Exploring Processor Caching - A CS243 Project

Caleb Nelson 74009810
Dec 13, 2017

Project code and data available at: https://github.com/calebnelson/cs243

Summary:

CPUs have multiple levels of caching to accelerate performance. This project explores the performance impact on two example application loads (summing columns in a matrix, multiplying two matrices into a third) on problem sizes that fit then exceed those caches, across four different data types (char, int, long, float), across three different platforms (MacBook, PC Laptop, Raspberry). I also explore the impact of alternate approaches to matrix multiplication to make better use of caching. Key observations include clear performance drops when data sizes exceed cache sizes, consistent anomalous behaviors (where smaller problems can take longer than larger problems), and significant performance gains from processing matrices in blocks that maximize cache usage.

Details:

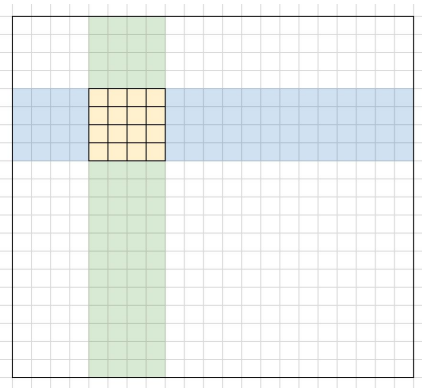
In order to understand why optimizing for memory caches is important, it's necessary to know what they are and why they exist. In all forms of memory, there is an inherent trade off between the size of the memory and the latency, or speed at which it can be accessed. However, modern technologies, advancements, and innovations hunger for both larger memory capacity and faster calculation. Thus, processor manufacturers have included memory caches of multiple different sizes on their processors, so that the lowest level operations can be kept fast while increasing the capacity for more complex calculations.

Different sized caches are designated by a capital L and then an integer, with L1 caches being the smallest and fastest, with L3 (and in some processors, even L4) caches being the largest and slowest. L1

caches are usually split into instruction caches (designated L1i) and data caches (designated L1d). This separation and specialization help increase efficiency even more where it is needed most. Both L1i caches and L1d caches are managed by an L2 cache, which stores both instructions and data. Each core of a processor has its own L1 and L2 caches, and each core of a processor will often share one large L3 cache, which handles distributing instructions among all the cores. If an operation is too big to fit on the L3, the processor will have to keep loading and unloading data from main memory, which causes significant slowdown.

For the first part of this project, I decided to investigate whether or not I could see the transition between when an operation could be completed entirely on the L3 cache and when it spilled over into main memory. In order to do this, I created two test programs, the first of which adds up all of the columns of a large matrix and adds the totals to another row added at the end, and one that multiplies two square matrices together and stores the result in a third matrix, where all three matrices are contiguous in memory. For both tests, a buffer size was passed in, and the matrices were constructed to fill up as much of the buffer as possible. For the sum test, the matrix was constructed large enough to almost fill the buffer entirely, leaving room for the extra row at the end. For the multiplication test, the buffer was split into thirds and square matrices of the largest possible size were constructed in the first two thirds, with the last third functioning as a repository for the resultant matrix. For the processing functions, I used templates that could instantiate different data types, and collected data on four types (char, int, long, float) for each test. I ran each of these 8 tests on three different machines - a Macbook Pro, an ASUS Laptop, and a Raspberry Pi. Each trial was run for 20 buffer sizes from 0.5MB to 10MB in 0.5 MB increments, and each individual trial was 20 times, giving a total of 20*20*8*3 = 9600 data points. These data points were consolidated into a single csv, and the mean and standard deviations for the length of time was calculated for each of the tests. In addition to the total time for each test, I also estimated the number of data operations (e.g. multiplies but not counting local variable arithmetic to calculate array indexes) and analyzed the number of operations per unit time to simplify comparisons. In the attached HTML file are graphs showing the results of the trials. In each plot, the X-axis

is the buffer size from 0.5 to 10 MB, and the Y-axis shows either total time, or number-operations/unit-time. Each graph has three lines - the black line is the actual graph, the red line is the mean + 1 standard deviation, and the blue line is the mean - 1 standard deviation, so if the three lines are close together, the variance is low, and if they drift apart, the variance increases. I expected non-linear increases in total time and big decreases in ops/time as the processor caches overflow requiring data to be repeatedly fetched from main memory, but unfortunately, save for a few cases, it doesn't work out quite so nicely. This is possibly because of how the compiler optimizes code like this, and because of other operations the operating system or other programs might be running. On the ASUS Laptop, the graphs were especially strange, most likely because it was running Windows 10 so I had to compile and run the code on cygwin, which undoubtedly introduced its own quirks into the architecture and getting us farther away from being able to benchmark the processor caches directly. Nethertheless, we can see this phenomena in a few of the graphs, such as the one for summing up matrices of ints, floats, or longs on the Macbook Pro.

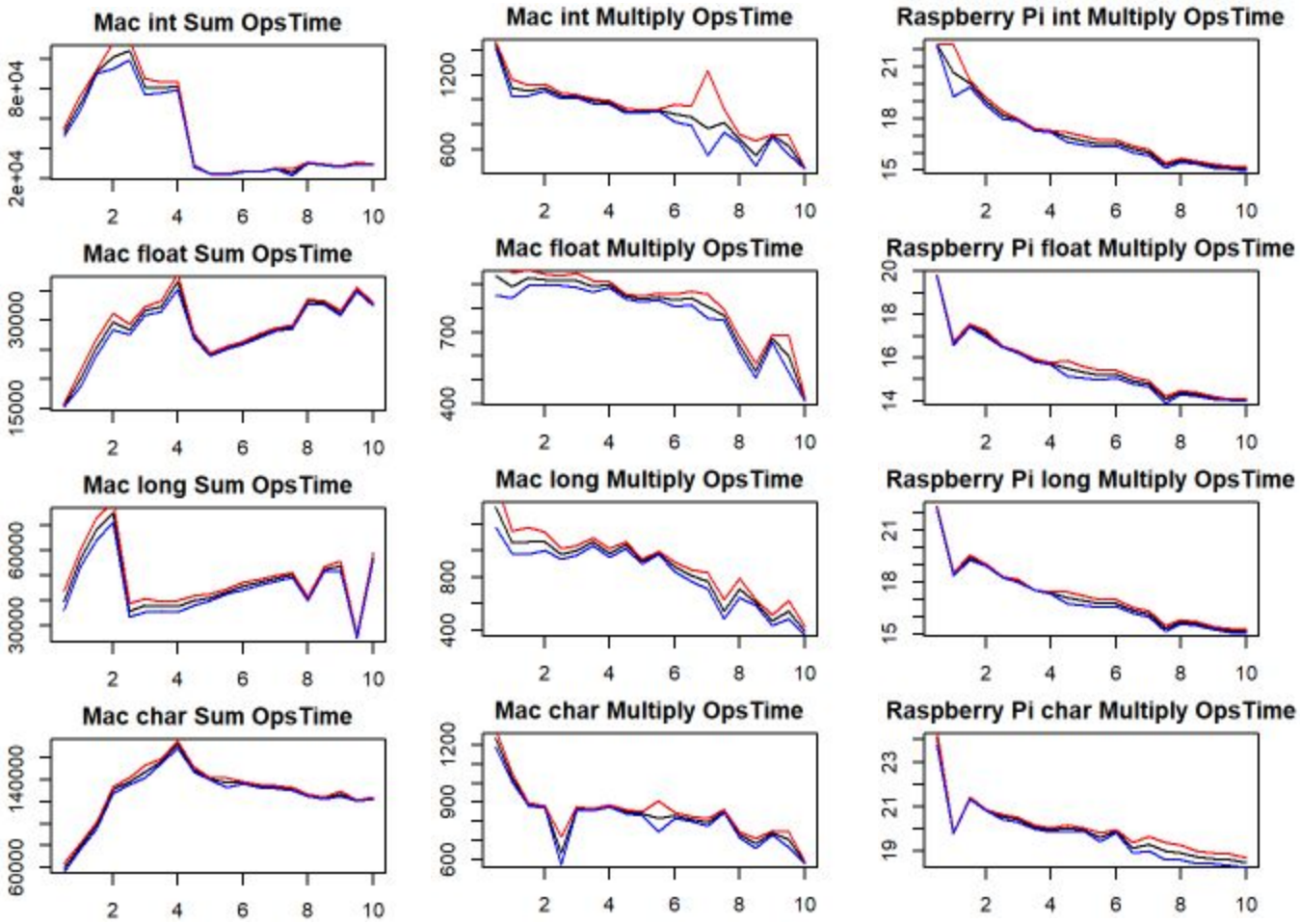


For the second part of this project, I explored alternate methods for matrix multiplication that made better use of the processor cache. Given two square matrices (A, B) with dimensions (NxN) being written into another square matrix (R), we ultimately have to multiply each row in the the first matrix, by each column in the second matrix to get each value in the result matrix. Using a naive `{ for r in rows { for c in cols { … } } }` flow, we would expect to load each row from A once where it would most likely stay in the cache, and then load each column from B once for every row. If B does not fit in the cache, we would expect to have to reload each column in B again from main memory for every row in A, resulting in N * N loads of B's columns. If instead we process the matrix in blocks, (e.g. fully calculate from rows(0:63)|cols(0:63), then rows(0:63)|cols(64:127) then ...) where all the data for the corresponding rows from A and the columns from B fit in the cache, then we will only need to fetch each row or column from main memory into the processor cache once per block.

If we divide the matrix into say 16 by 16 blocks (e.g. a 1k by 1k matrices processed in blocks of 64x64), then we need to cache each row in A up to 16 times, and each column in B up to 16 times. That would be a total of (16*N + 16*N) loads, which is significantly smaller than (N * N) loads from the non-blocked flow, and we'd expect to see significant performance increases on large matrices that no longer fit in the processor cache. The graphs of the results of this new multiplication flow (Mac only for each data type) are attached in the HTML file. The graphs use three different block sizes -- a block size of 1, which is equivalent to the unblocked algorithm, a block size of 32, and a block size of 64. As expected, as the buffer size gets larger and larger, which represents larger matrices being multiplied, the blocked implementation operates much faster.

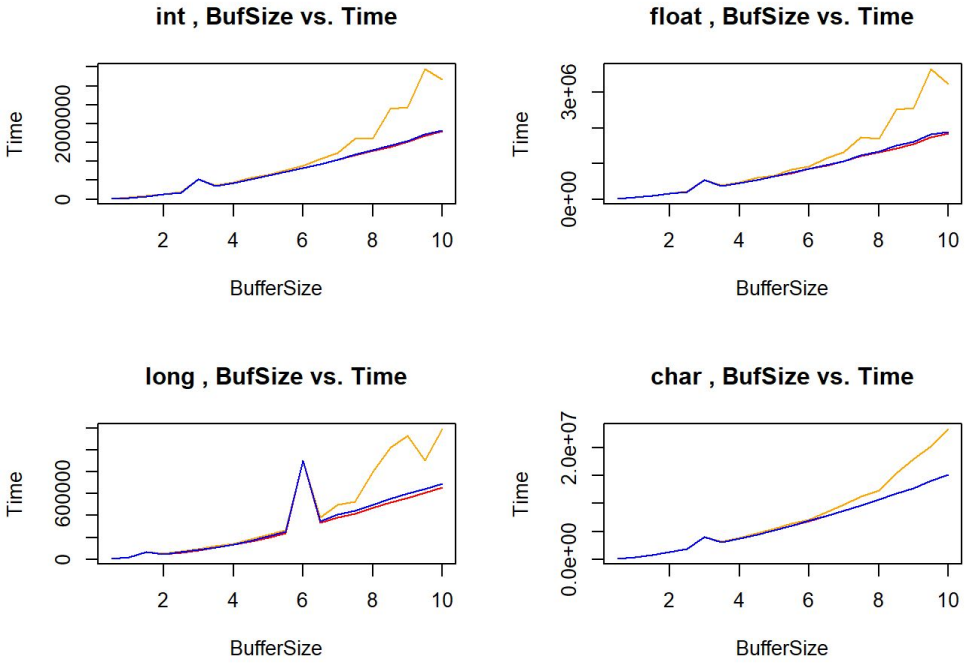
Selected interesting graphs and observations:

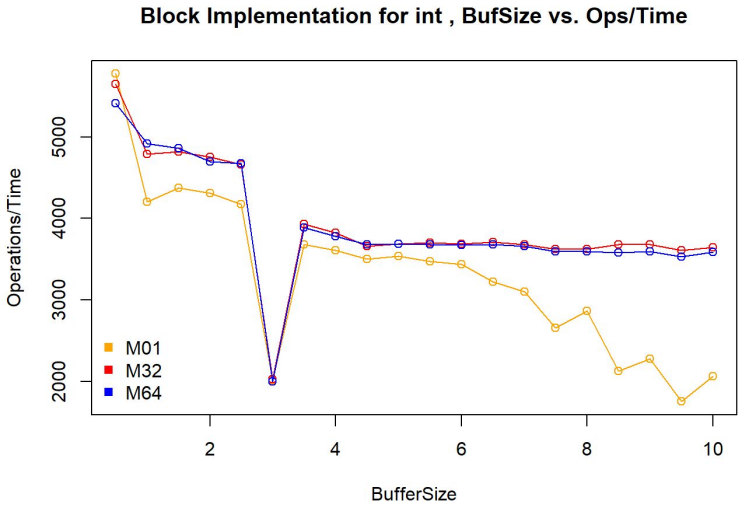
- {Mac, Integer, Sum}
    - A big drop in performance right at 4m+ problem size
- {Mac, Integer, Multiply} & {Mac, Long, Multiply}
    - Consistent anomalous behavior at 7m problem size -
      extremely high variance, even though it doesn't severely break the pattern
- {Raspberry Pi, Multiply} -
    - No L3 cache size, L2 cache size is 0.5 MB, so these graphs fit perfectly into what we should expect - very efficient at 0.5 MB, huge drop off after that, and then steady decline



Block Data Implementation

Note: All data measured on the Macbook Pro. orange:block size 1, red: block size 32, blue:block size of 64





Observations on blocking:

- Seeing consistent anomalous behavior for *int* at 3meg and *long* at 6meg buffer size
    - Suspect this is a page alignment issue from how matrices are stored, needs further exploration
- When matrix too big to fit in L3 cache, seeing significant performance improvement from blocking
- Minimal difference between block size of 32 and 64, must explore alternate sizes

System Specs

MacBook	ASUS Laptop	Raspberry Pi V3
3MB L3 Cache	6MB L3 Cache	512kb L2 Cache
MacBook Pro Retina, 13-inch, Late 2013 Model Identifier: MacBookPro11,1 Processor Name: Intel Core i5 Processor Speed: 2.4 GHz Number of Processors: 1 Total Number of Cores: 2 L2 Cache (per Core): 256 KB L3 Cache: 3 MB Memory: 8 GB 1600 MHz DDR3 Graphics: Intel Iris 1536 MBBoot ROM Version: MBP111.0138.B25 SMC Version (system): 2.16f68	ASUS GL552VW-DH74 Intel® Skylake™ i7-6700HQ (2.6GHz - 3.5GHz, 6MB Intel® Smart Cache) NVIDIA® GeForce™ GTX 960M (4.0GB) GDDR5 PCI-Express DX12 (Maxwell) w/ Optimus™ 2x8GB DDR4 2133MHz Dual Channel Memory	SoC: Broadcom BCM2837 CPU: 4× ARM Cortex-A53, 1.2GHz GPU: Broadcom VideoCore IV RAM: 1GB LPDDR2 (900 MHz) Networking: 10/100 Ethernet, 2.4GHz 802.11n wireless Bluetooth: Bluetooth 4.1 Classic, Bluetooth Low Energy Storage: microSD GPIO: 40-pin header, populated Ports: HDMI, 3.5mm analogue audio-video jack, 4× USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)