The purpose of this module is to understand how different code practices with regards to loops will improve code efficiency. Students can compile and run each example, with varying levels of compiler optimization options.

These activities could be used either as exercises for the students, in which they are responsible for compiling and running the code and describing their observations, or they could be used as examples within the context of a lecture.

A short set of lecture slides suggesting order of topics and describing each code is provided.

The concepts tested are loop invariant code, function inlining, strength reduction, positioning of conditions inside or outside of loop structure, as well as a variety of examples of out of order memory access using striding of arrays and changing order of access of a 2-D array.

Students will see varying levels of success with each of these in addressing the efficiency improvements simply by enabling compiler options, whereas others will not. In particular, students will see that the single most important aspect of loop optimization is reducing the use of out of order memory access wherever possible, due to the increased number of cache line reloads that are required. Strength reduction for the pow command with integer powers is also a noticeable improvement in many cases, and it is important to stress to students the computational cost of most built in mathematical functions.

Each of the codes in their current form use dynamic memory allocation, as an increased number of virtual resources available to students have restrictive limits on stack memory, leading to unexpected and erroneous behaviour when executing codes with large statically allocated arrays on resources with less memory.

For students with access to resources capable of keeping large arrays as statically allocated memory in stack (typical of work nodes on a professional cluster, for example, but possibly not for a student using a Colab instance as their compute instance), it can also be interesting to replace the dynamic memory allocation with static memory allocation to explore the degree to which optimizing compilers are better able to optimize static as opposed to dynamic array access.

Also, for every one of these examples, optimization is a moving target. The optimizing compilers of today are substantially better than those of a decade ago, and even more so than compiler versions from the early days of HPC. Students who have access to multiple versions of compilers might choose to explore how compiler brand (gcc, intel, portland) differs in performance as well as version (a stock gcc instance at an older stable version compared to a recently compiled version with better optimization and more advanced features).

The key message to the student is:

- Exploit memory locality. Cache misses cost time. Page faults cost time. Don't even talk to me about the translation lookaside buffer. If you can as much as possible access items that are close to each other in memory, do so. Don't use a linked list or a binary tree if you don't need to. Don't use an index array if you don't need to. Don't go through an array with non unit stride if you don't need to. And make sure in multi-dimensional arrays that your inner loop steps through your array contiguously through memory, unless you can't. The algorithm comes first of course, an inefficient order N log N algorithm beats the most well tuned order N squared algorithm any day of the week, but once you know your algorithm, clean it up for memory locality as much as possible.
- Move invariant commands out of loops. This one seems obvious, but it is easy to forget, or to not realize something is invariant. It's particularly easy to miss with nested loops. Say for example you have a (N by M) 2-D array, stored in memory as a 1-D array, and indexed as A[i*M+j]. You make many references to the expression i*M+j, so inside your loop you use a command "index_i_j = i*M+j". Assuming that the inner loop is over j (see the last point), you can perform the i*M portion of this once at the beginning of the j loop, or if you know that you have no issues with array padding you might even just keep a single counter running and incrementing by one each time in the inner loop and avoid the multiplies altogether. Move anything out of the loop or into the outer loops that you can.
- Function calls are expensive. If you have to choose between passing an array to a function that operates on the array, or sending each element of an array to a function that operates on the element value, pass the array. Get the function call outside of the loop.
- Conditions and loop overhead are expensive. Keep your loops fat (more than one command per time through the loop). If you need a skinny loop, keep it simply expressed, so that the compiler can try tricks like loop unrolling.
- Branching inside of a loop makes optimization difficult for the compiler. Optimizing compilers can try to do a lot with loops, but the more you do that causes your loop to not fit set patterns, the less likely it is that the compiler will optimize it. If you have a choice, keep the conditions outside of the loop.
- Dynamic memory allocation takes time, and makes it harder for the compiler to optimize. The difference you get from static memory allocation will vary a lot by compiler, and may not be worth it considering the great advantages to dynamic allocation.

Many of these examples are optimization problems that the compiler can try to improve. Depending on how the student compiles, they may get different results. It is recommended to try running both with -O0 (no optimization) and -O2 (standard release optimization) to see which of these issues can be resolved by the compiler and which cannot.

Students should be careful about their optimization setting when compiling the examples.

Note, each of these examples may run different on different machines. In particular, references to static memory allocation have been removed from a prior version of this example due to the greater restrictions on stack memory in many of the virtual machines that students may encounter. The example comparing static to dynamic memory allocation has been left in its original form, but its ability to run without a segmentation fault may depend on the stack size limits and available memory on the systems being used.