Computational Optimization
ISE 407

Lecture 2

Dr. Ted Ralphs

Reading for this Lecture

- "All You Ever Wanted to Know About Memory", Ulrich Drepper
- "What Scientists Must Know About Hardware to Write Fast Code," J. Nissen ⇐ This lecture borrows copiously from this article
- "Introduction to High Performance Computing", V. Eijkhout, Chapter 1.
- "Introduction to High Performance Computing for Scientists and Engineers," G. Hager and G. Wellein, Chapter 3.

Basic Architecture

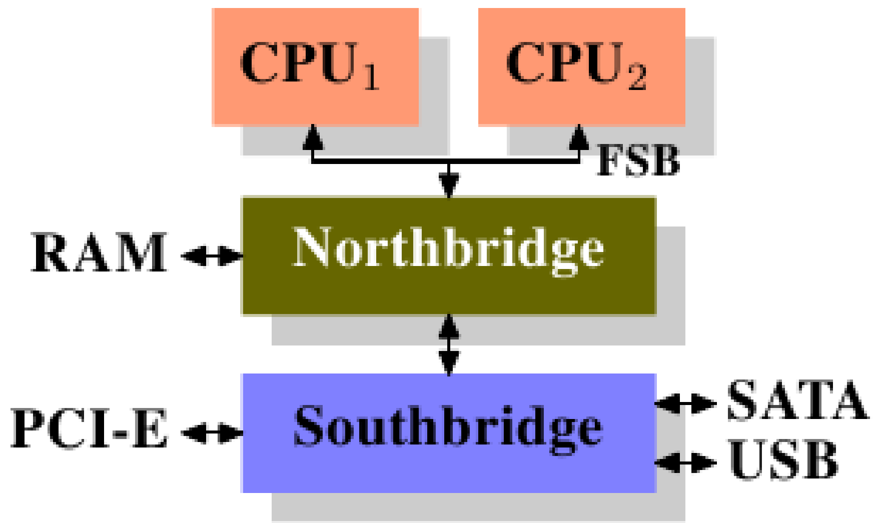


Figure 1: Basic architecture of a modern computer
Source: https://lwn.net/Articles/250967

Basic elements include

- CPU (Central processing unit)
- RAM (Random access memory)
- Storage
- Peripherals

The Memory Bottleneck

- There is an obvious bottleneck between CPU and memory.
- The bottleneck can be partially overcome with additional memory controllers.
- This increases complexity and expense.

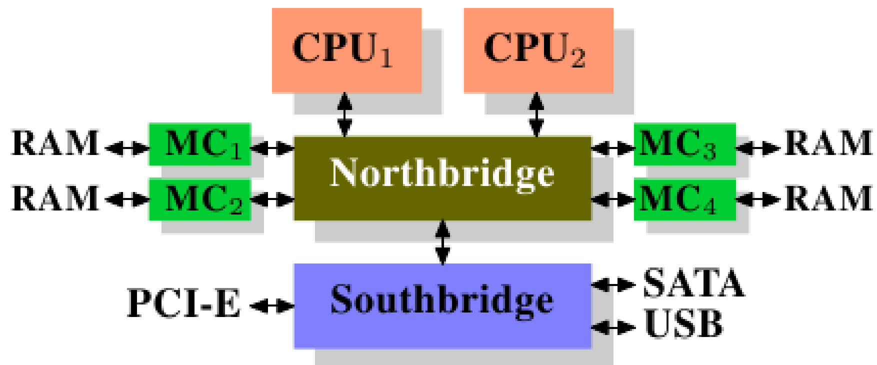


Figure 2: Adding memory controllers

Source: https://lwn.net/Articles/250967

Another Option

- A second option is to attach each CPU to local memory.
- This creates a small parallel architecture with an associated interconnection topology.
- All memory appears local, but access times are not uniform (called a NUMA architecture).

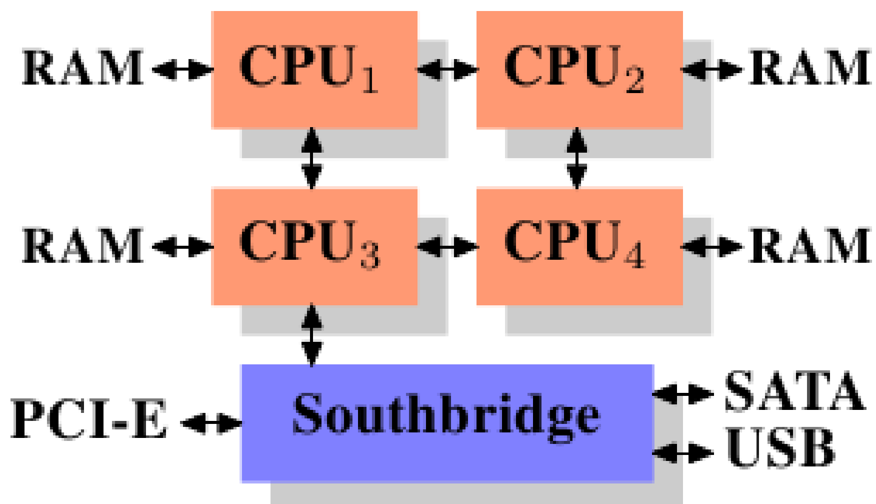


Figure 3: NUMA Architecture

Source: https://lwn.net/Articles/250967

Putting it Together

- Today's architectures consist of multiple processors, each with multiple cores.
- The resulting memory hierarchy is very complex and we only consider the simple case of a CPU with a single core for now.

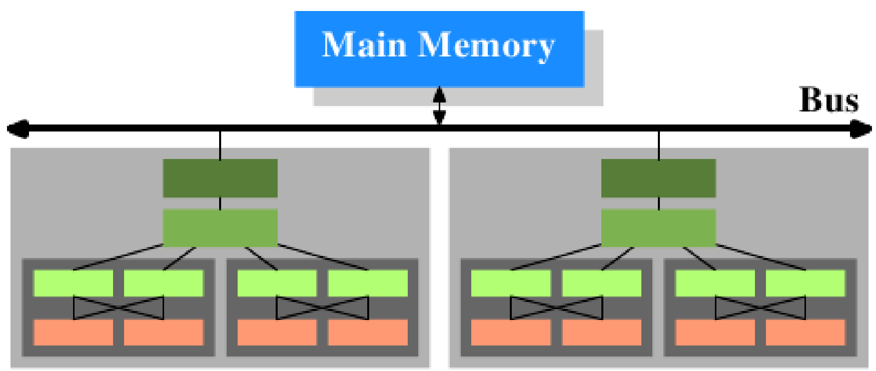
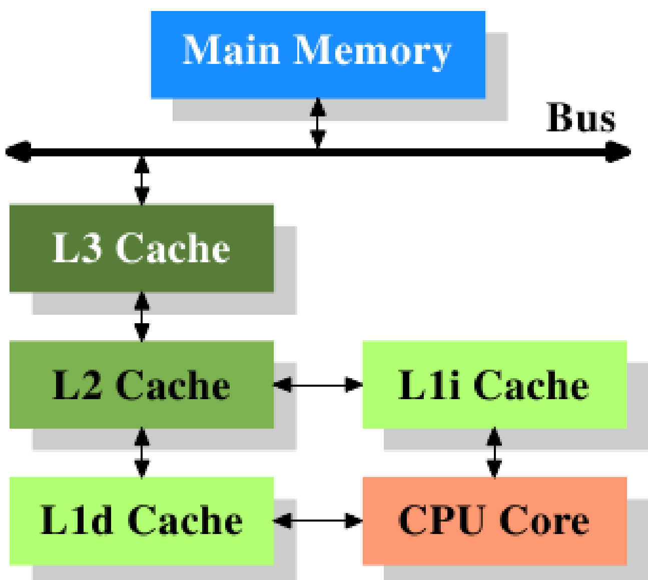


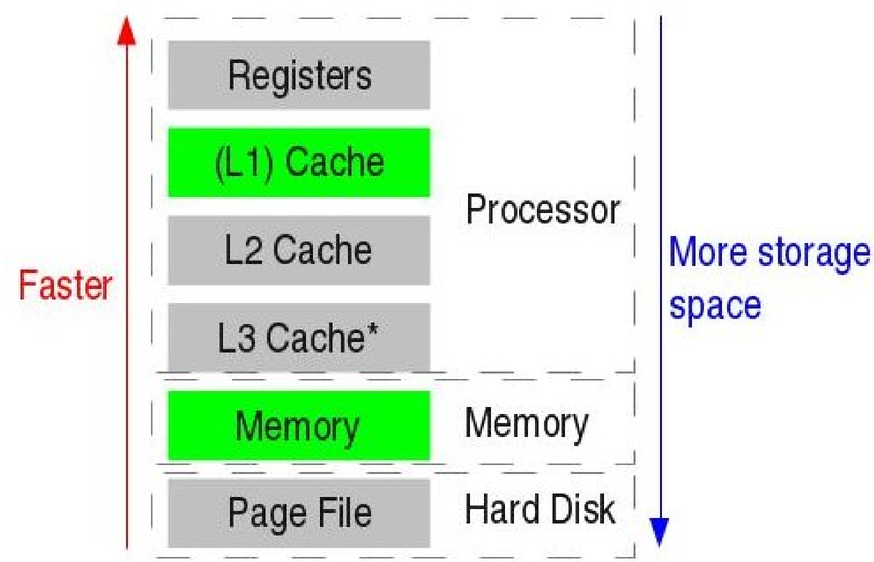
Figure 4: High-level view of entire architecture

Source: https://lwn.net/Articles/250967

Storage Hierarchy

- Even with the improvements discussed so far, there is a large gap between processor speeds and memory speeds.
- It is possible to produce faster memory, but it's expensive and takes much more physical space.
- As a compromise, we add small fast memory, called *cache*, for storing the most important data.
- There are stypically separate caches for instructions and data.





Source: https://lwn.net/Articles/250967

How Cache Works: Library Analogy

- Main memory is the shelf in the library filled with many books.
- The register is the book you have open: immediate access, but only one book.
- Level 1 cache are the books sitting on your desk: faster access, small capacity.
- Level 2 cache are the books on your book shelves.
- ...

Access Times

- Here are some representative access times
  - Register: 1 cycle
  - L1d: 3 cycles (64 kB)
  - L2: 14 cycles (512 kB)
  - L3: Usually shared, 6 MB
  - RAM: 240 cycles
- It is easy to see why it's important to understand the hierarchy.

Access Times Exemplified

```julia
function test_file(path)
    open(path) do file
        # Go to 1000'th byte of file and read it
        seek(file, 1000)
        read(file, UInt8)
    end
end
```

```
julia> @time test_file("Lecture2.tex")
  0.011654 seconds (16 allocations: 1.141 KiB)

julia> @time test_file("Lecture2.tex")
  0.000714 seconds (16 allocations: 1.141 KiB)
```

- This is the time access a single random byte in a file on my laptop.
- The drop in time when running the function again is because the file has now been cached.

Access Times Exemplified

```julia
function random_access(data::Vector{UInt}, N::Integer)
    n = rand(UInt)
    mask = length(data) - 1
    @inbounds for i in 1:N
        n = (n >>> 7) ⊻ data[n & mask + 1]
    end
    return n
end
```

```
julia> @time random_access(data, 1000000)
  0.159546 seconds
```

- This is the time to access 1000000 random bytes from an array.
- On my laptop, accessing random data in memory is roughly 70000x faster than accessing random bytes from a file.

Access Times Exemplified

```julia
function linear_access(data::Vector{UInt}, N::Integer)
    n = rand(UInt)
    mask = length(data) - 1
    @inbounds for i in 1:N
        n = (n >>> 7) ⊻ data[i & mask + 1]
    end
    return n
end
```

```
julia> @time linear_access(data, 1000000)
  0.004439 seconds
```

On my laptop, accessing data linearly in memory is roughly 35x faster than accessing data randomly.

How Does Cache Work?

- The big question is what do we put in the cache?
- Obviously, we want data that we'll be likely to need soon.
- This is very difficult to predict!
- How does cache works work with main memory?
  - When the CPU needs data, it first checks the cache.
  - If it finds what it needs, great! A *cache hit*.
  - Otherwise (a *cache miss*), it retrieves what it need from main memory and ejects something to make room.
  - Data is always fetched in blocks of a certain size (a *cache line*), even when only part of the block is needed.
- How do we predict what data will be used?
  - **Temporal locality**: Data used once will tend to be used again soon $\Rightarrow$ keep the most recently accessed data items closer to the CPU
  - **Spacial locality**: Data near data that has been recently used is likely to be used soon $\Rightarrow$ move contiguous closer to the CPU.

Example 1

Cache (4 lines, 1 byte per line);
Access time 1 cycle

index	valid	tag	data
00			
01			
10			
11			

RAM
Access time: 100 cycles

address	data
000000	data(0)
000001	data(1)
000010	data(2)
000011	data(3)
000100	data(4)
000101	data(5)
000110	data(6)
000111	data(7)
001000	data(8)
001001	data(9)
001010	data(10)
001011	data(11)

Core needs to access numbers in RAM in the following order

data	0	1	2	3	4	3	4	11
hit?								
miss?								
total cycles								

cache miss ratio:

Example 2

Cache (4 lines, 2 bytes per line)
Access time: 1 cycle

index	valid	tag	D0	D1
00				
01				
10				
11				

RAM
Access time: 100 cycles

address	data
000**00**0	data(0)
000**00**1	data(1)
000**01**0	data(2)
000**01**1	data(3)
000**10**0	data(4)
000**10**1	data(5)
000**11**0	data(6)
000**11**1	data(7)
001**00**0	data(8)
001**00**1	data(9)
001**01**0	data(10)
001**01**1	data(11)

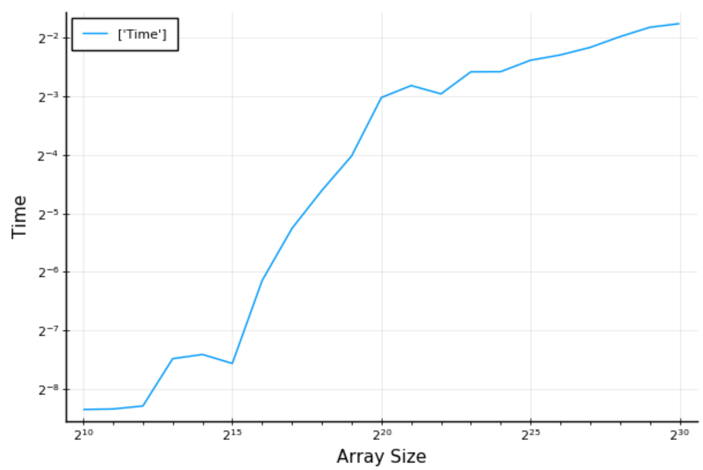
Core needs to access numbers in RAM in following order

data	0	1	2	3	4	3	4	11
hit?								
miss?								
cycles took								

cache miss ratio:

Some Further Experiments

- The sizes of the various caches can be queried (`getconf -a | grep CACHE` or with the `CpuId` package), but we can also derive them experimentally.
- On my laptop, the cache sizes are:
  - Level 1: 2^{12} 64-bit integers
  - Level 2: 2^{15} 64-bit integers
  - Level 3: 2^{20} 64-bit integers
- The following data were generated by random accesses into arrays of different sizes with the `random_access` function from earlier.



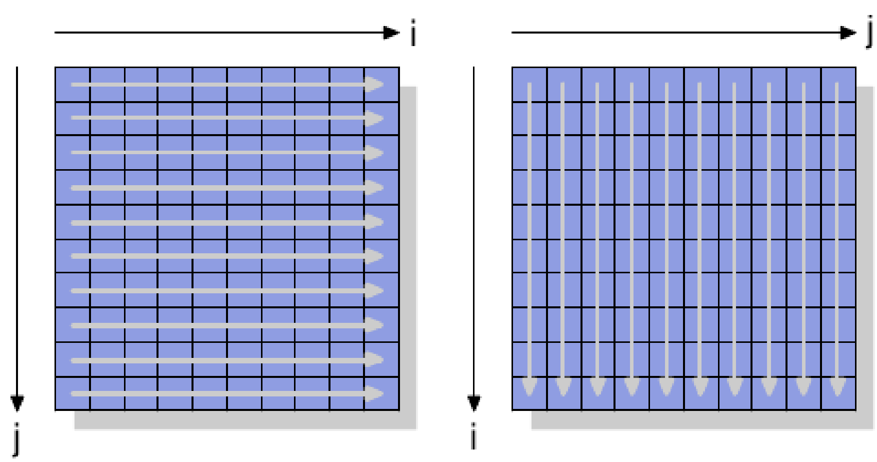
- The plateaus correspond exactly to the sizes of the caches.

Memory Layout

- For more complex data structures, it's important to keep in mind the layout in memory.
- In general languages vary in their default memory layout for multi-dimensional vectors according to convention.
  - Julia is a column-ordered language.
  - C/C++ is row-ordered.
  - Numpy in Python is is row-ordered (for general lists, the question doesn't make sense).
- This means that to loop over the elements of a multi-dimensional array in Julia, the outer-most loop should increment the last index.
- The inner-most loop should increment the first index.
- In C/C++, the opposite in true for statically allocated memory.
- Dynamically allocated memory is laid out manually and so can be laid out either way.

Impact of Memory Layout

- Consider the time to initialize a matrix.
- In Julia, matrices are stored column-wise.
- To move through the matrix element-by-element as the elements are laid out in memory, we iterate through the indices in order.
- We consider initializing column-wise and row-wise.



Source: https://lwn.net/Articles/250967

Matrix Initialization in Julia

```julia
function init_col_ordered(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for i ∈ inds
        out[:, i] = x
    end
    return out
end

function init_row_ordered(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for i ∈ inds
        out[i, :] = x
    end
    return out
end
```

```
julia> x = zeros(10000);

julia> @time init_col_ordered(x);
  1.808912 seconds (2 allocations: 762.940 MiB, 9.40% gc time)

julia> @time init_row_ordered(x);
  8.323898 seconds (14.71 k allocations: 763.676 MiB, 0.14% gc time)
```

Column ordered initialization is roughly four times faster.

Matrix Multiplication

- Now consider multiplying two matrices.
- A straightforward implementation in Julia would be

```julia
function matmult_naive!(C, A, B)
    # No checking for proper types or dimension match
    fill!(C, 0)
    for i ∈ 1:size(A, 1), j ∈ 1:size(B, 2), k ∈ 1:size(A, 2)
        C[i, j] += A[i, k] * B[k, j]
    end
    return(C)
end
```

```
julia> A = rand(1:10, 2^11, 2^11);
julia> B = rand(1:10, 2^11, 2^11);
julia> C = similar(A);
julia> @btime matmult_naive!($C, $A, $B);
  102.603 s (0 allocations: 0 bytes)
```

- Slow... because it is most natural to access one matrix row-wise and the other matrix column-wise, but this is bad.

The Improved Code

- What if we transpose one of the matrices first?

```
function matmult_trans!(C, A, B)
    fill!(C, 0)
    T = similar(A)
    @inbounds for i ∈ axes(A, 1), j ∈ axes(A, 2)
        T[i, j] = A[j, i]
    end
    @inbounds @fastmath for i ∈ axes(A, 1), j ∈ axes(T, 1)
        Cij = zero(eltype(C))
        for k ∈ axes(A, 2)
            Cij += T[k, i]*B[k, j]
        end
        C[i,j] = Cij
    end
end
```

```
julia> @btime matmult_trans!($C, $A, $B)
  5.544 s (2 allocations: 32.00 MiB)
```

- Although we're spending time to allocate memory and transpose the matrix first, it's much faster!

A Little More on Caching

- We may able to avoid transposing if we exploit how cache works.
- As we know, data is cached in lines that have a fixed length.
- Therefore, if we copy one element from an array into the cache, we will also get the next few elements for free.
- To get maximum performance, we should use all the data in the cache that we can before it gets evicted.
- In the matrix example, this means that we should do several inner products at the same time.

Cache-Aware Matrix Multiplication

```
function matmult_cache!(C, A, B)
    #Assume square matrices here to keep it simple
    fill!(C, 0)
    S = Int(cachelinesize()/sizeof(eltype(A)))
    N = size(A, 1) # Assume that N is a multiple of S
    @inbounds @fastmath for r ∈ 1:S:N, c ∈ 1:S:N, k ∈ 1:S:N
        for c2 ∈ c:c+S-1, k2 ∈ k:k+S-1,
            Bkc = B[k2, c2]
            for r2 ∈ r:r+S-1
                C[r2, c2] += A[r2, k2]*Bkc
            end
        end
    end
end
```

```
julia> @btime matmult_cache!($A, $B, $C);
  13.586 s (0 allocations: 0 bytes)
```

- We first compute the number of elements that a cache line can hold.
- Surprisingly slower than `matmult_trans!`, probably due to loop overhead.
- Perhaps we can combine the two ideas....

Cache-Aware Matrix Multiplication

```
function matmult_cache!(C, A, B)
    #Assume square matrices here
    fill!(C, 0)
    T = similar(A)
    @inbounds for i ∈ axes(A, 1), j ∈ axes(A, 2)
        T[i, j] = A[j, i]
    end
    S = Int(cachelinesize()/sizeof(eltype(A)))
    N = size(A, 1)
    @inbounds @fastmath for r ∈ 1:S:N, c ∈ 1:S:N, k ∈ 1:S:N
        for c2 ∈ c:c+S-1, r2 ∈ r:r+S-1,
            Crc = 0
            for k2 ∈ k:k+S-1
                Crc += A[k2, r2]*B[k2, c2]
            end
            C[r2, c2] = Crc
        end
    end
end
```

```
julia> @btime matmult_cache!($A, $B, $C);
  6.089 s (2 allocations: 32.00 MiB)
```

- An improvement, but no better than the naive transpose method.
- Are other memory tricks we can exploit? Yes!

Vectorization

- To allow for computations on data that doesn't fit in 64-bit registers, CPUs now have instructions that operate on special "wide registers".
- Typically, a wide register holds 4 64-bit numbers and are only utilized in very specific circumstances.
- The most common is a loop with fixed length and no branches where order doesn't matter.

```
julia> a = @SVector Int32[1,2,3,4,5,6,7,8]
julia> code_native(+, (typeof(a), typeof(a)), debuginfo=:none)
        .text
        movq    %rdi, %rax
        vmovdqu (%rdx), %ymm0
        vpaddd  (%rsi), %ymm0, %ymm0
        vmovdqu %ymm0, (%rdi)
        vzeroupper
        retq
        nopw    %cs:(%rax,%rax)
        nopl    (%rax)
```

- Note the vector instructions.
- The native code for the non-static vector is almost 500 lines!!

Vectorization Example

```julia
function sum_nosimd(x::Vector)
    n = zero(eltype(x))
    for i in eachindex(x)
        n += x[i]
    end
    return n
end
function sum_simd(x::Vector)
    n = zero(eltype(x))
    # By removing the bounds check, we allow automatic SIMD
    @inbounds for i in eachindex(x)
        n += x[i]
    end
    return n
end
```

```
julia> data = rand(UInt64, 4096) #Vector should fit in cache
julia> @btime sum_nosimd(data)
  2.233 µs (1 allocation: 16 bytes)
julia> @btime sum_simd(data)
  220.968 ns (1 allocation: 16 bytes)
```

Vectorization and Floating Point

- Suppose we want to sum the elements in an array `x` of 8 elements.
- In a non-vectorized loop, the result would be

```
(((((((x[1]+x[2]) + x[3]) + x[4]) + x[5]) +x[6]) + x[7] + x[8])
```

- With vectorization, the sum would be done in a different order

```
((((x[1]+x[5]) + (x[2] + x[6])) + (x[4]+x[7])) + (x[5] +x[8]))
```

- This is fine if the addition operator can be assumed commutative, but recall that floating point addition is not commutative!
- For this reason, loops involving float operations will not be auto-vectorized in general.

Auto-Vectorization

- If there was a way that we could indicate that the order of operations within the loop doesn't matter, then the compiler could auto-vectorize.
- There is a package called `LoopVectorization` that allows just that.

```
function matmult_avx!(C, A, B)
    @avx for m ∈ axes(A,1), n ∈ axes(B,2)
        Cmn = zero(eltype(C))
        for k ∈ axes(A,2)
            Cmn += A[m,k] * B[k,n]
        end
        C[m,n] = Cmn
    end
end
```

```
julia> @btime matmult_avx!($C, $A, $B);
  3.670 s (0 allocations: 0 bytes)
julia> @btime $A*$B;
  4.726 s (8 allocations: 32.00 MiB)
```

- With one macro, we achieve 30x speed-up with no manual optimization!

More Results

```
julia> A = rand(1:10, 2^10, 2^10);
julia> B = rand(1:10, 2^10, 2^10);
julia> C = rand(1:10, 2^10, 2^10);
julia> @btime matmult_naive!($C, $A, $B);
  6.974 s (0 allocations: 0 bytes)
julia> @btime matmult_trans!($C, $A, $B);
  572.207 ms (2 allocations: 8.00 MiB)
julia> @btime matmult_avx!($C, $A, $B);
  381.784 ms (0 allocations: 0 bytes)
julia> @btime $A*$B;
  571.936 ms (8 allocations: 8.00 MiB)
julia> A = rand(1:10, 2^8, 2^8)
julia> B = rand(1:10, 2^8, 2^8)
julia> C = rand(1:10, 2^8, 2^8)
julia> @btime matmult_naive!($C, $A, $B);
  25.747 ms (0 allocations: 0 bytes)
julia> @btime matmult_trans!($C, $A, $B);
  4.686 ms (2 allocations: 512.08 KiB)
julia> @btime matmult_avx!($C, $A, $B);
  3.133 ms (0 allocations: 0 bytes)
julia> @btime $A*$B
  6.940 ms (8 allocations: 512.41 KiB)
```

- Working with matrices of size 2^{10} is (relatively) faster, due to the Level 2 cache size.
- Note in the results that the native multiplication seems to be using the same trick of taking the transpose, but vectorization is still faster.

Different Integer Types

- For smaller integer types, the results look a bit different.
- `matmult_cache!` now dominates `matmult_trans!`, probably due to the larger number of elements per cache line.

```
julia> B = rand(UInt8(0):UInt8(1), 2^6, 2^6);
julia> A = rand(UInt8(0):UInt8(1), 2^6, 2^6);
julia> C = similar(A);

julia> @btime matmult_trans!($C, $B, $A)
  141.100 µs (1 allocation: 4.19 KiB)

julia> @btime matmult_cache!($C, $B, $A)
  73.400 µs (1 allocation: 4.19 KiB)

julia> @btime matmult_avx!($C, $B, $A)
  6.940 µs (0 allocations: 0 bytes)
```

Other Issues Related to Cache: Memory Alignment

- Because data is always moved in chunks to the cache, you can think of the memory as being divided into chunks the size of a cache line.
- Avoiding data structures that result in object representations straddling a cache-line boundary is another way to improve performance.
- The data structure must fit in cache, otherwise cache misses dominate.

```
function alignment_test(data::Vector{UInt}, offset::Integer)
    n = rand(UInt) # Jump randomly around the memory.
    mask = (length(data) - 9) ⊻ 7
    GC.@preserve data begin # protect the array from moving in memory
        ptr = pointer(data)
        iszero(UInt(ptr) & 63) || error("Array not aligned")
        ptr += (offset & 63)
        for i in 1:4096
            n = (n >>> 7) ⊻ unsafe_load(ptr, (n & mask + 1) % Int)
        end
    end
    return n
end
data = rand(UInt, 256 + 8) # Vector must fit in cache in order to see effect
```

```
julia> @btime alignment_test(data, 0)
  18.300 μs (0 allocations: 0 bytes)
julia> @btime alignment_test(data, 60)
  36.300 μs (0 allocations: 0 bytes)
```

Memory Alignment for Structs

- Alignment issues don't usually arise in practice because compilers usually take care of them automatically.
- For examples, if we create a 7-byte data structure and query, it's layout, in Julia, is reported to take up 8 bytes.
- Because this padding wastes memory (and for other reasons), it is often better to use a "struct of arrays" and than an "array of structs."

```
struct AlignmentTest
    a::UInt32 # 4 bytes +
    b::UInt16 # 2 bytes +
    c::UInt8  # 1 byte = 7 bytes?
end

struct AlignmentTestVector
    a::Vector{UInt32}
    b::Vector{UInt16}
    c::Vector{UInt8}
end
```

Memory Alignment for Structs

Julia allows you to query the memory layout in order to probe these kinds of issues.

```
function get_mem_layout(T)
    for fieldno in 1:fieldcount(T)
        println("Name: ", fieldname(T, fieldno), "\t",
                "Size: ", sizeof(fieldtype(T, fieldno)), " bytes\t",
                "Offset: ", fieldoffset(T, fieldno), " bytes.")
    end
end
```

```
julia> sizeof(AlignmentTest)
Size of AlignmentTest: 8 bytes.

julia> get_mem_layout(AlignmentTest)
Name: a Size: 4 bytes   Offset: 0 bytes.
Name: b Size: 2 bytes   Offset: 4 bytes.
Name: c Size: 1 bytes   Offset: 6 bytes.
```

Arrays of Structs

- Another reason why a struct of arrays is better than an array of structs is that a struct of arrays allows for vectorization.
- This is illustrated in the following experiment.

```
julia> Base.rand(::Type{AlignmentTest}) = AlignmentTest(rand(UInt32), rand(UInt16), rand(UInt8))

julia> N  = 1_000_000

julia> array_of_structs = [rand(AlignmentTest) for i in 1:N];

julia> struct_of_arrays = AlignmentTestVector(rand(UInt32, N), rand(UInt16, N), rand(UInt8, N));

julia> @btime sum(x -> x.a, array_of_structs)
  485.000 µs (1 allocation: 16 bytes)

julia> @btime sum(struct_of_arrays.a);
  93.800 µs (1 allocation: 16 bytes)
```

Memory Allocation

- We have so far avoided the issue of how memory is actually allocated/reserved and how it is deallocated/released again.
- In low-level languages like C, this is done by explicit commands.
- The C command `malloc()` simply asks for a raw block of memory to be allocated and the corresponding command `free()` deallocates it.
- In high-level languages (Julia, Python, Matlab), the memory allocation is hidden, but it's still important to be aware that it has a cost.
- These languages also have an automated system for memory deallocation, often called *garbage collection*.
- Internal pointers are kept for all memory blocks and when the user code no longer has access, the memory is deallocated.

```
myarray = [1, 2, 3, 4]
myarray = nothing
```

- After the pointer is changed, the memory is deallocated automatically.
- When the same happens in C, it results in a *memory leak*.

Cost of Memory Allocation

```julia
function increment(x::Vector{<:Integer})
    y = similar(x)
    @inbounds for i in eachindex(x)
        y[i] = x[i] + 1
    end
    return y
end

function increment!(x::Vector{<:Integer})
    @inbounds for i in eachindex(x)
        x[i] = x[i] + 1
    end
    return x
end
```

```julia
julia> data = rand(UInt, 2^10);
julia> @btime increment(data);
  942.222 ns (1 allocation: 8.13 KiB)
julia> @btime increment!(data);
  77.463 ns (0 allocations: 0 bytes)
```

Stack Versus Heap

- The program has access to two different blocks of RAM.
  - The *stack* is scratch space (generally of a fixed size) pre-allocated at the beginning of execution and can be accessed only in a FIFO manner.
  - The *heap* is memory memory available for dynamic allocation during execution.
- The stack is used to store function parameters, return addresses, local variables.
- Any data whose size is not too big and is known at compile time and whose value won't change can be stored on the stack.
- Stack memory is *much* cheaper to maintain, since there is only one pointer (the stack pointer), whose value changes by one unit at a time.
- On the heap, each block must be allocated/deallocated and has a separate pointer.
- Accessing memory inside the block requires pointer arithmetic.
- All in all, heap memory is relatively much more expensive.

Assembly for Heap Allocation

```
abstract type AllocatedInteger end

mutable struct HeapAllocated <: AllocatedInteger
    x::Int
end
```

```
julia> @code_native debuginfo=:none HeapAllocated(1)
        .text
        pushq   %rbx
        movq    %rsi, %rbx
        movq    %fs:0, %rdi
        addq    $-15712, %rdi               # imm = 0xC2A0
        movabsq $jl_gc_pool_alloc, %rax
        movl    $1400, %esi                 # imm = 0x578
        movl    $16, %edx
        callq   *%rax
        movabsq $140128568651936, %rcx  # imm = 0x7F72398EB0A0
        movq    %rcx, -8(%rax)
        movq    %rbx, (%rax)
        popq    %rbx
        retq
        nopl    (%rax)
```

Assembly for Stack Allocation

```julia
struct StackAllocated <: AllocatedInteger
    x::Int
end

Base.:+(x::Int, y::AllocatedInteger) = x + y.x
Base.:+(x::AllocatedInteger, y::AllocatedInteger) = x.x + y.x
```

```
julia> @code_native debuginfo=:none StackAllocated(1)
        .text
        movq    %rsi, %rax
        retq
        nopw    %cs:(%rax,%rax)
        nop

julia> @btime sum(data_stack)
  363.900 µs (1 allocation: 16 bytes)

julia> @btime sum(data_heap);
  2.320 ms (1 allocation: 16 bytes)
```