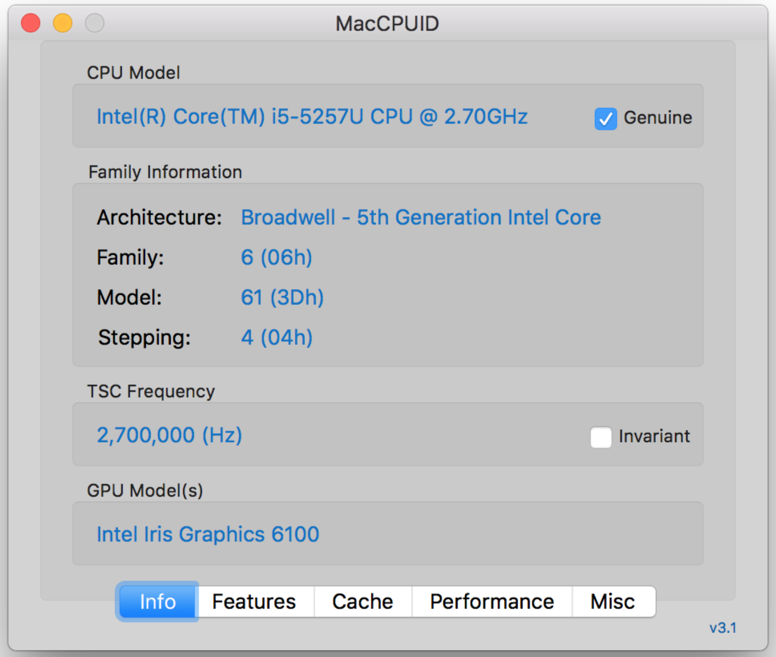
15-740 Homework 2

Jonathan Laurent, Matthew Kasper
jonathan.laurent@cs.cmu.edu, mkasper@andrew.cmu.edu
October 12, 2018

0: CPU Infos

We did this homework on a MacBook Pro (2015 edition). We used the MacCPUID tool from Intel to get the characteristics of our CPU. The results are shown Figures 1, 2 and 10.



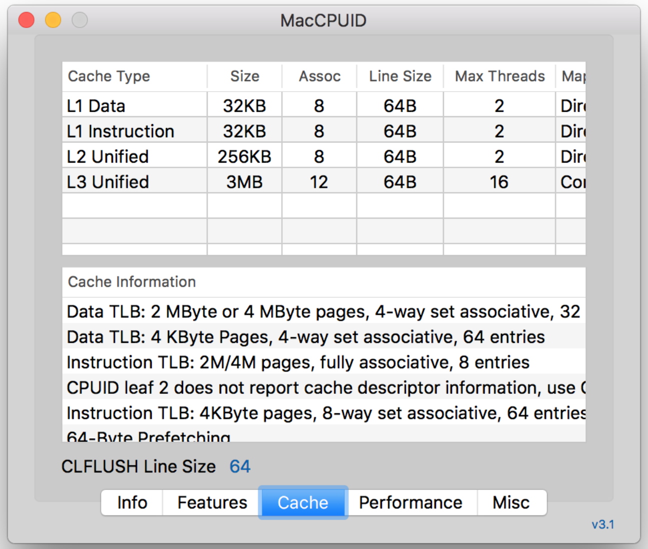
CPU Model
Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz ✓ Genuine

Family Information
Architecture: Broadwell - 5th Generation Intel Core
Family: 6 (06h)
Model: 61 (3Dh)
Stepping: 4 (04h)

TSC Frequency
2,700,000 (Hz) Invariant

GPU Model(s)
Intel Iris Graphics 6100

Figure 1: General CPU characteristics. See also Figures 2 and 10.



MacCPUID

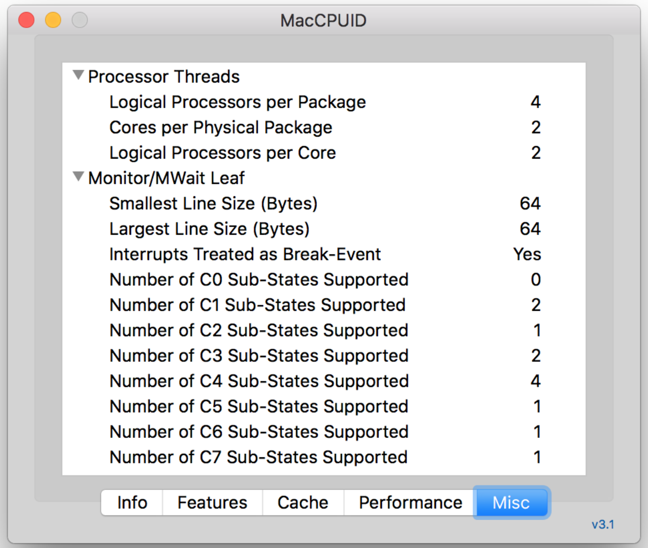
Cache Type	Size	Assoc	Line Size	Max Threads	Ma
L1 Data	32KB	8	64B	2	Dir
L1 Instruction	32KB	8	64B	2	Dir
L2 Unified	256KB	8	64B	2	Dir
L3 Unified	3MB	12	64B	16	Co

Cache Information

```
Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32
Data TLB: 4 KByte Pages, 4-way set associative, 64 entries
Instruction TLB: 2M/4M pages, fully associative, 8 entries
CPUID leaf 2 does not report cache descriptor information, use
Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
64-Byte Prefetching
```

CLFLUSH Line Size 64

Figure 2: Cache characteristics.



MacCPUID

```
▼ Processor Threads
    Logical Processors per Package          4
    Cores per Physical Package              2
    Logical Processors per Core             2
▼ Monitor/MWait Leaf
    Smallest Line Size (Bytes)             64
    Largest Line Size (Bytes)              64
    Interrupts Treated as Break-Event     Yes
    Number of C0 Sub-States Supported       0
    Number of C1 Sub-States Supported       2
    Number of C2 Sub-States Supported       1
    Number of C3 Sub-States Supported       2
    Number of C4 Sub-States Supported       4
    Number of C5 Sub-States Supported       1
    Number of C6 Sub-States Supported       1
    Number of C7 Sub-States Supported       1
```

Figure 3: Number of physical cores

1: Understanding the Memory Hierarchy

We show Figure 4 the memory mountain we generated for our machine. Here are our conclusions.

- Our machine has three different cache levels. Indeed, we can observe three plateaus corresponding to the following situations: the array fits in the L2, the array fits in the L3, the array fits in DRAM. Note that the L2 plateau is very small and is better seen on Figure 5.
- Looking at these different plateaus, we can make predictions about the sizes of the different caches. From Figure 5, we predict the following:
  - The size of the L2 is 2^{18}B = 256KB (end of the L2 plateau on Figure 5), which is correct.
  - The size of the L3 is 2^{21}B = 2MB (our processor spec says 3MB).
  - The size of the L1 is 2^{11}B = 2KB. This is a **wrong prediction**, as our L1 is $2^{15}B =$ 32KB. Surprisingly, the value `log2(size)=15` does not correspond to anything special on our mountain.
- Temporal locality seems to play almost no role as long as the array fits in the L3 cache. Indeed, the L2 and L3 plateaus are mostly unaffected by the stride parameter.
- In order to determine the line sizes of our different caches, we look at Figure 6, which shows the front section of the memory mountain, for a fixed value of `log2(size)=30`.
  - Naively, we would expect this curve to be a polyline with $k + 1$ linear segments, k being the number of distinct cache line sizes on our system. Moreover, we expect these segments to have descreasing slopes, the last one being flat (this corresponds to the situation where each reads refer to a new line at every cache level).
  - We do not explain the first plateau for stride values of [1,4].
  - Ignoring this first plateau, there is only one discontinuity on the curve, which (correctly) suggests that all caches have the same line size.
  - The flat part at the end of the curve must correspond to the part where each read accesses a different cache line. Therefore, cache lines must be $16 \times 8 = 128$B. It turns out that this prediction is **incorrect**: on our system, all caches have a line size of 64B. One possible explanation for this is that when loading a cache line after a miss, our processor also prefetches the next line in memory. Therefore, when going through a large array, memory is effectively cached in blocks of 128 bytes.
- To answer a question asked in the instructions, for a stride value of 1, there is little drop-off in performance as n exceeds the size of the largest cache on the chip. Indeed, time locality kicks in and $\frac{7}{8}$ of the reads do not generate cache misses when values are accessed contiguously (due to the prefetching mechanism conjectured above, the exact figure may rather be $\frac{15}{16}$).

Overall, we find the memory mountain Figure 4 quite surprising. In addition to the inconsistencies discussed above, there is only a 3× factor between the worst and best memory performance in our benchmark. Moreover, the L2 and L3 plateaus are almost indistinguishable. This questions the relevance of having three distinct cache levels. A possible answer here is that there is little gain of having a three-level cache hierarchy for executing sequential programs on a single core but that

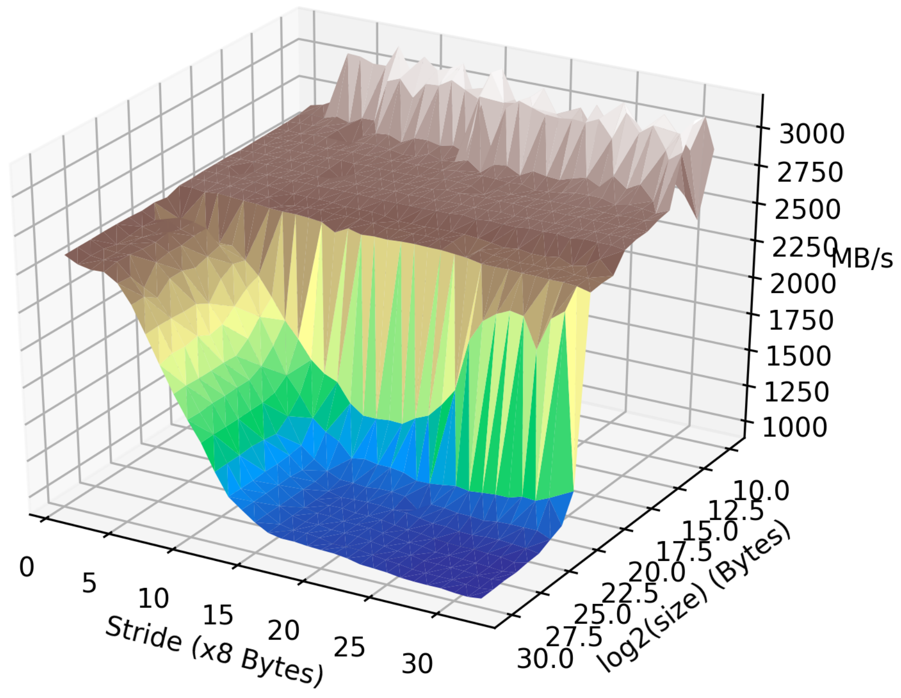


Figure 4: The memory Mountain

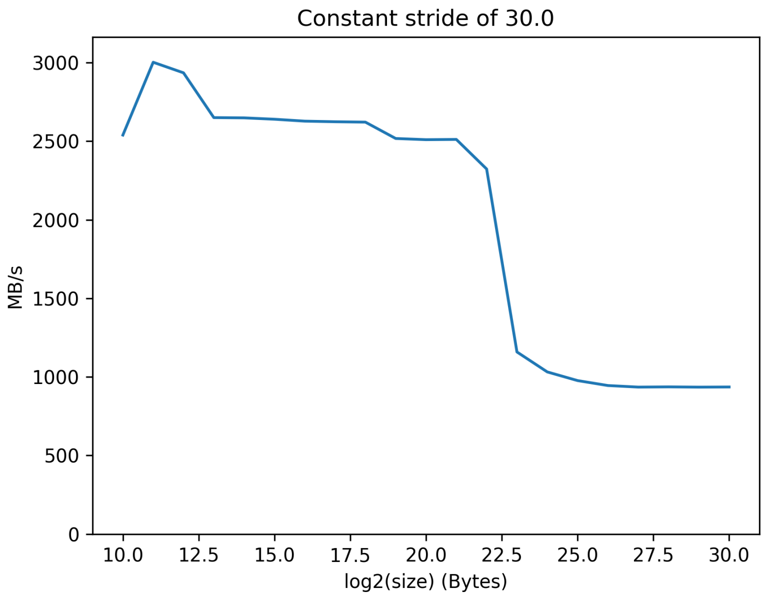


Figure 5: Section of the memory mountain for a fixed stride of 30 and varying values of `log2(size)`

this hierarchy makes sense on a multicore chip where L2 caches are private and the L3 cache is shared.

In order to get better results, we tried to run the benchmark while deactivating cache prefetching. Although it is easy to prevent the compiler to emit explicit prefetching instructions, instructions to disable hardware prefetching are only available at privelege level 0 (kernel). Other benchmarks with less predictible memory access patterns may be used instead but then it is unclear what meaning the *stride* parameter would have.

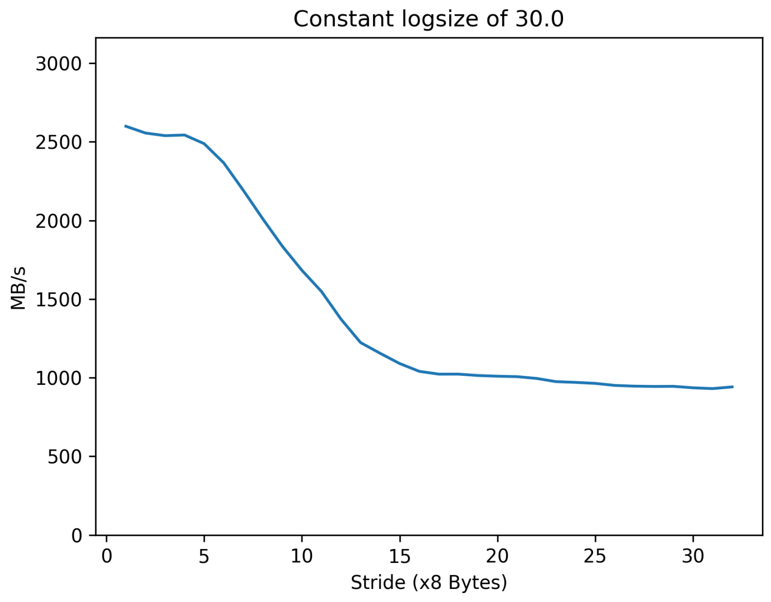
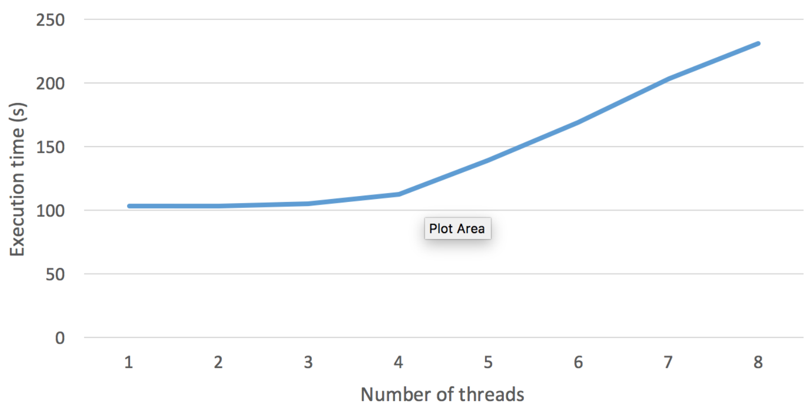


Figure 6: Section of the memory mountain for a fixed size of 2^{30}B and varying stride values

2: How many cores?

In order to compute an experimental estimate of the number of physical cores on our processor, we run n threads performing the same task independently and measure how much time it takes for all threads to terminate (for different values of n). The results are available Figure 7.



n	time (s)
1	103.50
2	103.82
3	105.38
4	113.13
5	139.29
6	169.64
7	203.46
8	231.21

Figure 7: Execution time of our micro-benchmark as a function of the number n of threads

From these results, it is clear that our processor features **four cores**. Indeed, our micro-benchmark takes about the same time to execute for $n \in \{1, 2, 3, 4\}$, when each thread can be executed on a different core (there is a slight overhead for $n = 4$, due to the fact that our machine was running other processes in addition to our benchmark). For $n > 4$, the execution time of the benchmark grows linearly in the number of threads, which is to be expected as threads start to compete with each other for cores.

3: Determining the Shared Cache Linesize

In `linesize.c`, we allocate an array of bytes (`uint8_t`) at an address `data` that is a multiple of the biggest cache size. Then, we spawn two threads that simultaneously and repeatedly increment the bytes at address `data` and `data + offset` respectively. We measure the number of operations per second performed by each thread for different values of `offset`. The result is shown Figure 8.

```
sep   Mops/s
----  ------
4     140.1
8     112.9
16    112.4
32    112.9
64    413.5
128   410.5
256   410.2
512   412.9
1024  410.8
```

Figure 8: Output of `linesize.c`

From these results, it is clear that our machine has a shared cache line size of 64B. Indeed, the performance of our micro-benchmark rises suddenly when the value of `offset` reaches 64, indicating that false sharing does not happen anymore, meaning that `data` and `data + offset` refer to different cache lines.

4: SMT Or Core?

To determine whether hyperthreading was occurring, what I did was bind each threat in my microbenchmark to the same CPU core. I then had a fixed number of threads each perform the same amount of work, and I timed how long it took for them all to complete. As I increased the number of threads, trends in the slope of the thread count vs. time curve hint at the presence of SMT. When the curve changes slope, going from something more flat to something a bit steeper, the number of threads this occurs at should be the number of logical cores associated with each physical core. This is because all threads are bound to the same physical core, so the only parallelism comes from hyperthreading. When the number actual of threads is smaller than the maximum number of SMT threads, they can all run and exploit at least some parallelism. However, when the number of threads created becomes larger than the maximum number of SMT threads, some of the threads will have to be descheduled and wait to use the CPU, causing a discontinuity in the slope of the performance curve. Our curve is shown in Figure 9.

Though subtle, it is clear that there is an increase in the slope of the curve at 2 threads, leading us to conclude that this is the number of SMT cores our system supports. It turned out to be hard to get clean data, however, and some of the discontinuities in the slope of this curve do not appear to have a direct correlation with the number of SMT cores.

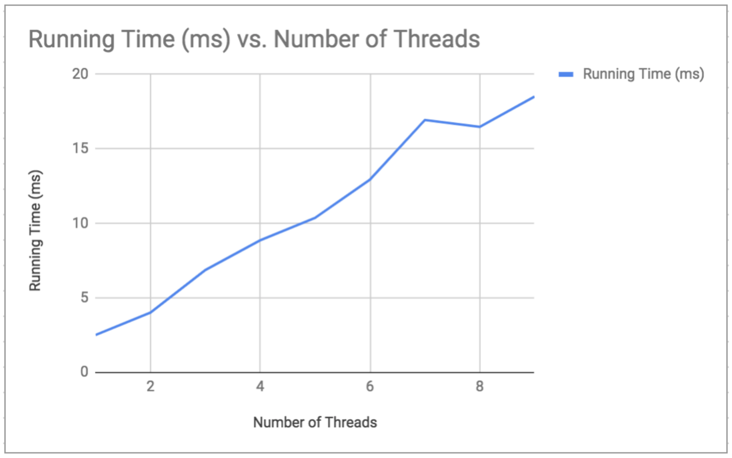


Figure 9: Running Time vs Number of Threads on Single Physical Core

5: A Better Lock

For this problem, I implemented an `atomic_increment` routine in x86-64 assembly, using the `XADDL` instruction. I then used this routine to increment a counter shared across many threads, and compared the results to another implementation which used a semaphore to guard the critical section where shared data was accessed. The results are shown below for various numbers of threads, ranging from 1 to 16.

```
Using semaphore with 1 threads: time = 0.022687s
Using semaphore with 2 threads: time = 0.078937s
Using semaphore with 3 threads: time = 0.194750s
Using semaphore with 4 threads: time = 0.285500s
Using semaphore with 5 threads: time = 0.275250s
Using semaphore with 6 threads: time = 0.298250s
Using semaphore with 7 threads: time = 0.345000s
Using semaphore with 8 threads: time = 0.323500s
Using semaphore with 9 threads: time = 0.319000s
Using semaphore with 10 threads: time = 0.346250s
Using semaphore with 11 threads: time = 0.339750s
Using semaphore with 12 threads: time = 0.323500s
Using semaphore with 13 threads: time = 0.286000s
Using semaphore with 14 threads: time = 0.318250s
Using semaphore with 15 threads: time = 0.310500s
Using semaphore with 16 threads: time = 0.291000s

Using atomic operations with 1 threads: time = 0.006844s
Using atomic operations with 2 threads: time = 0.019922s
Using atomic operations with 3 threads: time = 0.039062s
Using atomic operations with 4 threads: time = 0.051219s
Using atomic operations with 5 threads: time = 0.072125s
Using atomic operations with 6 threads: time = 0.091687s
Using atomic operations with 7 threads: time = 0.090750s
Using atomic operations with 8 threads: time = 0.103313s
Using atomic operations with 9 threads: time = 0.146000s
Using atomic operations with 10 threads: time = 0.145750s
Using atomic operations with 11 threads: time = 0.171000s
Using atomic operations with 12 threads: time = 0.179250s
Using atomic operations with 13 threads: time = 0.183625s
Using atomic operations with 14 threads: time = 0.193750s
Using atomic operations with 15 threads: time = 0.250500s
Using atomic operations with 16 threads: time = 0.288250s
```

It is clear that the `atomic_increment` performs better, especially for small numbers of threads. I imagine this is because a semaphore may rely on making system calls to explicitly deschedule threads that fail to enter the critical region. Since the critical section is so small, the overhead here becomes quite large by comparison, which is why the solution using `XADDL` performs much better for small numbers of threads.

However, as the number of threads becomes larger, the difference becomes less noticeable, even favoring the semaphore approach in some cases. This could be because the cache coherence implications for 16 threads issuing atomic instructions in parallel can become costly.

6: Optimized Matrix Multiplication

To find the optimal values to use for the number of threads and the block size, I first held the block size constant and slowly increased the number of threads I was using until my speedup plateaued (or even decreased a tiny bit). Then, I held the number of threads constant, while I varied the block size to find the value that yields optimal performance. I ultimately settled on a block size of 64 bytes and 32 threads.

Using these settings, I varied the size of the matrices I was multiplying, and measured the throughput at each size. The results are shown in Figure 9. Ultimately, increasing the data size increased the throughput as well. This makes sense intuitively, as the more data there is, the less noticeable the cost of creating 32 threads will be. What was surprising was that the curve did not plateau as I would have expected it to. Perhaps if we continued to increase the data size, this would occur.

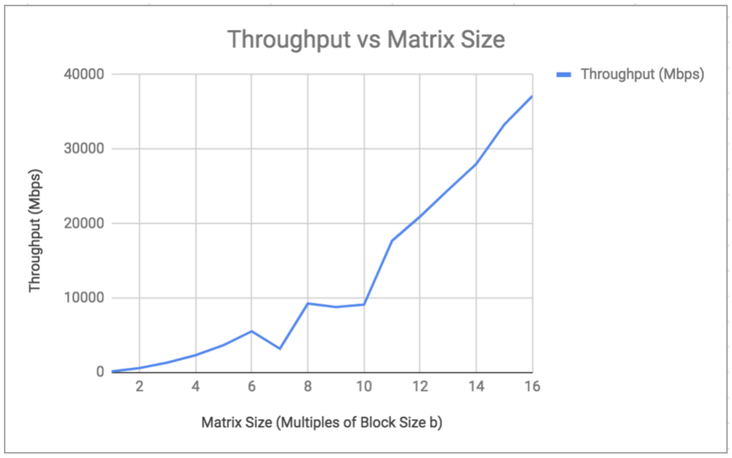


Figure 10: Matrix Multiply Throughput vs Matrix Size