Loop Optimization

Principles of Loop Optimization

- Access memory in order when possible
  - Reading memory from cache is much faster than reading from RAM
  - Go through arrays in sequence rather than out of order or in large strides
  - Be aware of your rapidly changing index in 2D arrays (and keep your 2D arrays contiguous instead of using lists of lists).
- Minimize operations in loops
  - Look for loop invariant code
  - Look for opportunities to replace complex operations with simple ones
- Minimize overhead in loops
  - Move conditions outside of loops if possible
  - Pass arrays to functions instead of calling functions on elements of an array if possible
  - Replace simple functions with code. #define statements can keep your code clean when doing this.

Array access examples

- arraystride.c: Order of loops over 2D array
  - This example allows you to loop through a 2D array changing elements, either in row-column order or in column-row order.
  - Knowing which is your rapidly changing variable, and looping over your rapidly changing variable, will allow for more efficient memory access.
- unitstride.c: Access array in order or in large steps
  - This example will allow you to access and set elements of an array either in steps of 1 (a[0], a[1], a[2], etc.) or in steps of some stride length (a[0], a[10], a[20],…).
  - Accessing memory out of order will require memory to have to be read into cache more often, and will slow down your code.

Minimizing operations in loops

- loopcondition.c
  - This example will allow you to perform a loop operation with a different operation on the first and last items of the loop. The condition can either be inside of the loop or outside.
- loopinvariantcode.c
  - This example will have loop invariant code, either inside or outside of a loop.
  - Calculate your loop invariants outside of the loop, not in.
- strengthred.c
  - This example will compare the efficiency of pow(x,2) to x*x.
  - Special cases of complex functions that can be simplified should be.

Minimizing overhead

- inlining.c
  - This example will compare the efficiency of replacing a small function with equivalent code.
  - While hardcoding repetitive calls can make for difficult to read code, and can violate software engineering principles, you can simplify this in C/C++ with the use of the #define statement, and in many cases the compiler can do it for you.
- arrayfunc.c
  - This example will compare passing an array to a simple function and looping within that function, compared to applying a simple function to each element of the loop.
  - Don’t introduce unnecessary function overhead.