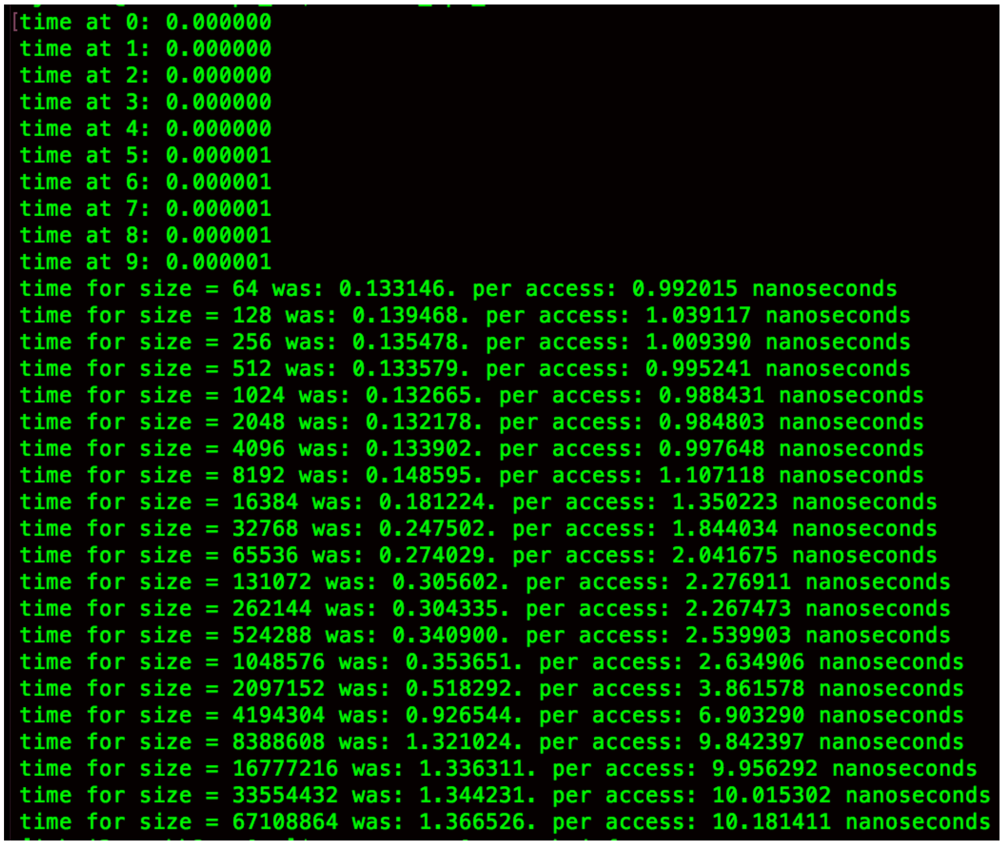
MP0

by Abhishek Johri (johri3)

JANUARY 30, 2017
UNIVERSITY OF ILLINOIS
CS 484

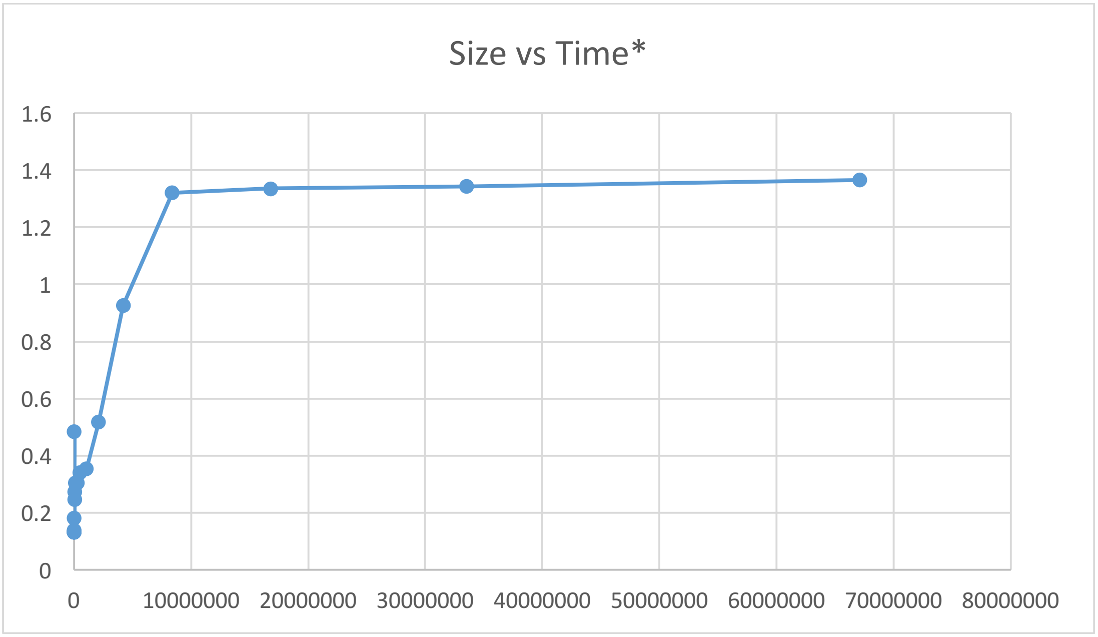
Part A:

Data:



```
[time at 0: 0.000000
 time at 1: 0.000000
 time at 2: 0.000000
 time at 3: 0.000000
 time at 4: 0.000000
 time at 5: 0.000001
 time at 6: 0.000001
 time at 7: 0.000001
 time at 8: 0.000001
 time at 9: 0.000001
 time for size = 64 was: 0.133146. per access: 0.992015 nanoseconds
 time for size = 128 was: 0.139468. per access: 1.039117 nanoseconds
 time for size = 256 was: 0.135478. per access: 1.009390 nanoseconds
 time for size = 512 was: 0.133579. per access: 0.995241 nanoseconds
 time for size = 1024 was: 0.132665. per access: 0.988431 nanoseconds
 time for size = 2048 was: 0.132178. per access: 0.984803 nanoseconds
 time for size = 4096 was: 0.133902. per access: 0.997648 nanoseconds
 time for size = 8192 was: 0.148595. per access: 1.107118 nanoseconds
 time for size = 16384 was: 0.181224. per access: 1.350223 nanoseconds
 time for size = 32768 was: 0.247502. per access: 1.844034 nanoseconds
 time for size = 65536 was: 0.274029. per access: 2.041675 nanoseconds
 time for size = 131072 was: 0.305602. per access: 2.276911 nanoseconds
 time for size = 262144 was: 0.304335. per access: 2.267473 nanoseconds
 time for size = 524288 was: 0.340900. per access: 2.539903 nanoseconds
 time for size = 1048576 was: 0.353651. per access: 2.634906 nanoseconds
 time for size = 2097152 was: 0.518292. per access: 3.861578 nanoseconds
 time for size = 4194304 was: 0.926544. per access: 6.903290 nanoseconds
 time for size = 8388608 was: 1.321024. per access: 9.842397 nanoseconds
 time for size = 16777216 was: 1.336311. per access: 9.956292 nanoseconds
 time for size = 33554432 was: 1.344231. per access: 10.015302 nanoseconds
 time for size = 67108864 was: 1.366526. per access: 10.181411 nanoseconds
```

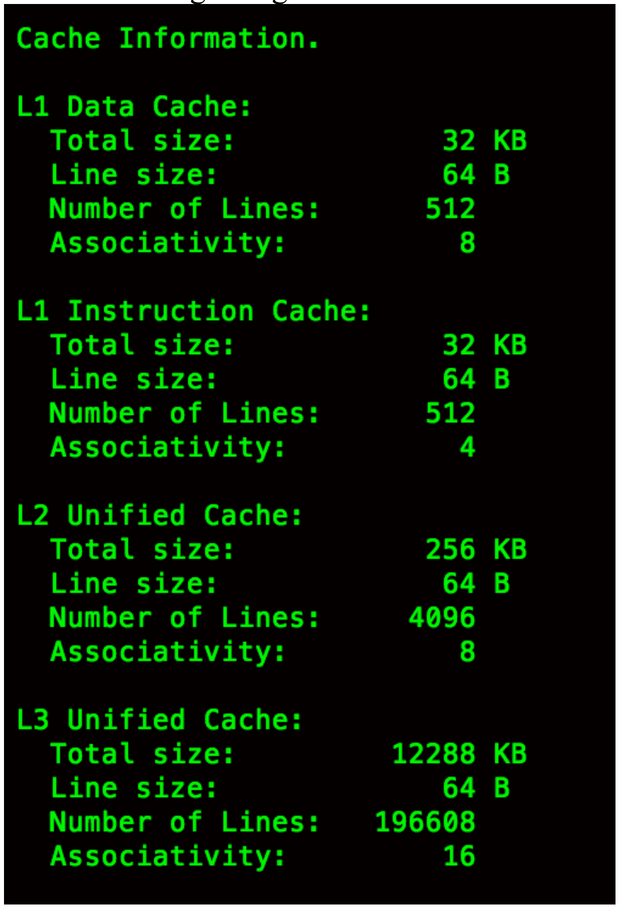
Graph:



*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:



```
Cache Information.

L1 Data Cache:
  Total size:           32 KB
  Line size:            64 B
  Number of Lines:     512
  Associativity:         8

L1 Instruction Cache:
  Total size:           32 KB
  Line size:            64 B
  Number of Lines:     512
  Associativity:         4

L2 Unified Cache:
  Total size:          256 KB
  Line size:            64 B
  Number of Lines:    4096
  Associativity:         8

L3 Unified Cache:
  Total size:        12288 KB
  Line size:            64 B
  Number of Lines:  196608
  Associativity:        16
```

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.