

MP0

by Abhishek Johri (johri3)



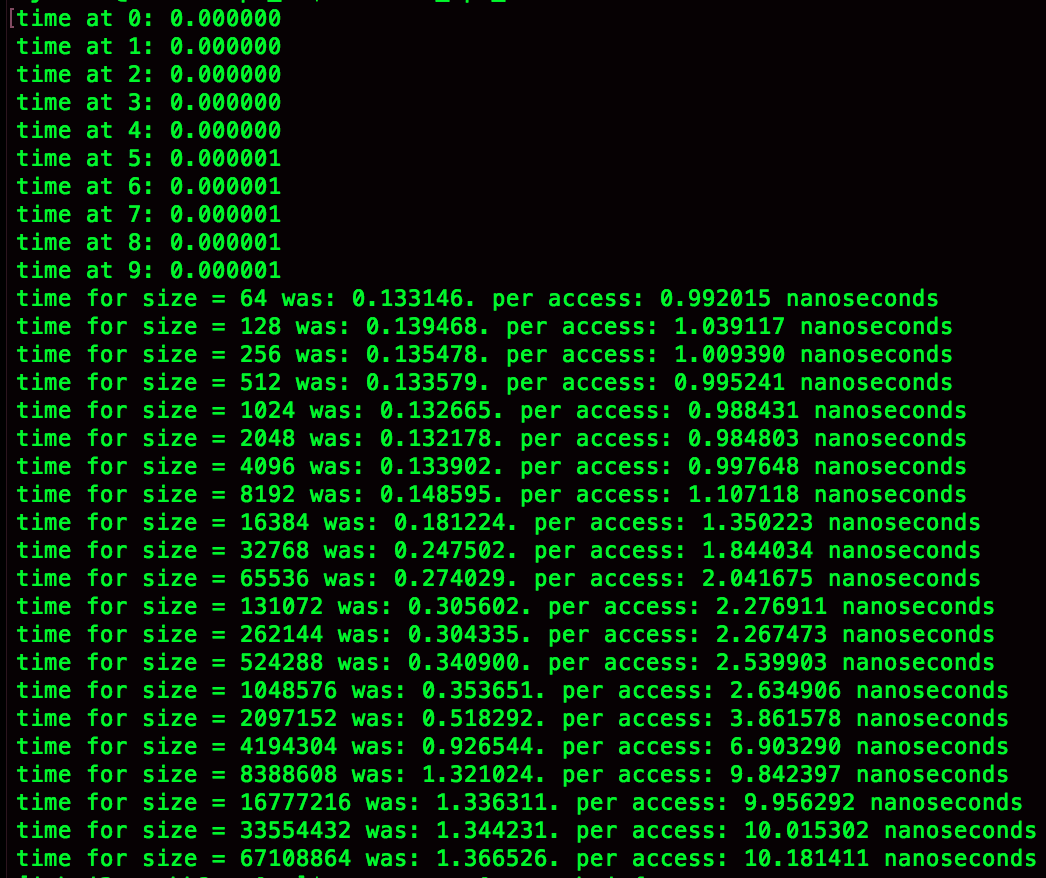
January 30, 2017

University of Illinois

CS 484

**Part A:**

Data:

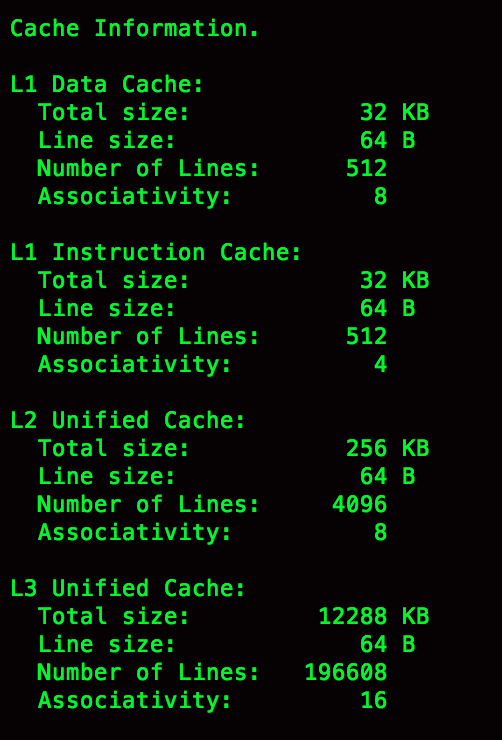


Graph:

\*X-axis is size and Y-axis is time

The general trend shown above is that as the size increased the time it took to do the same functionality increases up till a certain point. This trend simulates a log function, as it has a horizontal asymptote. This trend shows how effectively the entire system uses the cache. At smaller sizes the data needed by the program do not use up the entire cache or even the block size of the cache. As the size increases the program will have to access more than what the cache can contain. Because of this the program will have to fetch data from outside the cache, which is a slower process, thus increasing the time as shown.

Here is data regarding the amount of cache that we have:



Now for some calculations:

Each element in the array that we have is 4 bytes (size of int \*) This means that the size of our N sized array will be N \* 4 as N increases. In our data you can see a spike in time starting from when N = 1048576 to when N = 8388608. So we can see that the size of our data at N = 1048576 is 4,194,304 bytes. This is 4MB which is much greater than our L1 Data Cache, L1 Instruction Cache, and L2 Unified Cache. Because the data can not all be stored on the cache it means that you will need to access it from somewhere other than the cache. This causes the increase in time that we see. For when N = 8388608 we get that the size of our data will then be 33,554,432 bytes. This is roughly about 33MB which will now be greater that even our L3 Unified Cache. At this point it means that we are not able to use spatial locality to our advantage and will constantly be overwriting the cache. This causes our runtimes to gradually increase from here as shown by the graph reaching an asymptote.

**Part B:**

-O0 Optimization notes

Reduce compilation time and make debugging produce the expected results. This is the default.

did nothing so there was no loop optimization

-O1 Optimization notes

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

loop optimization was disabled for this program.

-O2 Optimization notes

Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code.

test1 was not vectorizable

test2 both loops were vectorizable

test3 both loops were vectorizable

test4 loop was not vectorizable

-O3 Optimization notes

Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-loop-vectorize, -ftree-loop-distribute-patterns, -fsplit-paths -ftree-slp-vectorize, -fvect-cost-model, -ftree-partial-pre, -fpeel-loops and -fipa-cp-clone options.

test1 was not vectorizable

test2 both loops were vectorizable

test3 both loops were vectorizable

test4 loop was not vectorizable

When Optimization flag is –O0

time taken for test1 = 0.0039025306701660

time taken for test2 version1 = 0.0042590618133545

Time taken for test2 version2 = 0.0034782886505127

Time taken for test3 version 1 = 0.0201032876968384

Time taken for test3 version2 = 0.0040760278701782

time taken for test4 = 0.0199074983596802

When Optimization flag is –O1

time taken for test1 = 0.0000000000000000

time taken for test2 version1 = 0.0013990640640259

Time taken for test2 version2 = 0.0013006210327148

Time taken for test3 version 1 = 0.0051163196563721

Time taken for test3 version2 = 0.0013778924942017

time taken for test4 = 0.0055504798889160

When Optimization flag is –O2

time taken for test1 = 0.0000000953674316

time taken for test2 version1 = 0.0011168479919434

Time taken for test2 version2 = 0.0012752771377563

Time taken for test3 version 1 = 0.0005084991455078

Time taken for test3 version2 = 0.0005094528198242

time taken for test4 = 0.0030553340911865

When Optimization flag is –O3

time taken for test1 = 0.0000000000000000

time taken for test2 version1 = 0.0011249303817749

Time taken for test2 version2 = 0.0012644052505493

Time taken for test3 version 1 = 0.0004981756210327

Time taken for test3 version2 = 0.0005082368850708

time taken for test4 = 0.0030636787414551

Analysis:

From the data above we can see that the –O2 and –O3 flags have the best performance compared to the –O0 and –O1 flags. The first difference that I noted between these pairs of flags was that –O2 and –O3 flags vectorized the program to speed up the process. The difference between –O0 and –O1is that –O1 will allot the space that is needed during compilation rather than the programming doing it on it’s own as it would in –O0. The difference between the –O2 and –O3 optimization flags is that –O3 enables more flags which aren’t used in our example. This is why the runtimes between –O2 and –O3 are close.