

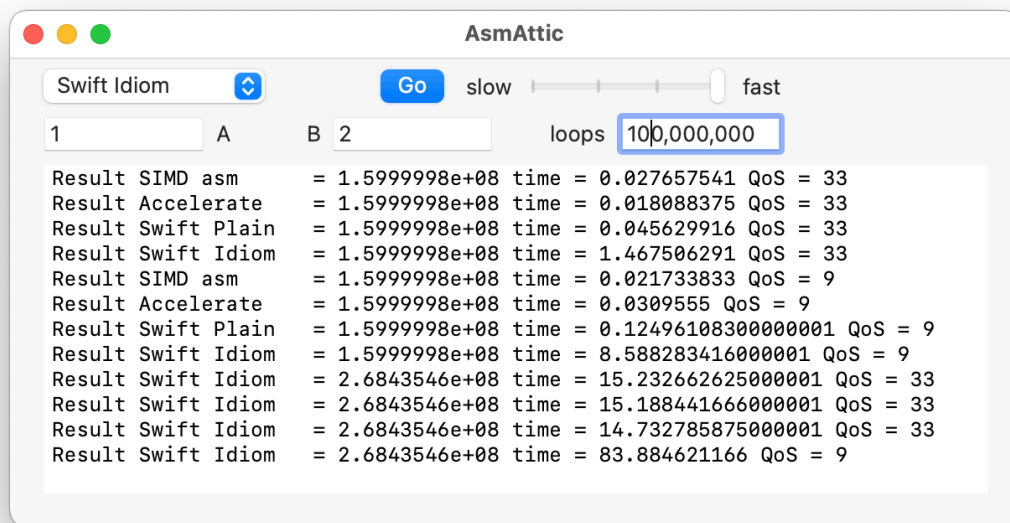
AsmAttic 4.0 for M1, M1 Pro and M1 Max

AsmAttic 4.0 is a utility for exploring performance of the different cores in Apple's M1 series of SoCs, as supplied in the following models of Mac:

- M1 MacBook Air 2020
- M1 MacBook Pro 13-inch 2020
- M1 Mac mini 2020
- M1 iMac 24-inch 2021
- M1 Pro MacBook Pro 2021
- M1 Max MacBook Pro 2021.

The app doesn't run *at all* on any Intel Mac, nor on any version of macOS before Big Sur 11.3.

Use



AsmAttic has a single window. When you close this, the app quits.

This window contains two rows of controls at the top, and a scrolling text view which fills the rest of the window and contains all results.

To run a benchmark test, select **which test** to run in the popup menu at the top left, set the slider at the right to determine its scheduling **QoS**, then enter values for **A**, **B** and the number of **loops**. When you're ready to run the test, click the **Go** button or press the Return key. After an interval, the result will appear in the scrolling view.

There are four different methods to perform what is essentially the same benchmark test:

- **SIMD asm** runs hand-coded Assembly language
- **Swift Plain** runs a simple version coded in Swift
- **Swift Idiom** runs a more sophisticated and less efficient version in Swift
- **Accelerate** runs a function in the macOS Accelerate library.

The **QoS** slider determines the 'Quality of Service', with **fast** being the fastest, and **slow** the slowest. Normally on the M1, slow runs the test entirely on the Efficiency cores, and the other three settings use the Performance cores.

A and **B** are starting values used to run the benchmark tests, and are floating point numbers. It's simplest to start with relatively low values, around 1.0 or 2.0, but you can use positive and negative values much smaller or larger as you wish. The A values are used to populate the vector **a**, and the B values for vector **b**. Each test then calculates the dot product

$$\mathbf{a} \cdot \mathbf{b} = a_1b_1 + a_2b_2 + a_3b_3 + a_4b_4$$

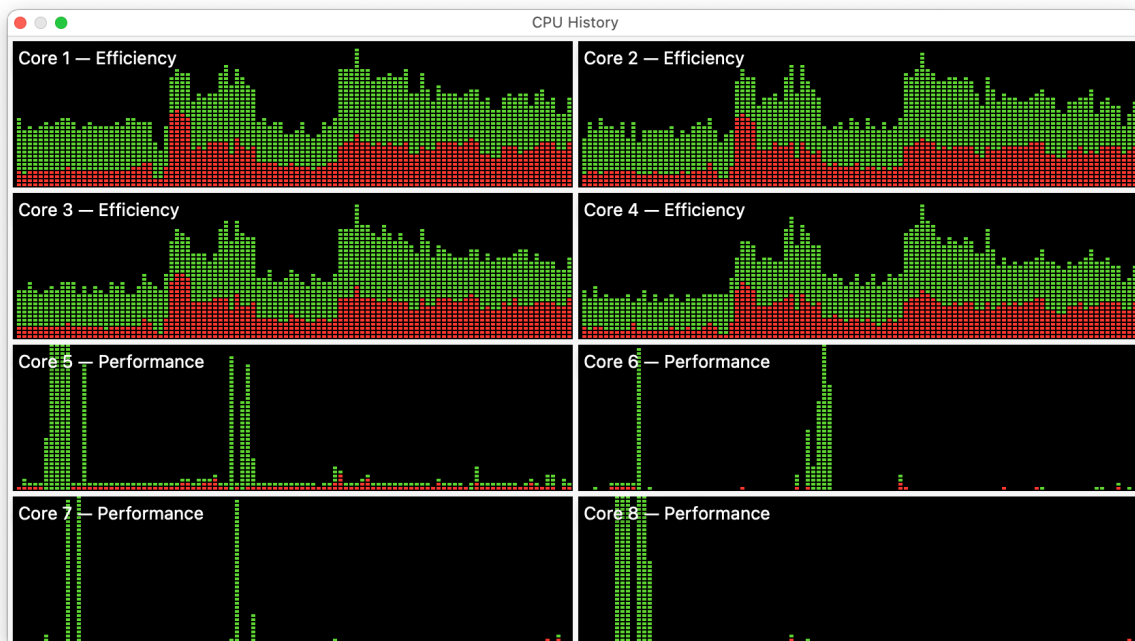
A third vector **c** is initialised with values of twice those in **a**. At the end of the dot product calculation, vector **a** is incremented by **c** ready for the next loop. The result returned is the dot product of the last loop completed.

The **loops** value determines how many times the benchmark calculation is performed. These typically need to be more than 1,000,000, and I normally use a positive integer between 10,000,000 and 100,000,000 to obtain sufficiently large times for the results.

Each test result reports:

- the test performed
- the result, which should be identical regardless of which test you perform, for any given A, B and loops values
- the time taken in seconds
- the QoS setting for that test.

Every test is run in a background thread using the control settings at the time the test was started using the Go button or Return key. You can easily set several tests running in rapid succession, using different QoS levels to have them run on different cores. Watch the effects in the CPU History view in Activity Monitor, to see which cores become loaded.



In this case, tests were run when the Efficiency cores were already loaded with a large Time Machine backup. Two tests were run at high QoS, as seen in the short peaks in load of the Performance cores, and two tests at low QoS, which are seen in the increased load of the Efficiency cores.

Tests

Each of the four tests used performs a dot-product operation between two floating-point vectors of length 4, containing 32-bit floating point values. Where possible, these use the Neon vector processing feature in the cores.

Assembly routine

```
_dotprod:
    STR    LR, [SP, #-16]!
    LDP    Q2, Q3, [X0]
    FADD    V4.4S, V2.4S, V2.4S
    MOV    X4, X1
    ADD    X4, X4, #1
dp_while_loop:
    SUBS    X4, X4, #1
    B.EQ    dp_while_done
    FMUL    V1.4S, V2.4S, V3.4S
    FADDP    V0.4S, V1.4S, V1.4S
    FADDP    V0.4S, V0.4S, V0.4S
    FADD    V2.4S, V2.4S, V4.4S
    B       dp_while_loop
dp_while_done:
    LDR    LR, [SP], #16
    RET
```

with the two vectors passed as (the address of) an Array of Floats, and the number of loops as an integer.

Swift Plain

```
func runSwiftTest2(theA: Float, theB: Float, theReps: Int) -> Float {
    var tempA: Float = theA
    var vA = [theA, theA, theA, theA]
    let vB = [theB, theB, theB, theB]
    let intC: Float = theA * 2.0
    let vC = [intC, intC, intC, intC]
    for _ in 1...theReps {
        tempA = 0.0
        for i in 0...3 {
            tempA += vA[i] * vB[i]
        }
        for i in 0...3 {
            vA[i] = vA[i] + vC[i]
        }
    }
    return tempA
}
```

Swift Idiom

```
func runSwiftTest(theA: Float, theB: Float, theReps: Int) -> Float {
    var tempA: Float = theA
    var vA = [theA, theA, theA, theA]
    let vB = [theB, theB, theB, theB]
    let vC = vA.map { $0 * 2.0 }
    for _ in 1...theReps {
        tempA = zip(vA, vB).map(*).reduce(0, +)
        for (index, value) in vA.enumerated() {
            vA[index] = value + vC[index]
        }
    }
    return tempA
}
```

AsmAttic 4.0 for M1, M1 Pro and M1 Max

Accelerate

```
func runAccTest(theA: Float, theB: Float, theReps: Int) -> Float {
    var tempA: Float = theA
    var vA = simd_float4(theA, theA, theA, theA)
    let vB = simd_float4(theB, theB, theB, theB)
    let vC = vA + vA
    for _ in 1...theReps {
        tempA = simd_dot(vA, vB)
        vA = vA + vC
    }
    return tempA
}
```

Background processing etc.

This is performed using an OperationQueue with the set QoS.

maxConcurrentOperationCount is currently fixed at 4, with results being returned on the main OperationQueue. Times are obtained using mach_absolute_time().

Have fun!