A NUMA architecture is one that has multiple nodes, each with their own local memory/cache. This speeds up access time by not having each core of a CPU locked on memory accesses that other cores might be accessing, memory latency, and memory consistency. NUMA nodes still may have to access external memory, which is usually handled by its own hardware, which ensures memory accesses stay consistent, with a speed penalty. According to some benchmarks, the speedup from local memory accesses is up to 33%, with bandwidth following the same trend.

https://www.kernel.org/doc/html/v6.9/mm/numa.html
https://learn.microsoft.com/en-us/windows/win32/procthread/numa-support
https://en.wikipedia.org/wiki/Non-uniform_memory_access
https://blog.e-zest.com/non-uniform-memory-architecture-numa/
https://techdocs.broadcom.com/us/en/storage-and-ethernet-connectivity/ethernet-nic-controllers/bcm957xxx/adapters/Tuning/tcp-performance-tuning/nic-tuning_22/numa-local-vs-non-local.html
https://stackoverflow.com/a/7262135

Compilers can affect parallel execution in regards to optimizing sequential code, specifically in regards to out of order optimizations it may apply, such as changing the order in which variables are initialized and memory allocated, which may be transparent to a sequential program but could lead out of of bounds memory accesses in parallel scenarios. There can also be positive effects on performance, specifically in cases where the compiler can determine data dependencies and can block of code to auto parallelize code, or in cases where the compiler can detect a use case for AVX instructions (SIMD).

A scenario in which parallelization affects the output of a program in unexpected ways is when you have data dependencies that are ran in parallel without consideration for each read and write, for example:

```
let a = 1
mutate_on_thread_1(a)
mutate_on_thread_2(a)
print(a)
```

There is no way to know what a will be, since a is getting mutated at the same time.

https://www.intel.com/content/www/us/en/developer/articles/technical/automatic-parallelization-with-intel-compilers.html
https://softwareengineering.stackexchange.com/a/421686
https://old.reddit.com/r/compsci/comments/dh2nld/whats_to_stop_or_limit_compilers_from/f3htpn4/

The difference between shared memory parallelism and distributed parallel computation is that threads operate on a shared memory space, taking consideration of acceses, using primitives like mutexs and semaphores, where distributed parallel computation takes place on computers with their own memory and no parallel execution, with commands dispatched to them. These models can be combined, with dispatched commands to shared memory machines.

https://lms.unb.ca/d2l/le/content/257020/viewContent/2935160/View?ou=257020

```
# This function the most simple manual way of doing matrix multiplication without builtins
# It does not take into account multithreading or any acceleration
# It does however use the least amount of memory, only allocating new memory for the sum and the results matrix
function for_loop(a, b)
    n = size(a, 1)
    m = size(b, 1)
    if (size(a, 2) != m)
        println("Matrixes not of multipliable sizes")
    end
    p = size(b, 2)

    c = Matrix{Int}(undef, n, p)

    for i in 1:n
        for j in 1:p
            sum = 0
            for k in 1:m
                a1 = a[i, k]
                b1 = b[k, j]
                sum = sum + (a1 * b1)
            end
            c[i, j] = sum
        end
    end
    c
end

# This function is very easy as a programmer to use, as it is builtin and uses OpenBLAS, which has more advanced algorithms for doing matrix multiplication
# However, it fails under a few scenarios, where you are memory bound doing a lot of multiplications on low dimension matrixes
# as it uses much more memory even for a simple 5x5 matrix than the for loop, presumably due to allocation costs and setup/error checking
function built_in(a, b)
    a * b
end

# In terms of speed, the for loop loses in pretty much every scenario, but wins initially in terms of memory usage, but as the
# matrixes begin to scale dimensions into hundreds or thousands, the builtin becomes much closer in terms of memory usage
function main()
    matrix_regex = r"(\d+)x(\d+)"

    println("Format is {num}x{num}. Eg: 3x3 or 100x100")

    print("Dimension of first matrix: ")
    input = readline()
```

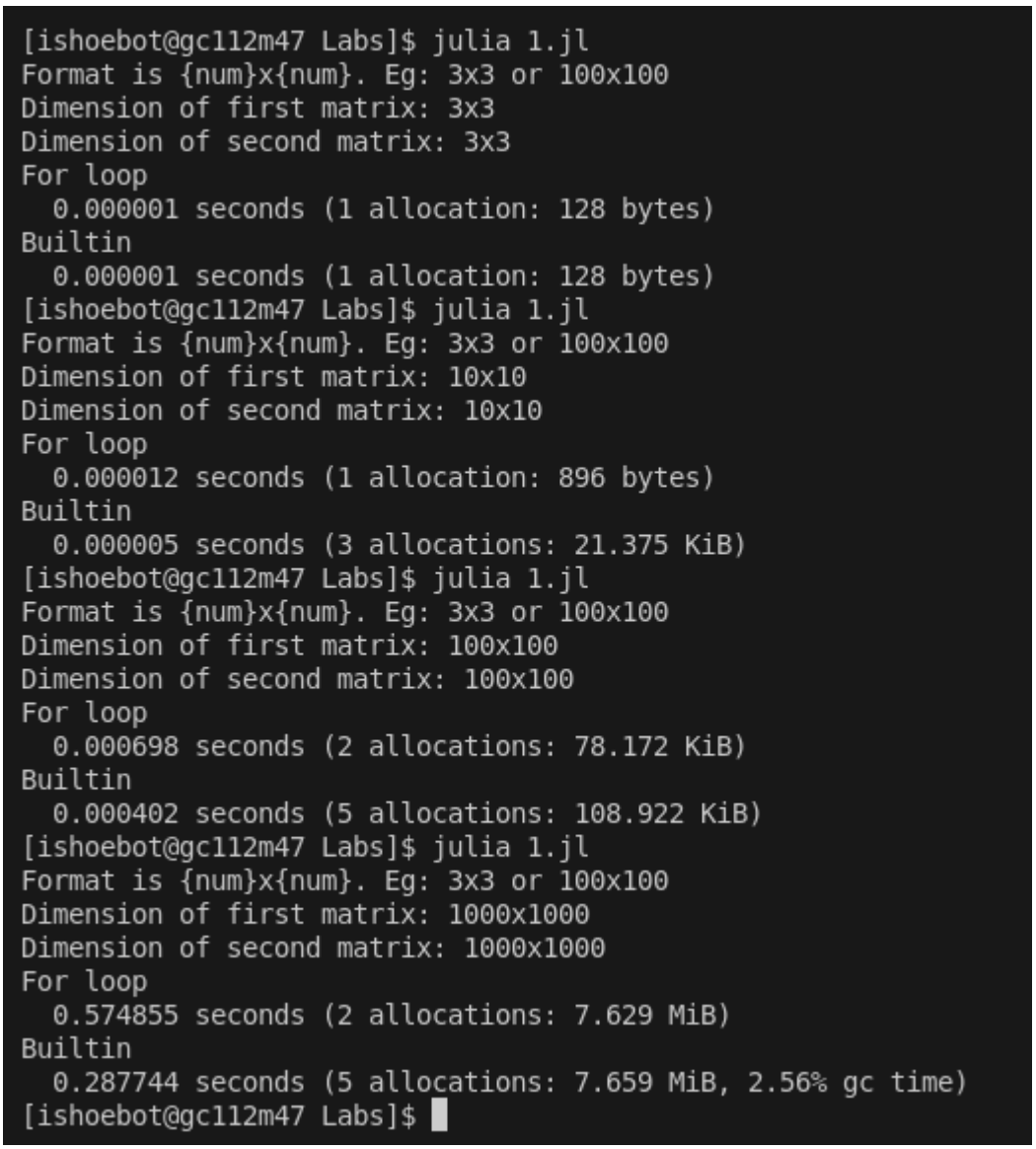
```
    matrix_match = match(matrix_regex, input)
    matrix_1 = rand(Int, (parse(Int, matrix_match[1]), parse(Int, matrix_match[2])))

    print("Dimension of second matrix: ")
    input = readline()
    matrix_match = match(matrix_regex, input)
    matrix_2 = rand(Int, (parse(Int, matrix_match[1]), parse(Int, matrix_match[2])))

    println("For loop")
    @time for_loop(matrix_1, matrix_2)

    println("Builtin")
    @time built_in(matrix_1, matrix_2)
end

main()
```



```
[ishoebot@gc112m47 Labs]$ julia 1.jl
Format is {num}x{num}. Eg: 3x3 or 100x100
Dimension of first matrix: 3x3
Dimension of second matrix: 3x3
For loop
  0.000001 seconds (1 allocation: 128 bytes)
Builtin
  0.000001 seconds (1 allocation: 128 bytes)
[ishoebot@gc112m47 Labs]$ julia 1.jl
Format is {num}x{num}. Eg: 3x3 or 100x100
Dimension of first matrix: 10x10
Dimension of second matrix: 10x10
For loop
  0.000012 seconds (1 allocation: 896 bytes)
Builtin
  0.000005 seconds (3 allocations: 21.375 KiB)
[ishoebot@gc112m47 Labs]$ julia 1.jl
Format is {num}x{num}. Eg: 3x3 or 100x100
Dimension of first matrix: 100x100
Dimension of second matrix: 100x100
For loop
  0.000698 seconds (2 allocations: 78.172 KiB)
Builtin
  0.000402 seconds (5 allocations: 108.922 KiB)
[ishoebot@gc112m47 Labs]$ julia 1.jl
Format is {num}x{num}. Eg: 3x3 or 100x100
Dimension of first matrix: 1000x1000
Dimension of second matrix: 1000x1000
For loop
  0.574855 seconds (2 allocations: 7.629 MiB)
Builtin
  0.287744 seconds (5 allocations: 7.659 MiB, 2.56% gc time)
[ishoebot@gc112m47 Labs]$
```