CLEM Matrix Performance

Cyrus L. Harmon

April 23, 2007

1 CLEM Performance

1.1 Introduction

Common Lisp is a high-level language that is a modern-day member of the LISP family of languages. Common Lisp can be either interpreted or compiled, or both, depending on the implementation and modern compilers offer the promise of performance on the order of that achieved by C and fortran compilers. CLEM is a matrix math package for Common Lisp that strives to offer high-performance matrix math routines, written in Common Lisp. Common Lisp has a sophisticated type system and it is a goal of CLEM to offer matrix representation and opreations that exploit the features of this type system. Furthermore, Common Lisp has a sophisticated object model, the Common Lisp Object System (CLOS), and CLEM uses features of CLOS and its companion metaobject system, the Meta-object Protocol (MOP) to define its classes, objects and methods.

Common Lisp implementations vary greatly in their peformance, but the Common Lisp standard provides an infrastructure for high-performance-capable implementations to generate efficient code through the use of compiler declarations of types and optimization settings. Lisp implementations are free to compile lisp code to native machine code, rather than either interpreting the lisp code on the fly, or compiling the code to a byte-code representation, that is then executed by a virtual machine. This suggests that, subject to the limits placed on the output by the common lisp language and to the implementation details of the particular lisp system, a lisp environment should be able to produce code that approaches the speed and efficiency of other compiled languages. SBCL is a Common Lisp implementation that compiles to native code and has a sophisticated compiler called, somewhat confusingly, Python, although the Python compiler predates the Python interpreted language by many years. SBCL's compiler uses type and optimization declarations to generate efficient code by producing optimized routines that are specific to the declared types, often representing data in an "unboxed" format, ideally one that matches the type representation handled directly by the CPU. One of CLEM's main goals is to provide efficient matrix operations that utilize the capabilities of the lisp compiler to generate efficient code.

1.1.1 Boxed and Unboxed Representation of Lisp Data Objects

Boxed representations of lisp values are generally stored as tagged data objects where the tags serve to identify the type of the particular object. Unboxed representations, on the other hand, are merely represented by the data values themselves, without any identifying metadata being stored with the value. Obviously, the language environment needs to know, or to be able to determine, the type of a particular data value. For boxed values, the language environment can rely on the fact that the type information is stored with the data value directly. For unboxed values, the language system must keep track of the value stored in a particular directly. The main advantage of using unboxed datatypes is that the language environment, or compiled code it produces, does not have to bother extracting the type information from the boxed value and extracting the data as appropriate. However, this has the disadvantage that the type of data is then fixed to be that represented by the unboxed type. Often, hybrid representations are used in structures such as an array, where the array itself will be a dynamically typed, boxed object while the values of the array will be unboxed values placed in an a particular region of memory. By using unboxed values for the elements of the array, the code can directly access these values without the overhead of "unboxing" the data. Boxed memory access often allocates memory, or "conses" in lisp parlance, as a storage area is needed to hold the unboxed value. In summary, sing boxed datatypes introduces (at least) two important areas where work has to be done by the language environment to access the data, the allocation of temporary to store the resulting values that will be unboxed from the data, and additional work on the part of the code to unbox the data in the first place.

For these reasons, it is higly desirable for CLEM to use unboxed data where possibly. In the inner loop of a matrix operation, for instance, accessing a boxed data type can introduce substantial performance penalties. Therefore, CLEM has gone to great lengths to provide matrix representations that yield unboxed values and operations that can operate on these values with allocation of a minimum amount of memory.

1.1.2 Avoiding Unneccessary Memory Allocation

2 CLEM Design Goals

2.1 CLEM Implementation Choices

3 Matrix Data Representation

What options do we have for storing matrix data? Main choices are lisp arrays or an external block of memory. Are there other options here?

3.1 Reification of Matrix Data Objects

Defering, for a moment, the question of what form the actual matrix data will take, let us consider the form of the matrix object itself. It could be the object that represents the data directly, or it could be an object, such as an instance of a class or struct, that contains a reference to the object that holds the data. In a sense, the simplest approach to providing matrix arithmetic operations is just to use common lisp arrays both to hold the data and to be the direct representation of the matrix object. The CLEM design assumes that there is additional data, besides the data values stored in the array, or what have you, that will be needed and that just using a lisp array as the matrix itself is insufficient.

3.2 Common Lisp Arrays

Lisp arrays have the advantage that they are likely to take advantage of the lisp type system. Yes, an implementation may choose to ignroe this information, or continue to produce the same code as it would for untyped arrays, but a sufficently smart compiler, such as Python, should use the array type information to produce efficient code. Of course this also means that in order to get this efficiency we have to provide this type information to the compiler. As we try to modularize various pieces, it is often the case that one would like to have generic code that can work on matrices of any type. It these cases, additional measures may be needed to coax the compiler into generating efficient code, while doing so in a generic manner.

3.2.1 One-dimensional or Multi-dimensional Arrays?

One issue in dealing with lisp arrays is whether to use the lisp facilitty for multi-dimensional array. One argument in favor of native multi-dimensional arrays is that the compiler can generate efficent code to access data in certain multi-dimensional arrays, provided that this information is known and passed to the compiler at compile-time. On the other hand, using one-dimensional arrays puts both the burden of and the flexibility of computing array indices on the matrix package.

3.2.2 Extensible arrays?

3.2.3 What About Lists?

Lists are convenient for representing matrices in that the iteration functions can be used to traverse the elements of the matrix, yielding the famous trivial transpose operation using mapcar and list. However, lists aren't designed for efficient random access and are a poor choice for representing anything but trivially small matrices.

It is worth mentioning the possibility of other approaches, such as an external block of memory, perhaps allocated with either non-standard routines of the lisp system, or via a foreign-function interface, and to determine the offsets into this block of memory, coerce the contents to a given lisp type and obtain the results. This approach is used by some libraries that use clem, such as ch-image, to access matrix/array data stored in memory in a non-native-lisp form, such as matlisp matrices (which are really BLAS/LAPACK matrices), fftw matrices, and arrays of data from TIFF images. While this is nice to be able to do, it is unlikely to be practical for storing matrix data, given the alternative of using lisp arrays. Coercing of the contents of the matrix to lisp types is done for us by optimized code in the compiler. It is unlikely that we would be able to do a better job of this than the compiler. This approach is useful for conversion of matrix data to other in-memory formats, but unlikely to be useful for the typed lisp matrices for which CLEM is designed. If we were to go this route, it would make sense to use other, optimized, code libraries for operating on this data, and this is what matlisp does, handing these blocks of memory off to BLAS/LAPACK for processing in highly-optimized routines written in C, Fortran or Assembly Language.

4 Matrix Data Access

4.1 Now that we have chosen a matrix representation, how do we access the data in it?

4.2 Slow and Fast Data Access Paths

4.3 Flexibility

4.3.1 Resizing Matrices

4.3.2 Matrix Index "Recycling"

4.4 Macros

4.5 Compiler Macros

4.6 SBCL-specific Compiler Features

4.6.1 defknown/deftransform/defoptimizer

5 Benchmarks

We'll start with some simple benchmarks of matrix operations and examine the effect of the various implementation strategies for representing matrices on these operations.

5.1 2-Dimensional Lisp Arrays

```
(defparameter b1
  (make-array '(1024 1024) :element-type 'double-float :initial-element 1.0d0
              :adjustable nil :fill-pointer nil))
B1

(defparameter b2
  (make-array '(1024 1024) :element-type 'double-float :initial-element 1.0d0
              :adjustable nil :fill-pointer nil))
B2

(defparameter b3
  (make-array '(1024 1024) :element-type 'double-float :initial-element 1.0d0
              :adjustable nil :fill-pointer nil))
B3
```

Now, our function to add the two arrays:

```
(defun bench/add-matrix/aref (a1 a2 a3)
  (destructuring-bind
      (rows cols)
      (array-dimensions a1)
    (dotimes (i rows)
      (dotimes (j cols) (setf (aref a3 i j) (+ (aref a1 i j) (aref a2 i j)))))))
BENCH/ADD-MATRIX/AREF
```

Now, we time how long it takes to run bench/addmatrix/aref:

```
(ch-util:time-to-string (bench/add-matrix/aref b1 b2 b3))
"Evaluation took:
  0.215 seconds of real time
  0.1958 seconds of user run time
  0.018307 seconds of system run time
  [Run times include 0.083 seconds GC run time.]
  0 calls to %EVAL
  0 page faults and
  50,331,032 bytes consed.
"
```

5.2 1-Dimensional Lisp Arrays

5.3 A CLOS object holding a reference to a 2-Dimensional Lisp Array

5.4 A CLOS object holding a reference to a 1-Dimensional Lisp Array