EC527: High Performance Programming with Multicore and GPUs
Lab 1 — Memory Optimization

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** (on Blackboard in Course Documents > Readings)

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

Hardware – Basic knowledge of cache and memory.
Software – 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 part 2 (matrix combine) and 5 (matrix transpose) your runs should be short, just a few seconds. However, parts 3 and 4 (matrix multiply) may end up taking a few minutes per data point as you exercise the memory hierarchy with matrices bigger than the cache. For these you may wish to break your experiments into different runs, with different settings for `A`, `B`, and `C`. For example, start with values that cover the whole range; then "zoom in" on interesting parts by setting `A` to 0 (so it's a linear distribution using `B` and `C`, which you would also adjust appropriately).

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

Part 1: Memory bandwidth profile of your system

Get the files "`mem_bench.c`" and "`multicore_stream.pl`". Make sure the script "multicore_stream.pl" is executable with the command:

```
chmod +x multicore_stream.pl
```

Then run it:

```
./multicore_stream.pl
```

This is a Perl script that compiles the C program, then runs it many times to perform memory-copy speed measurements. Some of the measurements involve running multiple copies of the program simultaneously.

The data are output in "comma-separated" format. Like last week, get this output into a CSV file to import into a spreadsheet of your choice, or search online for instructions on how to copy/paste data into a spreadsheet (the answer might involve a menu command like **Edit > Paste Special...** or **Data > Text to Columns...**).

Make a chart presenting the data, using:

- Horizontal axis: memory size in bytes (log scale)
- Vertical axis: bandwidth (*not* "BW/task") in bytes per second (log scale)
- Please *show all the data in a single chart* by using different colors or symbols (triangle, square, X, O) for 1, 2, 4, ... tasks.
- Label the axes and include a legend (telling which color or symbol means 1 task, 2 tasks etc.)

1a. When running just 1 task, the total bandwidth is the same as the "BW/task" (bandwidth per task). What memory size(s) give the highest bandwidth? (If there are several sizes that are close to giving the highest, indicate this.)

1b. When running just 1 task, what memory size(s) give the lowest bandwidth? What is the ratio between the highest and lowest bandwidths across all the 1-task tests?

1c. How many tasks and what memory size gives the highest total bandwidth? (If there are several combinations that are close to giving the highest, indicate this.)
➔ If the answer to question 1c is more than 1 task, the bandwidth achieved is probably larger than in questions 1a/1b. This is usually because the memory bandwidth capability of a single core is less than that of the entire processor (chip) which contains multiple cores.

1d. For the largest memory size, look at the bandwidth achieved when running just one task. When using that largest memory size, how many tasks can you run at the same time on this machine, and get *close* to the same bandwidth per task as what you get with just one task?

Note: If the data don't seem to make sense, the machine might have been busy during part of the test. If you are on a lab machine, remember that you can use the commands "`w`" and "`who`" to find out if others are actively using it. It may help to run the `multicore_stream.pl` script again and compare the numbers to what you got the first time.

Part 2: Testing code transformations — Task: Optimize combining data from 2D arrays

Reading: B&O Section 5.3 (in one of the PDFs, perhaps the one called "BO_chapter5_2.pdf"). The code in this part 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 from B&O 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 `#define`'d constants: `A`, `B`, `C`, and `NUM_TESTS`. These control the sizes of the array dimensions to be tried. Note that these are the row length, and are also used for the height (number of rows) of the 2D arrays. To get the total number of elements you need to square the row length.

• *timing capture*

• *prints/displays*

In these experiments you will often be testing a wide range of numbers, i.e., over many orders of magnitude. You will need to use a log axis (on one or both axes) to properly interpret your data.

To do — Overall (details below): run code, capture data, and interpret using a spread sheet. Measure various relationships, try different scales of array sizes, gather more data for array sizes where there are interesting transitions in performance.

Some data that I have found useful to capture (or generate from captured data) are:

(a) number of cycles for `combine2d` (log scale)

(b) number of cycles for `combine2d_rev` (log scale)

(c) speedup of `combine2d` over `combine2d_rev` (linear scale)

(d) ratio of number of cycles to number of elements for `combine2d` (linear scale)

(e) ratio of number of cycles to number of elements for `combine2d_rev` (linear scale)

When plotting a wide range of data, the horizontal axis should be the *total* number of elements in the array (height times width), using a log scale.

2a. Set `A`, `B`, and `C` 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"? Either it prints an error and won't run at all, or it is hopelessly slow. How do you find how big something is without knowing how big it is? Try successive doubling!)*

➔ 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 a zigzag shape like* /\/\/\/\ or _/__/__/_*.)*

2b. Get data for two narrow ranges of array sizes, one small and one large (by adjusting `A`, `B`, and `C`).

➔ 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). Remember that these are 2D arrays: the number of elements is the product of the height and width: compute the "number of elements" correctly.

➔ Applying your knowledge of the computer, interpret your observations here.

2c. (If you don't see these effects don't spend too much time, just continue on to the next Part of this lab.) 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 `A`, `B`, and `C`).

➔ *(d)* should have a "u" shaped behavior (or "v" or "bathtub" depending on your scales). If you do not see such a thing, make sure you are including data for very small sizes (10x10 array size, or even smaller) *as well as* large sizes (2000 x 2000 or larger). 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 as the size gets very large (this is more obvious).

➔ *(e)* should have behavior that is more "jumpy." If you see a periodic pattern like this: /\/\/\/\ or like this: _/__/__/__ it is most probably caused by cache utilisation of the program's access pattern,

which depends on `(row_length*sizeof(data_t))` modulo `(cache_size/associativity)`, using the stats for whichever level of cache (L1, L2, ...) is not big enough to handle the data access pattern. If the jumpiness annoys you, choose `BASE` and `DELTA` in such a way that `BASE+i*DELTA` is always an odd number. (This works because odd numbers are relatively prime to powers of 2).

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 3: 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 the previous Part, the data type is a parameter (called `data_t`, initially defined as `double`), but, again, varying it is not a central part of this lab.
- initialization code, calling sequences, timing capture, and prints/displays are as in the previous Part.

Again, 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-innermost-loop relates to matrix size (or number of operations, N^3). Try different scales of array sizes and zoom in on interesting transitions in the data.

In particular, try to reproduce Figure 6.48 (see lecture slides) for three of the six combinations, but for a wider range of matrix sizes than in that Figure. Plot performance of the three variations of MMM as a function of matrix size. To match figure 6.48, the performance (vertical axis) *must be* determined as **cycles per innermost loop iteration** (total cycles divided by the total number of times the code inside the innermost loop is run); and the matrix size (horizontal axis) should be on a log scale (they used row length, but rows × columns would be okay too).

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 (like in 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 tasks:

3a. ***For the first loop permutation*** (ijk) ➔

i. How many plateaus are there? To find the answer to this, try a range of matrix sizes from 10x10 going up to at least to 2Kx2K. (If you are wondering *"why 2K?"*, compute how many bytes it takes for three 2Kx2K matrices, and compare to the L3 cache size of your machine.) Combine your data together into a single graph. Set the axis types and scales so that you do not squeeze all the small sizes together into a corner.

ii. For each plateau, what is the number of cycles per innermost loop iteration (*as defined above*)? Run new experiments to determine these, averaging results from several array sizes in the middle of the plateau (no new graph needed for this).

iii. Between plateaus are the "transistions". Expressing your answer in terms of the matrix sizes (horizontal axis), where do the transitions occur? You may need to test more sizes to see where each transition starts and ends.

3b. ***Repeat i., ii., iii. for the loop permutation*** "jki"

3c. ***Repeat i., ii., iii. for the loop permutation*** "kij"

Deliverables: For everything in part 3, hand in your answers (with graph for part i.) and any other data you used to get those answers.

Part 4: Use these ideas on more complex code — Task: Optimize MMM using blocking.

Reading: B&O Web Supplement on Blocking (Course Documents > Readings > Bryant & O'Halloron Chapter 6 parts > BO_CH6_blocking.pdf).

Note in particular the blocked version of MMM called "`bijk()`" shown in their figure 1.

Test code: Use `test_mmm_inter.c` as a template, create **`test_mmm_block.c`** by starting with your working "ijk" version and adding two more loops like the way B&O did. *Do not use textbook code directly, it will not work as written and might not even compile!* Instead, you must understand what B&O are doing and figure out how to do the blocking in your own code. Parameterize your code so that you can vary the block size as well as the matrix size (note that all block sizes must divide evenly into the matrix sizes). You will end up with five nested loops and if you do it right, the code inside the innermost loop will run exactly the same number of times as the normal ijk code, because it is doing the same calculations, just in a different order.

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

4a. The benefit of blocking as a function of matrix size.

4b. The optimal block size as a function of matrix size. It should be sufficient to test exponential variations, e.g., 8, 16, 32, 64, etc. For each block size you try, remember to set your coefficients `A`, `B`, and `C` so that all tested sizes are a multiple of your block size.

There are a lot of combinations of matrix size and block size, but if you try them all it will take too long. Through judicious design of experiments, you should be able to cut this way down and still get good answers. (Also, you are *not* being asked to try blocking for jik or kji, only for ijk!)

Extra Credit: Serious MMM optimization

4c. For your best MMM, compute the FP utilization.

4d. While the best blocked MMM is probably better than the best non-blocked, the difference might be somewhat less than the analysis predicts. What's wrong with the analysis? How would the application have to change for blocked code to show more benefit?

4e. There is probably an easy way to enhance the distinction between blocked and non-blocked methods. Make the changes and try them out.

Deliverables: Hand in your new code, a description of your experiment(s), your graph, and your answers.

Part 5: 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.
To do:

5a. 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.

5b. Examine and ij and ji loop permutations over a wide range of matrix sizes as you did for MMM (including small enough to fit in L1 cache, and big enough to *not* fit in L3 cache).

5c. Block the code. How did you do this? Why should this help? Does this help? For what block sizes?

FYI: In the Bryant & O'Hallaron textbook on page 641 (not in any of the PDFs on Blackboard) there is a Problem 6.46 that asks you to do something similar based on the code shown below. You don't have to use this code — in fact you may want to reuse as much of the code from `test_combine2d.c` as you can. Also, that same Problem 6.46 asks you to create *"a transpose routine that runs as fast as possible."* For now, however, you only need to look at cache optimizations. We will revisit transpose in a few weeks!

```
/* Transpose example from B&O textbook page 641 */
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 6: Quality Control
6a. How long did this take?
6b. Did any part take an "unreasonable" amount of time for what it is trying to accomplish?
6c. Are you missing skills needed to carry out this assignment?
6d. Are there problems with the lab?