
(a) number of cycles for combine2d (log)
(b) number of cycles for combine2d_rev (log)
(c) speedup of combine2d over combine2d_rev (raw)
(d) ratio of number of cycles to number of elements for combine2d, (raw) and
(e) ratio of number of cycles to number of elements for combine2d_rev (raw).

1. Set BASE to 0 and set DELTA and ITERS so that you test a wide range of array sizes. In particular
→ How big can you make the array before you run into execution problems? (What's an "execution problem"? Here we use the informal notion that either the function can't run on your computer or that it is hopelessly slow.)
→ Which function is faster? Roughly by how much?
→ At what array size(s) do the "interesting" things happen in the data? Why? (What's "interesting"? Something unexpected such as a sudden transition or a change of slope from positive to negative or vv.)

2. Zoom in on two narrow ranges of array sizes, one small and one large (by adjusting BASE, ITERS, and DELTA).
→ Use the methods you learned in Assignment 0 to find the number of cycles per element for those two ranges for the two different functions (four values in all).
→ Apply your knowledge of the computer interpret your observations here.

3. You should find that the graphs of quantities (d) and (e) are particularly interesting. Zoom in on the region in (d) where it just becomes level and makes a transition (again, by adjusting BASE, ITERS, and DELTA).
→ (d) should have a "u" shaped behavior (or "v" or "bathtub" depending on your scales). Explain why performance improves for a while (this is not obvious at all and may require that you run some more tests, including writing some new code). Explain why it gets worse (this is more obvious).
→ (e) should have behavior that is more "jumpy." We will look at reasons why in a few weeks. But for now, do you have any qualitative ideas to explain this?
Note: on some systems these effects may be more or less apparent. If you don't see them that's OK, but be sure you are running your code correctly.

Deliverables: For all three questions in this part, hand in your answers and graphs justifying those answers.

Part 2: Use these ideas on a more complex code -- Task: Optimize MMM using loop interchange.
Reading: B&O Section 6.6. Especially be sure that you thoroughly understand Section 6.6.2.
Test code: **test_mmm_inter.c**. As always, start by reading the code. You will see

- three versions of MMM: ijk, jki, and kij. As in Part 1, datatype is a parameter, but again varying it is not a central part of this lab.
- initialization code, calling sequences, timing capture, and prints/displays are as in Part 1.

As in Part 1, you will be testing over many orders of magnitude and so will need to use log and log-log axes to properly interpret your data.

To do -- Overall, run code, capture data, and interpret using a spreadsheet. In particular, measure how the cycles-per-inner-loop relates to matrix size. Try different scales of array sizes and zoom in on interesting transitions in the data. In particular, try to reproduce Figure 6.48 for three of the six combinations, but for a wider range of matrix sizes than in that Figure. Data to be plotted, all versus the matrix size (log scale) in a scatter plot, are performance of the three variations of MMM as determined in cycles in the inner loop (total cycles/number of times the inner loop is run or length cubed).

You will quickly find two problems in generating the data.

- Data are noisy. You are likely to find that while the overall shapes of the curves look like those in 6.48, some points will be way out of scale. The way that B&O deal with this is to generate each point with a series of experiments and find the slope through linear regression (see Lab 0!). Here it is sufficient to "zoom" in on regions of interest for additional runs, rather than having separate runs for each data point. You are also invited to try and find the reasons for these anomalies.
- As matrices get large, certainly when they get larger than 1K on a side, each matrix multiply will take significant time to run (getting up into the minutes). By designing your experiments judiciously you should still be able to run all of your experiments in at most a few hours.

Specific task: ***For each loop permutation*** (ijk, jki, and kij) →

1. How many plateaus are there? To find the answer to this, try a range of matrix sizes going up to at least to length = 2K. Why 2K?
2. For each plateau, what is the number of cycles per iteration? Run new experiments to determine these, zooming in on points in the middle of the plateau.
3. Where are the transitions? Again, run new experiments, this time zooming in on the transitions.

Deliverables: For all three questions in this part, hand in your answers and graphs justifying those answers.

Part 3: Use these ideas on a more complex code -- Task: Optimize MMM using blocking.

Reading: B&O Web Supplement on Blocking (see web page).

Test code: Use test_mmm_inter.c as a template, create **test_mmm_block.c** using the blocked version of MMM (from the Web Supplement). Parameterize your code so that you can vary the block size as well as the matrix size.

To do: Use the methods that you learned and practiced in Parts 1&2 to find the following:

1. The benefit of blocking as a function matrix size.
2. The optimal block size as a function of matrix size. It should be sufficient to test exponential variations, e.g., 25, 50, 100, 200, etc.

Ideally you would run a wide range of matrix and block combinations, but again through judicious design of experiments you should be able to cut this way down and still get good answers. Please note: if you run all combinations this will take a very long time.

Deliverables: Hand in your new code, a description of your experiment, your graphs, and your answers.

Part 4: Use these ideas on an entirely new application. Task: Optimize Matrix Transpose.

The idea here is to write some of your own code to practice using loop interchange and blocking.

Test code: Use test_combine2d.c as a template, create **test_transpose.c**. Parameterize your code so that you can vary the block size as well as the matrix size.

To do:

1. Implement test_transpose.c as described.
2. Examine and ij and ji loop permutations over a wide range of matrix sizes as in Part 1.
3. Block the code. How did you do this? Why should this help? Does this help? For what block sizes?

FYI: Problem 6.46 asks you to do something similar based on the code shown below. You don't have to use it -- in fact you may want to reuse as much of the code from test_combine2d.c as you can. Also, Problem 6.46 asks you to use "all optimizations possible." For now, however, you only need to look at cache optimizations. We will revisit transpose in a few weeks!

```
void transpose(int *dst, int *src, int dim)
{
  int i,j;
  for (i = 0; i < dim; i++)
    for (j = 0; j < dim; j++) dst[j*dim+i] = src[i*dim+j];
}
```

Deliverables: Hand in your new code, a description of your experiments, your graphs, and your answers.

Part 5: QC

How long did this take?

Did any part take and "unreasonable" amount of time for what it is trying to accomplish?

Are you missing skills needed to carry out this assignment?

Are there problems with the lab?