Dynamic arrays in Ruby can contain various object types:

```ruby
[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.

```ruby
[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?

```ruby
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 comparison chart of different Ruby & Python's runtimes (lower is better)

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 functions should be as fast as possible.

MRI implementation of `rb_ary_max`

https://github.com/ruby/ruby/blob/trunk/array.c#L4335

```c
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
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.


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 Rosicky)
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 referen

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

#4 - 10/07/2021 12:00 AM - dsisnero (Dominic Sisneros)
saw from https://chrisseaton.com/truffleruby/rubykaigi21/ - the shape of ruby objects. Maybe if we can get this implemented it would help out with this. Or use collection storage strategies to implement this.
https://tratt.net/laurie/research/pubs/html/bolz_diekmann_tratt__storage_strategies_for_collections_in_dynamicallyTyped_languages/

#5 - 12/03/2021 02:41 AM - mrkn (Kenta Murata)
I guess what you want is like pandas.Series. Is my speculation is correct?

As far as I understand, this ticket proposes an optimization of the Array representation, for example, by using raw memory of long* when all the elements are Fixnums, or by using double* when all are Floats. The idea is interesting, but I believe we should start discussing this after anyone implements it and performs an experiment with some real-world applications like Rails.

Aside from SIMD, such an optimization will reduce GC time because we don't have to mark the elements of Fixnum and Float arrays. However, it may introduce additional overhead when degenerating to the current VALUE* representation occurs frequently. What we need is an experiment.

the same performance could be achievable by a C extension or a specialized type like Numo::Int32, but these are just performance hacks. What I'm proposing is that the language should determine, that an array [1,2,3] contains integers only, so it should be stored using raw memory of long* instead of objects which reference the actual value.

such an optimization will reduce GC time

sure. It'll also reduce memory consumption & cache locality.

knowing the type has other advantages. For instance, the original example with Array#max is hard to optimize by the compiler

```c
for (i = 0; i < RARRAY_LEN(ary); i++) {
    v = RARRAY_AREF(ary, i); // we need to dereference the value
    if (result == Qundef || OPTIMIZED_CMP(v, result, cmp_opt) > 0) {
        result = v;
    }
}
```

// a bunch of type checks for each iteration

```c
#define OPTIMIZED_CMP(a, b, data) \
    (((FIXNUM_P(a) && FIXNUM_P(b) && CMP_OPTIMIZABLE(data, Integer)) ? \ 
        ((long)a > (long)b) ? 1 : ((long)a < (long)b) ? -1 : 0) : \ 
        (STRING_P(a) && STRING_P(b) && CMP_OPTIMIZABLE(data, String)) ? \ 
        rb_str_cmp(a, b) : \ 
        (RB_FLOAT_TYPE_P(a) && RB_FLOAT_TYPE_P(b) && CMP_OPTIMIZABLE(data, Float)) ? \ 
        rb_float_cmp(a, b) : \ 
        rb_cmpint(rb_funcallv(a, id_cmp, 1, &b), a, b))
```

@mrkn (Kenta Murata) did an optimization, but they're even more type checks which make the code complicated

https://github.com/ruby/ruby/pull/3325/files
case array_shape
when Integers
for (i = 0; i < RARRAY_LEN(ary); i++) {
    if ((long)vmax < (long)v) {
        vmax = v;
    }
    return vmax;
}
end

this pattern is very well known and easily optimizable because there're no type checks for each element anymore. There's also a high chance these values will be prefetched or read from a cache.  
https://godbolt.org/z/ieEa8roPn

it's applicable to most array methods, but Array#max is simple to explain.

However, it may introduce additional overhead when degenerating to the current VALUE*representation occurs frequently

you're right, there're bad scenarios that could actually lead to a slower performance

array = [1,2,3] -> Integer array
array << "string" -> Object array (needs reallocation of the whole array instead of just putting a new element
)

there're ways how to overcome this, but it's the hardest part to implement without a serious performance impact. It's a corner case that should still work, but it isn't very common. Most arrays aren't changing shapes at runtime like this.

besides https://github.com/oracle/truffleruby and https://www.pypy.org which have the optimization implemented, there's also an old pull request about the same idea https://github.com/topazproject/topaz/pull/555 (topaz is a dead project today)

the idea itself isn't hard to understand. I'm not as good in C to implement it to CRuby, but I'll gladly help if someone comes with initial implementation.

Files
clipboard-202112111356-0rasx.png  17.8 KB  12/11/2021  ahorek (Pavel Rosicky)