ArrayAllocators.jl and NumaAllocators.jl for JuliaCon 2022

Mark Kittisopikul, PhD
Software Engineer II
Scientific Computing, Janelia Research Campus
Howard Hughes Medical Institute

- ArrayAllocators.jl is a Julia package that extends the ability to allocate memory for arrays in Julia.
- NumaAllocators.jl is an ArrayAllocators.jl subpackage that allows array allocation on specific Non-Uniform Memory Access (NUMA)

Motivation

`numpy.zeros` from Python seems faster than `Base.zeros` in Julia.

Python

```
In[2]: %timeit np.zeros((256, 1024, 1024))
31.8 µs ± 2.49 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Julia

```
julia> @time zeros(1024, 1024, 256);
  0.901352 seconds (2 allocations: 2.000 GiB, 0.62% gc time)
```

Array Allocation in Julia

```
# Simple, legacy syntax
zeros(1024, 1024) # Allocate an Array{Float64}, init to 0
zeros(Int, (16, 16)) # Allocate an Array{Int64}, init to 0
ones(UInt8, 1024) # Allocate a vector of bytes, init to 1
trues(32) # Allocate a vector of Bool

# Preferred syntax
A = Array{Float64}(undef, 1024, 1024)
fill!(A, 0) # equivalent to init to 0
```

- Also see `Base.unsafe_wrap`, the basis for this package.

Preferred Julia syntax for arrays

```
ArrayType{ElementType}(initializer, size...)
```

- `ArrayType` - type of the array (e.g. `Array` or `OffsetArray`)
- `ElementType` - type of each component (e.g. `Int` or `Float64`)
- `initializer` - `undef` (undefined), `nothing`, `missing`
- `size` - a tuple of `Int`s describing the dimensions, column major

ArrayAllocators.jl adds new "initializers"

```
Array{UInt8}(malloc, (3,4,5))
OffsetArray{UInt8}(calloc, 16)
Array{UInt8}(MemAlign(2^16), 1024)
```

New allocators and initializers

- `malloc` - Standard C memory allocator. `Libc.malloc`
- `calloc` - Initializes all bytes to `0`. `Libc.calloc`
- `MemAlign` - Align memory to specific boundaries

Calloc: Allocate and initialize to zero

```
julia> using ArrayAllocators

julia> @time C = Array{Int}(calloc, 1024, 1024, 256);
  0.000033 seconds (4 allocations: 2.000 GiB)

julia> @time A = Array{Int}(undef, 1024, 1024, 256);
  0.000037 seconds (2 allocations: 2.000 GiB)

julia> @time Z = zeros(1024, 1024, 256);
  0.671271 seconds (2 allocations: 2.000 GiB, 9.43% gc time)

julia> C == Z # always
true

julia> A == Z # sometimes true
false
```

ArrayAllocators.zeros

- `ArrayAllocators.zeros` is a drop-in replacement for `Base.zeros` using `calloc`.
- Requires ArrayAllocators.jl v0.3 or greater

```
julia> import ArrayAllocators: zeros, ArrayAllocators

julia> @time ArrayAllocators.zeros(1024, 1024, 256);
  0.000041 seconds (4 allocations: 2.000 GiB)

julia> @time zeros(1024, 1024, 256);
  0.000033 seconds (4 allocations: 2.000 GiB)

julia> @time Base.zeros(1024, 1024, 256);
  0.660283 seconds (2 allocations: 2.000 GiB, 0.70% gc time)
```

Discussion on Calloc

See https://discourse.julialang.org/t/faster-zeros-with-calloc/69860

- `calloc` allocates and initializes memory lazily
- `Base.zeros` allocates and initializes memory eagerly
- This can confound your benchmarking. Allocation time may occur when writing to memory!

```
julia> @time Z = zeros(Int8, 1024, 1024);
  0.000324 seconds (2 allocations: 1.000 MiB)

julia> @time fill!(Z, 1);
  0.000138 seconds
```

MemAlign: Aligned memory

Aligning memory can help the processor optimize cache read and write operations, especially when using SIMD.

```
julia> pointer(Array{Int}(undef, 1024))
Ptr{Int64} @0x0000000001349a40

julia> using ArrayAllocators

julia> pointer(Array{Int}(MemAlign(2^16), 1024))
Ptr{Int64} @0x0000000001520000

julia> mod(Int(ans), 2^16)
0
```

NumaAllocators.jl: Non-Uniform Memory Access (NUMA)

- High performance workstations and compute nodes have multiple processors with non-uniform access to memory.
- NumaAllocators.jl allows for allocation of memory on a specific NUMA node
- `numa(node_number)` - initializer for a specific NUMA node number
- Available for Linux (libnuma, numactl) and Windows

NumaAllocators.jl Demonstration

```
julia> using NumaAllocators
a0 = Array{Int8}(numa(0), 1024, 1024);
b0 = Array{Int8}(numa(0), 1024, 1024);
a1 = Array{Int8}(numa(1), 1024, 1024);
b1 = Array{Int8}(numa(1), 1024, 1024);
julia> @time copyto!(b0, a0); # Local -> Local, relatively slow?
  0.000439 seconds

julia> @time copyto!(b1, a0); # Local -> Remote, fastest!
  0.000287 seconds

julia> @time copyto!(b1, a1); # Remote -> Remote
  0.000376 seconds

julia> @time copyto!(b0, a1); # Remote -> Local
  0.000455 seconds
```

Summary

ArrayAllocators.jl

- Provides basic framework to add new array allocators to Julia through the standard `Array{eltype}(init, size...)` syntax
- Provides `calloc` which allows for lazy initialization of zeros like `numpy.zeros`.

NumaAllocators.jl

- Extends ArrayAllocators.jl for high performance computing with multiple processors

Acknowledgements

- Janelia Research Campus, Howard Hughes Medical Institute
  - Lab of Philipp Keller
  - Scientific Computing
    - Stephan Preibisch, Director of Scientific Computing
- Janelia is hiring.
  - https://www.janelia.org/you-janelia/careers
  - https://www.janelia.org/open-science/overview/open-science-software-initiative-ossi/open-positions

Links

Packages

- https://github.com/mkitti/ArrayAllocators.jl
- https://github.com/mkitti/ArrayAllocators.jl/tree/main/NumaAllocators

Documentation

- https://mkitti.github.io/ArrayAllocators.jl/stable/

Topics I did not have time to discuss

Ask a question if interested!

- The `ByteCalculators` submodule
  - Protects against integer overflow when calculating the number of bytes needed.
  - Important for memory safety and security.
- jemalloc and mimalloc extensions
- Custom memory allocators implemented in Julia