Ruby master - Feature #14794

Primitive arrays (Ruby 3x3)

05/29/2018 03:29 PM - ahorek (Pavel Rosický)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

dynamic arrays in ruby can contain various object types:

```
[1, 1.0, 'text', Object.new]
```

however if I create a primitive array, let say only with integers (very common case). It should be more efficient.

```
[1, 2, 3]
```

let me show you an example. I have an array and I want to find a maximum. I can use a native method(:max) or a naive pure ruby implementation.
I expect that the native function will always be much faster because it's written in C right?

```
require 'benchmark/ips'
arr_int = Array.new(50000) { rand 10000 }

def max_ruby(arr)
  max = arr[0]
  size = arr.size
  i = 1
  while i < size
    if arr[i] > max
      max = arr[i]
    end
    i += 1
  end
  max
end

benchmark.ips do |x|
  x.report('native') { arr_int.max }
  x.report('pure')   { max_ruby(arr_int) }
  x.compare!
end
```

here's a comparsion chart of different ruby & python's runtimes (lower is better)
obrazek.png

as expected on ruby 2.6, the native function was faster.

Let's compare it if we use a JIT
native – no difference
pure ruby – sometimes even faster than native

It's because JIT can't do anything with native functions (inlining, type checks etc.). Native fuctions should be as fast as possible.

MRI implementation of rb_ary_max
https://github.com/ruby/ruby/blob/trunk/array.c#L4335

```
for (i = 0; i < RARRAY_LEN(ary); i++) {
       v = RARRAY_AREF(ary, i);
       if (result == Qundef || OPTIMIZED_CMP(v, result, cmp_opt) > 0) {
           result = v;
       }
}
```

this is great for mixed arrays, but for primitive arrays it's quite ineffective. C compiler can't optimize it.
1/ unbox it if possible, don't dereference objects and load it in chunks
2/ if <=> is not redefined, we can use a simplier algorithm
3/ it's a trivial example that can be written in SIMD #14328, there's even a special instruction for it
https://software.intel.com/en-us/node/524201
C compiler can do it for us, but this metod it too complex for auto-vectorization and the data type isn't known during compile time.

Array max is just an example, but the same strategy could be applied for many other methods.

I found a great article about it, check it out.

http://tratt.net/laurie/research/pubs/html/bolz_diekmann_tratt__storage_strategies_for_collections_in_dynamically_typed_languages/

I think this feature could speed-up real ruby applications significantly and it shouldn't be very hard to implement. We also don't have to change ruby syntax, define types etc. No compatibility issues.

History

#1 - 05/30/2018 02:26 AM - mrkn (Kenta Murata)

Use numo-narray or nmatrix for homogeneous numeric arrays.

https://github.com/ruby-numo/numo-narray

https://github.com/SciRuby/nmatrix

#2 - 05/30/2018 01:27 PM - ahorek (Pavel Rosický)

I'm interested to improve Ruby array's performance without specifying custom types or C extensions, it should just work out of the box.

```
Numo::Int32.new(1,100)
Array.new(100, type: :integer)
```

or something like that isn't very idiomatic for Ruby. I don't want to care about types, but I expect that Ruby will store it efficiently and change the store strategy if necessary:

```
[1,2,3] << 'test'
```

narray can't handle this case or numeric overflows...

but yes it's faster

```
             Numo::Int32 max: 20723.0 i/s
             Numo::Int64 max: 11975.4 i/s - 1.73x  slower
             ruby max (int):  8191.7 i/s  - 2.53x  slower
```

```
The flexibility and dynamism of dynamically typed languages frustrates most traditional static optimisations. Just-In-Time (JIT) compilers defer optimisations until run-time, when the types of objects at specific points in a program can be identified, and specialised code can be generated. In particular, variables which reference common types such as integers can be 'unboxed' [8 , 24]: rather than being references to an object in the heap, they are stored directly where they are used. This lowers memory consumption, improves cache locality, and reduces the overhead on the garbage collector. Unboxing is an important technique in optimising such languages.

Dynamically typed languages therefore pay a significant performance penalty for the possibility that collections may store heterogeneously typed elements, even for programs which create no such collections. Statically typed languages can determine efficient storage representations of collections storing elements of a primitive type based on a collection's static types.
```

#3 - 05/30/2018 01:58 PM - ahorek (Pavel Rosický)

btw: 40% of arrays on my rails app contains only primitive elements