Implementing Arrays and Tensors with Ruby and libJIT

J. Wedekind, H. Abdul-Rahman, M. Howarth, K. Dutton
Materials and Engineering Research Institute
Sheffield Hallam University
Sheffield S1 1WB, UK
{J.Wedekind,H.Abdul-Rahman,M.Howarth,K.Dutton}@shu.ac.uk

A. J. Lockwood
Department of Engineering Materials
The University of Sheffield
Sheffield S1 3JD, UK
A.Lockwood@sheffield.ac.uk

31.9.2009

Abstract

Implementing numerical algorithms typically demands a trade-off between readability and performance of the code. However the way we write programs affects our understanding of the problems we are trying to solve. Statically typed languages represent numbers and arrays with primitive types. This means that the most relevant data types for signal processing have only very limited capabilities in those languages. The dynamically typed programming language Ruby is pure object-oriented and makes it easy to implement algorithms which will work on arrays with elements of any numerical type. However current Ruby interpreter is lacking in terms of processing speed. This paper presents a Ruby extension which makes use of the libJIT just-in-time compiler. Dynamical typing is used to analyze closures and translate corresponding element-wise operations for arrays to machine code. The same approach is used to compile tensor operations thus allowing the use of Einstein notation in Ruby.

1 Introduction

Developers of real-time computer vision systems need to address multiple challenges and constraints. The demand of achieving general applicability conflicts with the requirement to achieve real-time performance. In practise numerical algorithms are often developed and tested using a computer algebra system and later on manually ported and optimized. In general terms the problem is that of a widening semantic gap between the abstract execution model and the actual hardware architecture. There is a need for virtual machines which provide an abstract execution model without impacting on performance too much.

Recently there has been a renewed interest in dynamically typed programming languages[6]. While they are often rejected due to performance concerns, the potential savings in developer-time should be a compelling reason to invest effort to overcome these issues. A publication by Roman *et al.*[4] demonstrates that robotic projects can greatly benefit from the dynamic properties of the Ruby programming language.

In this paper we show that it only requires a moderate effort to combine Ruby and the just-in-time compiler library libJIT[1] to bring together high performance and the agility of a dynamically typed language in a single system.

In the next section we show present approaches and the motivation of our work. In Section 3 we use dynamic typing for just-in-time compilation of array- and tensor-operations. In Section 4 we compare the resulting performance to that of a traditional ahead-of-time compilation. Section 5 shows extraction of features and descriptors as an example of a concise implementation created with this approach. In Section 6 we conclude and suggest future work.

2 State of the art

2.1 Static library for static language

Most object oriented, statically typed languages have a split type system. There are primitive types which directly correspond to registers of the hardware and there are object types which support inheritance and dynamic dispatch.

In C++ not only integers and floating point numbers but also arrays are primitive types. Unfortunately these are the relevant data types for representing images. To implement a basic operation such as adding two values so that it will work on different types, one needs to make extensive use of template meta-programming. Another problem is that when a developer wants to modify one aspect of the system, the static typing can force numerous rewrites in unrelated parts of the source code[6].

To avoid this problem the OpenCV library does not use the type system of the programming language to handle different pixel types of images. The information about the pixel type is stored in member variables of the image type instead. However this means that it is not possible to use dynamic dispatch to implement algorithms which have to work on different types of images. The required mechanisms have to be implemented explicitly.

[1]http://www.gnu.org/software/dotgnu/libjit-doc/libjit.html

Listing 1: Numerical types in Ruby

```
require 'mathn'
require 'complex'
x = -8 / 6
# -4/3
y = x * Complex( 1, 2 )
# Complex(-4/3, -8/3)
z = 2 ** 80
# 1208925819614629174706176
y + z
# Complex(3626777458843887524118524/3, -8/3)
```

Listing 2: Higher-order functions in Ruby

```
arr = [ 2, 3, 5, 7 ]
arr.collect { |x| x * x }
# [4, 9, 25, 49]
arr.inject( 0 ) { |a,b| a + b }
# 17
```

2.2 Ruby programming language

There are several computer vision libraries which provide extensions to a dynamically typed language. For example EasyVision, Camellia, Lush, Gamera are libraries offering extensions to such different languages as Haskell, Ruby, LISP, and Python. OpenCV provides bindings for Python as well. When writing such an extension one typically uses a multi-layered approach. For example when using Ruby, the Ruby application uses a module written in Ruby. This module in turn wraps the library which is written in C/C++[7].

Listing 1 illustrates how Ruby uses dynamical typing to support scientific computing. Intermediate results are shown in comment lines (preceded by "#"). Ruby also offers the higher-order functions "Array.collect" and "Array.inject" for performing mapping or accumulating operations (see listing 2). For further details about the properties of the Ruby programming language we refer to Y. Matsumoto's article[3].

Listing 3: Array operations using Ruby and NArray

```
n = NArray.int( 4 ).indgen! 1
# NArray.int(4):
# [ 1, 2, 3, 4 ]
m = NArray.scomplex( 4 ).fill! Complex::I
# NArray.scomplex(4):
# [ 0.0+1.0i, 0.0+1.0i, 0.0+1.0i, 0.0+1.0i ]
n + m
# NArray.scomplex(4):
# [ 1.0+1.0i, 2.0+1.0i, 3.0+1.0i, 4.0+1.0i ]
```

2.3 Static library for dynamic language

When implementing a module for computer vision, one needs to deal with various combinations of pixel types and operations. Existing computer vision extensions either support only a limited set of all possible combinations, or the bindings do not hide the properties of the static type system beneath very well.

A notable exception is M. Tanaka's NArray[2]. NArray is a Ruby extension implemented in C. It provides fast operations for multi-dimensional arrays with elements of a single type. Listing 3 shows some array manipulations and type coercions using NArray.

2.4 Partially dynamic library

However even the NArray implementation has limitations. It is not possible to add support for a new element-type to the array class without modifying the C source code and recompiling the extension.

To address this issue one can implement array classes which allow definition of custom element-types[8]. To accomplished this, the array data types are largely implemented in Ruby. Custom element-types are supported by adding native methods for performing basic operations on an array of elements of that type.

Table 1 for example shows a customisable list of native methods for filling one-dimensional arrays. In Ruby the existence of a method with a certain name can be checked during run-time using the method "Object.respond_to?". The Ruby part of the extension contains glue code which tries to invoke an efficient native method before falling back to using a slower generic implementation (see listing 4).

[2]http://narray.rubyforge.org/SPEC.en

Table 1: Native methods for filling arrays

method	element-type
`Sequence.`	
`fill_ubyte`	8-bit unsigned integers
`fill_byte`	8-bit signed integers
`fill_usint`	16-bit unsigned integers
`fill_sint`	16-bit signed integers
...	...

Listing 4: Ruby wrapper method for filling an array

```
class Sequence
  def fill!( value )
    message = "fill_#{@typecode.name.downcase}"
    if Sequence.respond_to? message
      Sequence.send message, @data, @size, value
    else
      for i in 0...@size
        self[ i ] = value
      end
    end
    return self
  end
end
```

3 Combining Ruby and libJIT

3.1 Just-in-time compilers

There is recent work aimed at improving the general performance of the Ruby virtual machine by adding just-in-time compiler support to the Ruby virtual machine[5]. This raises the question of whether implementing a Ruby extension for the mere purpose of increasing the performance of machine vision algorithms will soon be unnecessary. However as shown in listing 1, Ruby avoids numeric overflow by performing range checks and instantiating a big number object when required. Furthermore the array class of the Ruby Core library allows arrays to

Table 2: Just-in-time compilers

	LLVM	libJIT	lightning	Asmjit	Xbyak
register allocation	✓	✓	×	×	×
platform independence	✓	✓	✓	×	×
global optimization	✓	×	×	×	×

have elements of different type. This means that for the foreseeable future there is a need for efficient data types which reflect the behavior of the hardware more closely.

While there still is a need for efficient data types, it makes sense to make use of a just-in-time compiler for implementing them. All approaches presented in Section 2 require instantiation and ahead-of-time compilation of all possible combinations of element-types and operations. This leads to large binaries and a lengthy compilation phase. Using a just-in-time compiler is much more economical.

Table 2 shows several software projects which can be used to perform just-in-time compilation. As one can see the level of support varies greatly. On the one hand there are libraries which just offer methods for writing machine code to memory and running it. On the other hand there are libraries which offer platform independence, automatic register allocation, and global optimization. We have written a Ruby extension for using libJIT. We have preferred libJIT over LLVM because it is a much smaller library and the simple application programming interface makes adoption easy.

3.2 One-dimensional arrays

Figure 1 shows an example of how the JIT-compiler can be used from within Ruby. The method "JITFunction.compile" accepts a closure as an argument (the beginning and end of the closure is marked by the keywords "do" and "end"). The closure is interpreted by passing "JITTerm"-objects as parameters "a" and "b". The objects represent (virtual) registers containing the results of reading the first and second parameter from the stack. Executing "a + b" then creates a new "JITTerm"-object which represents a new register containing the result of adding "a" and "b".

Listing 5: Compiling and calling a binary operation

```
Sequence.define_binary_jit_op( "+" ) { |x,y| x + y }
s = Sequence.ubytergb( 3 ).indgen! RGB( 3, 2, 1 ), 1
# Sequenceubytergb(3):
# [RGB( 3, 2, 1 ), RGB( 4, 3, 2 ), RGB( 5, 4, 3 )]
t = Sequence.sint( 3 ).indgen! -2, 2
# Sequencesint(3):
# [-2, 0, 2]
r = s + t
# Sequencesintrgb(3):
# [RGB( 1, 0, -1 ), RGB( 4, 3, 2 ), RGB( 7, 6, 5 )]
```

Using meta-programming and just-in-time compilation it is possible to write a method which accepts a method name and a closure containing a scalar operation and then defines an element-wise array operation. Such a method can then be used as shown in listing 5 to compile and subsequently call an element-wise "+" operator (here the closure begins with "{" and ends with "}"). Note that once the generic meta-programming methods are implemented, one line of code is sufficient to create a "+"-operator with JIT-support for all combinations of the known types (here 24-bit unsigned RGB triplets and 16-bit signed integers). Using meta-programming all unary and binary element-wise operations can be implemented swiftly. In a similar way to the NArray Ruby extension, the convention of implementing the "coerce" method is used to also facilitate scalar-array and array-scalar operations.

Table 3 gives an overview of generic array operations which we have found to occur in computer vision algorithms. "a" and "b" are parameters, and "r" is the result. Each of "r", "a", and "b" is a scalar or an array depending on the operation. In some cases a function "f" is involved. "i" and "j" are loop variables. However this set of operations is neither minimal nor complete. Further study is needed to decide whether there is a finite and complete set of generic array operations and what a minimal set would consist of.

```
plus = JITFunction.compile( JITType::INT,
                            JITType::INT,
                            JITType::INT ) do |f,a,b|
  a + b
end
plus.call 2, 3
# 5
```

```
function f0(i1:int,i2:int):int
  incoming_frame_posn(i1, 8)
  incoming_frame_posn(i2, 12)
  i5 = i1 + i2
  return_int(i5)
  ends_in_dead
end
```

```
;function f0(int,int):int
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
add    0xc(%ebp),%eax
pop    %ebp
ret
```

Figure 1: Using libJIT with x86 back-end from within Ruby

i	*shape*[*i*]	*strides*[*i*]	*x*[*i*]
0	2	1	1
1	4	2	2
2	3	8	0

$$strides[i] = \prod_{k=0}^{k} shape[k]$$

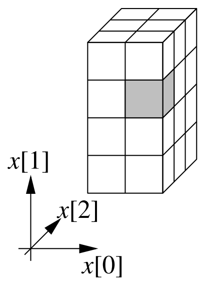


Figure 2: Shape and strides for a three-dimensional array

3.3 Multi-dimensional arrays and tensors

The difference between multi-dimensional arrays and one-dimensional arrays is merely the way they are indexed. Figure 2 shows an array with three dimensions and $2 \times 4 \times 3 = 24$ elements and the element with $x = [1, 2, 0]$ being accessed. Internally the element is stored in a one-dimensional array and the index is $1 * 1 + 2 * 2 + 8 * 0 = 5$.

The operations listed in table 3 all have their multi-dimensional counterpart. However in addition to this there are tensor operations. A notable example of a library which supports tensor operations is the FTensor library[2]. FTensor uses C++ templates to support the use of Einstein notation (with up to four indices) in a C++ program. Listing 6 shows how one can perform a matrix multiplication. Tensor operations are very generic. Multiplication, trace, and transpose of matrices are just a few special cases of tensor operations.

However using Ruby and libJIT it is possible to implement high performance tensor operations with support for an arbitrary number of indices and arbitrary combinations of element-types and operations, while providing automatic determination of the result's element-type and dimensions.

Listing 6: Tensor operations with the FTensor library

```
Index< 'i', 3 > i;
Index< 'j', 3 > j;
Index< 'k', 3 > k;
Tensor2< double, 3, 3 > r, a, b;
r( i, k ) = a( i, j ) * b( j, k );
```

As shown above (see figure 1) it is possible to compile a closure to machine code. In a similar fashion one can use dynamic typing to analyze a closure specifying a tensor operation. *E.g.* the tensor method call in listing 7 specifies a result with one index. The closure takes two variables as arguments. The first variable is the array index of the result. The other variable "j" becomes a summation index. The analysis of the closure will determine the usage of the variables as shown in table 4. Note that although only one array (the array "m") is used in the tensor operation, the two occurrences need to be distinguished ("arg0" and "arg1"), since the variable "j" is used as a different index in each case. The array strides (see figure 2) describe the offset between two successive data elements along each dimension. They are used to generate a fast implementation based on pointer arithmetic which is equivalent to the naive implementation shown in listing 8.

Table 3: Generic set of array operations

operation	loop body	loop variable
write element	r[b] =a	-
read element	r =a[b]	-
write sub-array	r[b+i]=a[i]	i
read sub-array	r[i] =a[i+b]	i
fill	r[i] =a	i
index array	r[i] =i	i
unary function	r[i] =f(a[i])	i
binary function	r[i] =f(a,b[i])	i
binary function	r[i] =f(a[i],b)	i
binary function	r[i] =f(a[i],b[i])	i
accumulate	r =f(r,a[i])	i
warp/mask	r[i] =a[b[i]]	i
unmask	r[b[i]]=a[i]	i
downsampling	r[i] =a[b*i]	i
upsampling	r[b*i] =a[i]	i
integral	r[i] =r[i-1]+a[i]	i
map	r[i] =b[a[i]]	i
histogram	r[a[i]]=r[a[i]]+1	i
weighted hist.	r[a[i]]=r[a[i]]+b[i]	i
correlation	r[i] =r[i]+a[i+j]*b[j]	i,j

4 Performance

Figure 3 shows different operations and the time required for performing them 1000 times with HornetsEye (Ruby 1.8.6 and libJIT 0.1.2), NArray (Ruby 1.8.6 and GCC 4.1.3), and a naive C++ implementation (G++ 4.1.3). The tests were performed on an AMD Duron™ 1.2GHz Processor. The arrays "m" and "n" are single-precision floating point arrays with 500×500 and 100×100 elements.

The results show that in general HornetsEye takes only about twice as much processing time as the C++ implementation. The C++ code is faster due to sophisticated optimization by the GNU C++ compiler. The fact that NArray is almost as fast as the C++ implementation shows that the overhead incurred by Ruby is negligible. We are not sure why the matrix multiplication in C++ is slower than in HornetsEye. A possible reason is a conflict of loop optimisation and the behaviour of the memory cache.

Listing 7: Tensor operation in Ruby

```
m = MultiArray.int( 3, 3 ).indgen!
# MultiArrayint(3,3):
# [ [ 0, 1, 2 ],
#   [ 3, 4, 5 ],
#   [ 6, 7, 8 ] ]
r = tensor( 1 ) { |i,j| m[i,j] * m[j,1] }
# MultiArrayint(3):
# [ 42, 54, 66 ]
```

Table 4: Use of array indices in listing 7

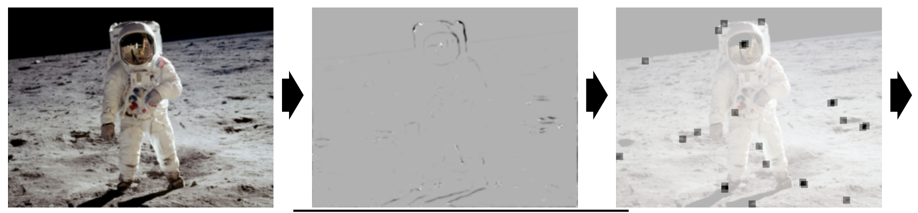
variable	parameter	array index
i	result	0
i	arg0	0
j	arg0	1
j	arg1	0

5 Concise implementations

Listing 9 shows an implementation of corner detection, non-maxima extraction, and extraction of feature patches. Ruby has open classes. This means that a class definition can be altered even after objects of this class have been instantiated (*e.g.* lines 1–30 in listing 9 add methods to the class "MultiArray"). In line 32 the color image is loaded.

Listing 8: Naive implementation of tensor operation in listing 7

```
r = MultiArray.int 3
for i in 0...3
  v = 0
  for j in 0...3
    v += m[i,j] * m[j,1]
  end
  r[i]=v
end
```



i	0	1	...
locs[i] % w	186	138	...
locs[i] / w	18	20	...

descr =

Figure 4: Computing feature locations and descriptors (“w” is the width of the image)

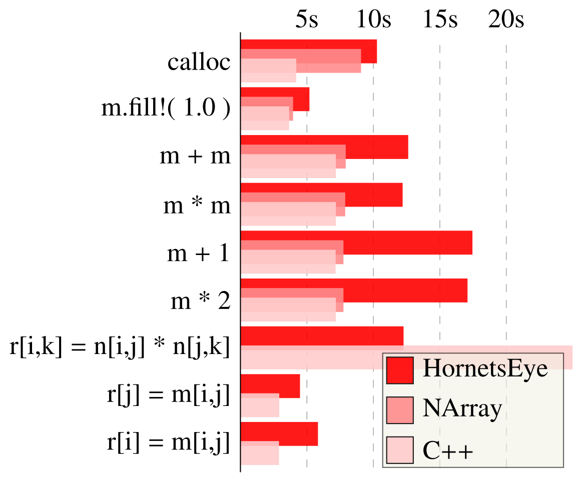


Figure 3: Performance comparison of different array operations

In line 33 the image is converted to gray scale and the feature extraction method is called. Lines 15–20 are an implementation of the Harris-Stephens[1] edge- and corner detector (see figure 4). Line 21 then calls non-maxima suppression which is implemented using thresholding and comparison with the dilated image (local maximum in a 3×3 area). The method “crop” discards features near the boundary to avoid problems when extracting the descriptors later. The method call finally returns a boolean image with the feature locations.

Line 34 then calls the “descriptors” method for extracting local feature patches. In line 24 an index-array is generated. In line 25 an array of offsets is computed. The masking operation in line 26 creates a one-dimensional array with the indices of the feature locations. Line 27 then uses a tensor operation to compute a three-dimensional array with indices pointing to each pixel which is part of a descriptor. Line 28 then calls a warp function which uses the index array as a lookup table for computing the result. The result is a three-dimensional array where each slice is a descriptor of a feature (see figure 4).

6 Conclusion and future work

We have shown how developers of computer vision software can benefit from the properties of a dynamically typed language such as Ruby. Our approach only requires wrappers for accessing the just-in-time compiler and the input/output-libraries.

Improving the performance (see figure 3) could be accomplished by using a just-in-time compiler which supports similar optimization methods as GNU C++. It is also possible to use C++ on-the-fly (using the Ruby-Inline extension). Another possibility is to integrate GPU hardware using native methods and glue code in a simi-

Listing 9: Computing feature locations and descriptors

```ruby
class MultiArray
  def crop( fringe )
    mask = MultiArray.bool( *@shape ).fill!
    mask[ fringe...(@shape[0]-fringe),
          fringe...(@shape[1]-fringe) ] = true
    self.and mask
  end
  def threshold( value, fringe )
    ( self >= max * value ).crop fringe
  end
  def maxima( value, fringe )
    threshold( value, fringe ).and eq( dilate )
  end
  def features( thresh, fringe, sigma, k )
    gx, gy = gauss_gradient sigma
    cov = [ gx ** 2, gy ** 2, gx * gy ]
    a, b, c = cov.collect { |arr| arr.gauss_blur sigma }
    trace = a + b
    determinant = a * b - c ** 2
    feat = determinant - k * trace ** 2
    feat.maxima( thresh, fringe )
  end
  def descriptors( img, r )
    id = MultiArray.int( *@shape ).indgen!
    patch = id[ 0..( 2 * r ), 0..( 2 * r ) ] - id[ r, r ]
    locs = id.mask self
    field = tensor( 3 ) { |i,j,k| patch[i,j] + locs[k] }
    img.to_sequence.warp( field )
  end
end
t, r, s, k = 0.17, 4, 1.0, 0.1
img = MultiArray.load_rgb24 'astronaut.jpg'
msk = img.to_sfloat.features t, r, s, k
descr = msk.descriptors img, r
```

lar fashion to that shown in Section 2.4. Furthermore one could parallelize algorithms using multi-threading which is supported by Ruby version 1.9.

We have shown that it is possible to develop algorithms for signal processing without having to compromise on performance or abstraction. Research aimed at a stronger theoretical foundation of machine vision algorithms could benefit from this approach. *E.g.* eigentransforms and normalization of descriptors as well as feature similarity matrices can be implemented using tensor operations. There is also potential for implementing new data types such as hypercomplex numbers and elements of a Lie algebra. However more research is required to apply the results of this work to implementations of feature matching algorithms such as Geometric Hashing, RANSAC, or the Hough Transform.

This work was supported by the Nanorobotics EPSRC Basic Technology grant GR/S85696/01. We also would like to acknowledge the help of Kirill Kononenko and Klaus Treichel from the libJIT team.

References

[1] C. G. Harris and M. Stephens. A combined corner and edge detector. *Proceedings 4th Alvey Vision Conference*, pages 147–151, 1988. 7

[2] W. Landry. Implementing a high performance tensor library. *Scientific Programming*, 11(4):273–90, 2003. 5

[3] Y. Matsumoto. The Ruby programming language. *informIT*, June 2000. `http://www.informit.com/articles/article.aspx?p=18225`. 2

[4] B. Roman et al. Controlling a robotic marine environmental sampler with the Ruby scripting language. *Journal of the Association for Laboratory Automation*, 12(1):56–61, 2007. 1

[5] K. Sasada. Future of Ruby VM. Oral presentation at RubyConf, 2008. `http://www.atdot.net/~ko1/activities/rubyconf2008_ko1.pdf`. 3

[6] L. Tratt and R. Wuyts. Guest editors' introduction: Dynamically typed languages. *IEEE Software*, 24(5):28–30, 2007. 1, 2

[7] S. Vinoski. Ruby extensions [dynamic language application]. *IEEE Internet Computing*, 10(5):85–7, 2006. 2

[8] J. Wedekind et al. A machine vision extension for the Ruby programming language. In *2008 International Conference on Information and Automation (ICIA)*, pages 991–6, June 2008. 3