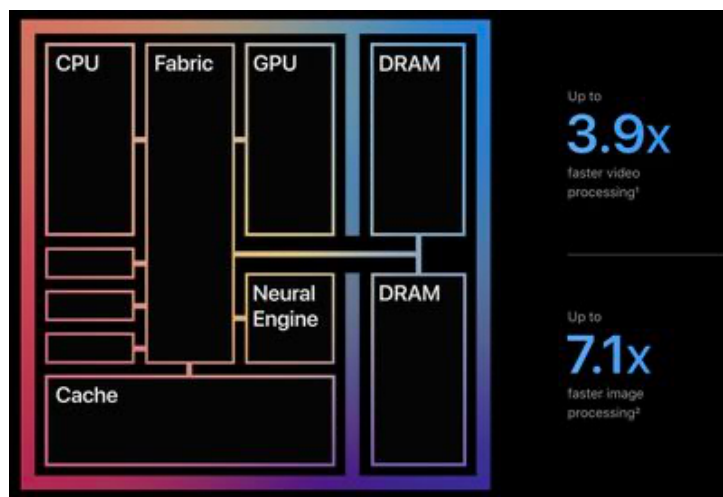


P9a Project: The Apple M2 Chip

This report delves into the Apple M2 chip, its capabilities and its limitations. We will briefly explain the design of the system, after which the theoretical limitations of both the CPU and GPU will be discussed. Finally, both processing units will be tested in practice to compare with their theoretical performances.






The M2 belongs to the series of Apple's own processors, which they unrolled back in 2020. It is a so-called system-on-chip design, that is an integrated circuit containing several different processing units and auxiliary components on the same silicon-based structure. This allows for optimization in both size, power consumption and efficiency, and thus these SoC designs are often found in portable devices.



As noted in the picture, the chip encompasses both a central processing unit and a graphics processing unit, as well as cache and (dynamic) random access memory. Furthermore, it contains a neural engine, which is a specific processing unit highly optimized for neural network related operations. The unified memory design mitigates many speed bottlenecks related to accessing and storing data in different parts of memory. Memory-related bottlenecks become increasingly important as one optimizes a program, and often serve as the limiting factor for further speedups after a certain point. The M2 memory is shared by all cores of the processors, meaning we skip unnecessary data copies. The drawback of this unified design is the lack of upgradeability. The M2 chip on my device contains 16GB of RAM.

A) Theoretical Limitations:

The CPU contains eight cores. These can be split into four performance cores, running at 3.49 GHz clock frequency, and four energy-efficient cores, running at 2.42 GHz. The performance cores contain 192 KB instruction cache, 128 KB L1-level cache and 16 MB L2-level cache memory, while the efficiency cores contain 128 KB instruction cache, 64 KB L1-level cache and 4 MB of shared L2-level cache. Additionally, we have 8 MB of L3-level cache that is shared with the GPU. The unified memory bandwidth is 100 GB/s, although I could not exactly find out if this number holds for data transfers between all the memory types on the chip. The following table aggregates the CPU specifications:

Series	Apple Apple M2		
Series: Apple M2			
Apple M2 Max 	2.42 - 3.7 GHz	12 / 12 	48 MB L3
Apple M2 Pro 	2.42 - 3.5 GHz	12 / 12 	24 MB L3
Apple M2 Pro 10-Core 	2.42 - 3.7 GHz	10 / 10 	24 MB L3
Apple M2 «	2.42 - 3.48 GHz	8 / 8 	8 MB L3
Clock Rate	2424 - 3480 MHz		
Level 1 Cache	2 MB		
Level 2 Cache	20 MB		
Level 3 Cache	8 MB		
Number of Cores / Threads	8 / 8		

Let us now reason about the theoretical maximal performance of this CPU. We start by focusing on a single high-performance core @3.49 GHz. A common design paradigm in the Apple Silicon chip-series that enables performance at low power-consumption is trying to increase the number of operations / instructions per cycle (per core), without increasing the clock rate too much. Thus, only looking at the clock rate will not serve as a good indication of the performance of the CPU. I had a hard time finding specific instruction statistics for the M2 chip, but managed to find research by Dougall Johnson on the M1 instruction latencies. As discussed in the sources, the instruction throughput and latency variation is not large, so using these numbers, one would expect a somewhat accurate result.

Apple Microarchitecture Research by [Dougall Johnson](#)

M1/A14 P-core (Firestorm): [Overview](#) | [Base Instructions](#) | [SIMD and FP Instructions](#)
M1/A14 E-core (Icestorm): [Overview](#) | [Base Instructions](#) | [SIMD and FP Instructions](#)

Firestorm SIMD and FP Instructions

LAT: latency in cycles (cycles per instruction when latency bound)

TP: reciprocal throughput (cycles per instruction when throughput bound)

Uops: uop count (towards the pipeline width limit)

Int/Mem/FP: issue counts for each group of units (towards the one per unit per cycle limit)

Units: best guess of units used

	LAT	TP	Uops	Int	Mem	FP	Units
► ABS	3	0.25	1	-	-	1	u11-14
► ADD	2	0.25	1	-	-	1	u11-14

As we see, the ADD operation's reciprocal throughput seems to be 0.25, or a throughput of 4 instructions per clock cycle. Assuming this is the statistics for a single floating-point addition between two numbers, then we could estimate a simple theoretical max limit for floating-point additions per second by the formula

$$T_{\text{max}} = 1/TP * C$$

where TP denotes the reciprocal throughput and C the clock frequency. The latency is not that relevant when considering longer-term performance: it is not about the delay to execute a single instruction, rather the dominant factor becomes how fast you can serially execute a 'train of similar operations'. Find below a table with the relevant information and theoretical calculations for floating-point addition, subtraction, multiplication and division for both a performance-core and an energy-efficient core. All values are calculated using the data in Johnson's findings.

performant / efficient		(cycles / instruction)	(M2 clock rates)	(FLOPS)		
Core (P / E)	Instruction	TP = Reciprocal Throughput	C = Clock Rate (GHz)	T_max = 1/TP * C	ALL four P-cores in use	ALL four E-cores in use
P	FADD	0.25	3.49	1.40E+10	5.58E+10	
P	FSUB	0.25	3.49	1.40E+10	5.58E+10	
P	FMUL	0.25	3.49	1.40E+10	5.58E+10	
P	FDIV	1	3.49	3.49E+09	1.40E+10	
E	FADD	0.5	2.42	4.84E+09		1.94E+10
E	FSUB	0.5	2.42	4.84E+09		1.94E+10
E	FMUL	0.5	2.42	4.84E+09		1.94E+10
E	FDIV	1	2.42	2.42E+09		9.68E+09
Instruction		ALL cores in use				
FADD	7.52E+10					
FSUB	7.52E+10					
FMUL	7.52E+10					
FDIV	2.36E+10					

The GPU also contains eight cores, each with 16 execution units. These units themselves contain eight arithmetic logic units, yielding a total of 1024 ALUs. Several sources (gpu-monkey.com, notebookcheck.net) cite a clock rate of 1.4 GHz, an expected decrease since GPUs are often optimized for parallel processing. These sources also all point to a total performance of 3.6 TFLOPS for the 10-core version, but we can also try to calculate it manually using the microarchitecture research by Philip Turner. This relates to the 10-core version, but one can assume fairly similar numbers for my 8-core version. The throughput is given in pairs for different parameter setups, but the author claims that the 2nd number is more accurate, so we will use that. I could not find throughput numbers for subtraction, but let us assume for simplicity that it equals that of addition.

▼ Floating-point performance

Float Instruction	Throughput	Raw Latency	Adjusted Latency
FADD16	1, 1	2.97-3.33	2.16
FMUL16	1, 1	2.98-3.34	2.17
FFMA16	1, 1	2.97-3.35	2.18
FADD32	2, 1	3.50-3.90	2.20
FMUL32	2, 1	3.50-3.91	2.21
FFMA32	2, 1	3.50-3.91	2.21

Let us construct a similar table as for the GPU:

Instruction	TP	C	T_max	ALL cores in use
FADD		1	1.4	1.40E+09
FSUB	-		1.4	-
FMUL		1	1.4	1.40E+09
FDIV		6	1.4	2.33E+08

For some reason, these calculations are wildly off from the expected range of FLOPs, in the order of 10^{12} . This is perhaps due to a grave misunderstanding on my part of somehow not including parallel multiplicity factors. Per notebookcheck.net, the 8-core GPU theoretical performance is around 3 GFLOPs.

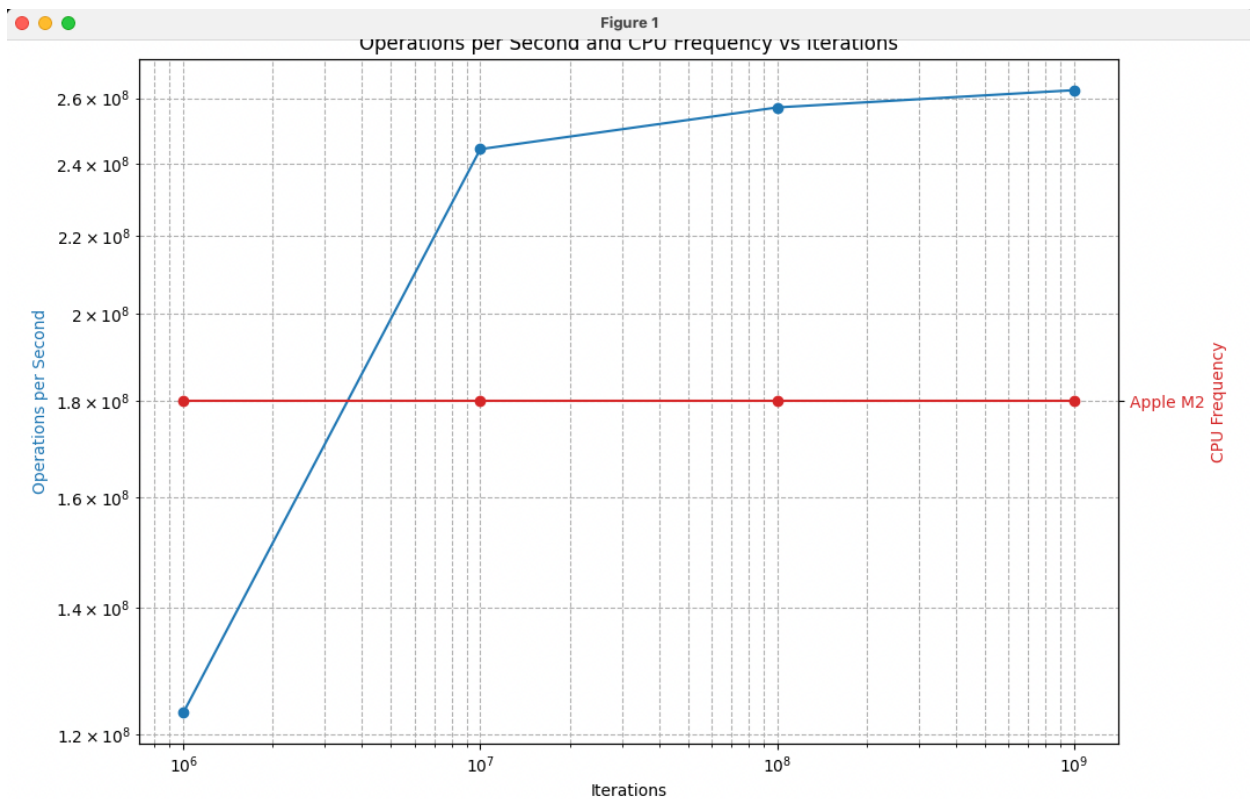
B) Practical Limitations:

Let us now put the hardware to the test and see what performance we achieve in practice. We will use C++ programs that perform as many floating-point operations as possible in a given time-frame. Let us first consider the addition, subtraction and multiplication operations, since they are equally fast. Since CUDA is not supported on my hardware, and I am not familiar with any other GPU-programming library, I will only run CPU-side tests.

The main C++ program is essentially an inner loop that performs four scalar operations. We define a result variable, and float operands a, b and c. We then perform:

1. `Result += a * b;`
2. `Result -= c;`
3. `Result *= a;`
4. `Result /= b;`

, so two multiplications, one addition, one subtraction and one division. We store all intermediary results back in Result. We measure the wall clock time for the inner loop using the functions of the Chrono library. The outer loop defines that we start by doing this loop a million times, incrementing by a factor of 10 up to 1 billion iterations. The nested-loop structure ensures that we perform four passes with different input sizes. We then store the calculated data of operations per second along with the iteration count to a csv file, which we later graph with a Python script using Pandas and Matplotlib. Moreover, we measure the CPU clock rate using MacOS systctl library, although from what I understand, it only gives the base frequency as opposed to the actual operating frequency. This is also corroborated by the constant line in the graph. The program was compiled with “g++ -o FLOPperformanceTest FLOPperformanceTest.cc”, so no additional optimization flags. Below are the results of the the test FLOPperformanceTest.cc, plotted in the Python script:

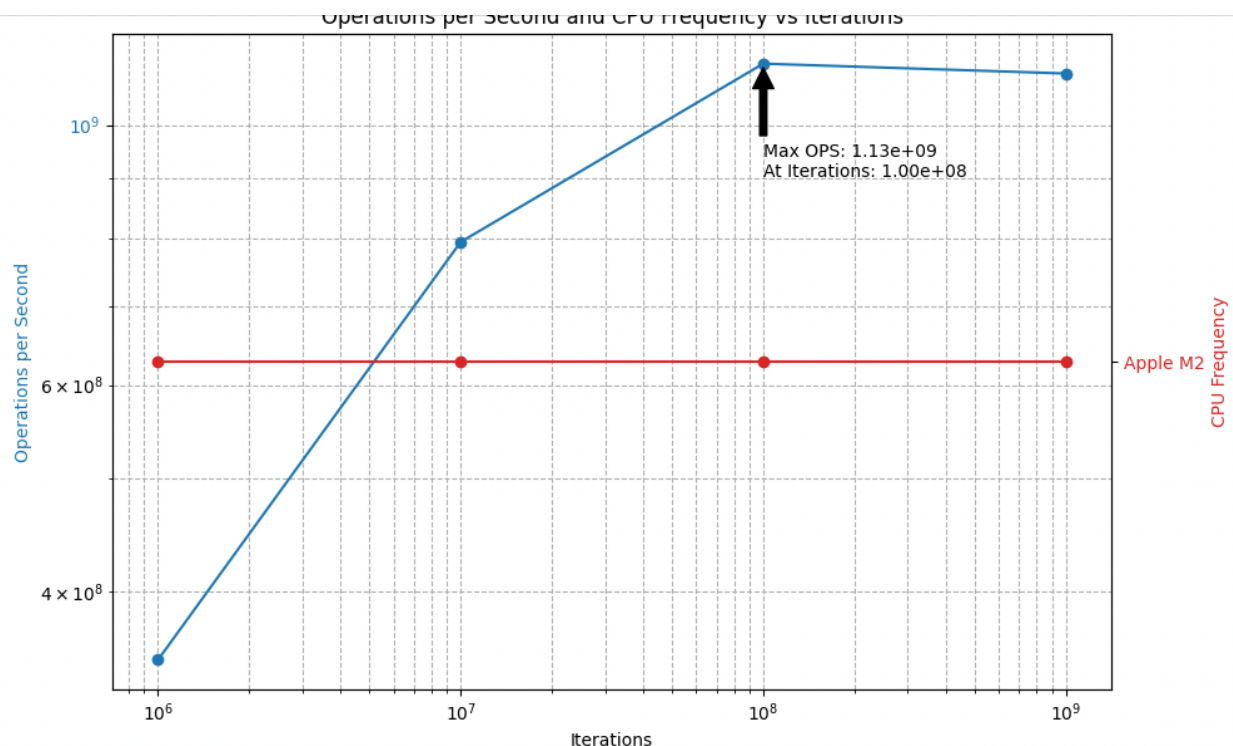


The experimental results for this single-threaded program indicate that the performance tapers off at roughly 2.6×10^8 FLOPs, which differs from the theoretical maximum of the performance core by a factor of 10^2 . Compiling with further optimization flags resulted at best in $\approx 3 \times 10^8$ FLOPs. This is a significant discrepancy, so let us try to reason about why this is.

Firstly, the theoretical calculations might be wrong. The numbers are not found in official Apple-provided documents, but rather individuals' own research on the topic. I might also have misunderstood the material and somehow calculated wrong results. Secondly, the theoretical results assume maximal performance in all aspects, e.g. operating at the highest achievable clock frequency. Since I couldn't get the operating clock frequency, it is hard to tell how close we were to this maximum, so there is a possibility of the CPU not operating at full speed due to e.g. thermal throttling. In terms of memory bottlenecks, the program is operating on a few values inside the innermost loop, without accessing data from other places, so I do not think that this is an issue here. I am also only assuming that we operated on a performance core; if the program was run on an efficiency core, we would only be off by a factor of 10. The program uses no SIMD, but from what I gathered, the

numbers used for the theoretical calculations are also for scalar operations. Other small factor that adds to the discrepancy is the running of other programs simultaneously and measurement errors, although these are unlikely the main culprits for this difference.

I also created a similar multi-threaded version of the test, where I run an openmp pragma to utilize as many cores as possible on my machine. The results below indicate that we only then reach the ballpark of the theoretical maximum of a single performance core. We also achieved a speedup from the sequential test by roughly a factor of 10, indicating that all eight cores were probably heavily used. However, this result is still off by a factor of 10 from the theoretical maximum when using all eight cores available. Running -O2 or -O3 optimize flags results in a wild result in order of 10^{17} in magnitude, which indicates that the optimizations heavily changed the code beyond its inherent structure, and thus these results are not meaningful. Below are the results of the test MTFLOPperformanceTest.cc:



I also included the corresponding assembly files for the tests, although I am not on a level where I can analyze it to see if things work as expected.

C) Comparison with results from CP

I will now compare the practical results obtained in b) with the results obtained from the course task Correlated Pairs. The task consists of computing the correlations between the row vectors of an input matrix, and outputting these correlation coefficients in an output matrix. The built-in checks for cycles and instructions are not available on my OS, so the measurements will be rather crude. I will try to manually estimate the amount of floating-point operations the program executes, and then run the standard benchmark to measure wallclock time. I run the command `./grading benchmark benchmarks/name.txt` to only focus on the speed and ignore validity checks. I do not know what optimization flags are used in this compiling process, which can lead to unexpected results. Since I only performed two tests in b), a sequential and the multi-threaded version, I will try to compare with corresponding solutions CP1 and CP2b.

The CP1 solution is a straight-forward sequential computation. We have n_y rows of vectors, and each vector is dotted with another. One dot product is n_x multiplications and n_x additions to the sum variable. Since we only compute the upper triangular result matrix, which is square with dimensions $n_y \times n_y$, we compute $n_y(n_y+1)/2$ elements. Therefore, we perform roughly $n_y(n_y+1) / 2 \times (2 \times n_x)$ operations for the actual computation, excluding incrementing loops and operations related to element access.

We also perform normalization: for mean normalization, we perform $n_y \times n_x$ additions and n_y divisions. In the norm normalization, we perform $n_y \times n_x$ additions, $n_y \times n_x$ multiplications, $n_y \times n_x$ divisions and $n_y \times n_x$ square roots. I am not sure how many floating point operations a square root comprises, but let us assume one for simplicity. In total, the normalization requires $5 \times n_y \times n_x + n_y$ operations.

Benchmark 2 uses a 4000×1000 matrix as input. This results in 16 024 004 000 FLOPs. Running the benchmark results in 6.168s of wallclock time, and the crude calculation gets us $\approx 2,6 \times 10^9$ FLOPs. This is larger than the results in b) by a factor of 10. Again, an unexpected result, since the test in b) is much simpler, and does not pose the same problems with regards to memory bottlenecks. This result is, however, closer to the theoretical maximum of a single performance core. Compiler settings might have affected the result.

CP2b solves the same task in a multi-threaded approach. The structure of the code is exactly the same, but both normalization loops and the matrix multiplication loop are parallelized using the openmp parallel for pragma. We use a dynamic schedule with one unit of work per thread. This means we have the same amount of operations in the program, and only need to divide by the faster wallclock time. The benchmark resulted in 1.079s of wallclock time, which then yields $14,9 \cdot 10^9$ FLOPs. Furthermore, the test shows we used roughly 7 simultaneous hardware threads on average. Again, comparing to the theoretical calculations, we see that we run at only roughly 10% of the maximal speed one could obtain. The possible reasons for the discrepancy is already outlined in the section for the result for CP1.

```
● danielmichaeli@Daniels-Laptop cp1 % ./grading benchmark benchmarks/2.txt
Running benchmark
Compiling...
Compiled
test           time  result
benchmarks/2.txt 6.177s pass

Standard error:
Could not add performance counter 'cycles': Performance counter 'cycles' is currently not supported on this OS.
Could not add performance counter 'instructions': Performance counter 'instructions' is currently not supported on this OS.
Could not add performance counter 'branches': Performance counter 'branches' is currently not supported on this OS.
Could not add performance counter 'branch_misses': Performance counter 'branch_misses' is currently not supported on this OS.

Your code used 6.177 sec of wallclock time, and 6.168 sec of CPU time
≈ you used 1.0 simultaneous hardware threads on average
```

```
● danielmichaeli@Daniels-Laptop cp2b % ./grading benchmark benchmarks/2.txt
Running benchmark
Compiling...
Compiled
test           time  result
benchmarks/2.txt 1.079s pass

Standard error:
Could not add performance counter 'cycles': Performance counter 'cycles' is currently not supported on this OS.

Your code used 1.079 sec of wallclock time, and 7.828 sec of CPU time
≈ you used 7.3 simultaneous hardware threads on average
```

Note: these screenshots were taken the calculations above, with new test runs. Wallclock time is roughly the same, and will not affect the calculations.

Sources:

General M2 / Macbook Air 2022 design:

<https://support.apple.com/en-us/111867>

<https://www.youtube.com/watch?v=WZeaQwwwc8s>

https://en.wikipedia.org/wiki/System_on_a_chip

<https://www.apple.com/se/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/>

<https://www.macrumors.com/guide/m2/>

<https://www.ansys.com/blog/what-is-system-on-a-chip>

<https://github.com/hollance/neural-engine>

M2 CPU specs:

<https://www.notebookcheck.net/Apple-M2-Processor-Benchmarks-and-Specs.632312.0.html>

<https://versus.com/en/apple-m2>

https://www.cpu-monkey.com/en/cpu-apple_m2

Theoretical calculations:

<https://stackoverflow.com/questions/74637333/cycle-count-neon-for-m2>

<https://scicomp.stackexchange.com/questions/36306/how-to-properly-calculate-cpu-and-gpu-flops-performance>

<https://news.ycombinator.com/item?id=27134434>

Dougall Johnson: data for theoretical CPU calculations:

<https://dougallj.github.io/applecpu/firestorm.html>

M2 GPU specs:

https://www.gpu-monkey.com/en/gpu-apple_m2_8_core_gpu

<https://www.notebookcheck.net/Apple-M2-8-Core-GPU-GPU-Benchmarks-and-Specs.635064.0.html>

Philip Turner: data for theoretical GPU calculations:
<https://github.com/philipturner/metal-benchmarks>

