Performance Beyond Expectations

Lynn H. Quam
quam@ai.sri.com

Abstract

The performance of Common Lisp based Image Understanding Systems has been significantly improved by the careful choice of declarations, object representations, and method dispatch in a small number of low-level primitives. In matrix multiplication and image pixel access, the performance achieved is within a factor of two of optimized C code. Effective Lisp compiler register allocation, fast CLOS slot access, and fast generic function dispatch are critical. For large grain operations, performance can be further increased using foreign function libraries. The paper closes with a "laundry list" of features that a Common Lisp implementation should provide for improving performance.

Keywords: Common Lisp, numerical performance, CLOS performance, matrix multiply, object-oriented image access, foreign functions.

1 INTRODUCTION

It is widely believed that Common Lisp, particularly CLOS, has inherently poor performance for numerical and image related tasks. This paper presents techniques applied to image pixel access and matrix multiplication to achieve high levels of performance by the careful use of declarations, closures, foreign functions, and a few tricks. These examples were chosen because they represent preformance critical code fragments used in real systems.

2 BACKGROUND

The techniques presented here are based on over 20 years of experience by the author in implementing Lisp-based Image Understanding Systems at SRI International (Figure 1). In the early 1980s, ImagCalc [3] was implemented on the Symbolics Lisp Machine, which had a delightful, single-user Lisp environment and user-friendly GUI. Coordinate transforms and 3D models were added to ImagCalc to form the SRI Cartographic Modeling Environment (CME)[4]. DARPA and ORD sponsored the RADIUS Common Development Environment (RCDE)[6, 5] in the 1990s. RCDE was a reimplementation of CME on Sun and SGI Unix systems using Lucid Common Lisp, X11/Motif, and a small foreign function library needed for performance. FREEDIUS is an open source reimplementation of CME, started in late 1990s and continuing today, supporting broader variety of platforms using CMUCL and Allegro CL.

3 BENCHMARKS

The example code has been benchmarked on the configurations shown in Figure 5. For brevity, most of the performance results are shown only for the CMUCL/X86/Linux configuration. Considerable effort has been taken to make the benchmarks fair to both CMUCL and Allegro, and to all hardware platforms.[1]

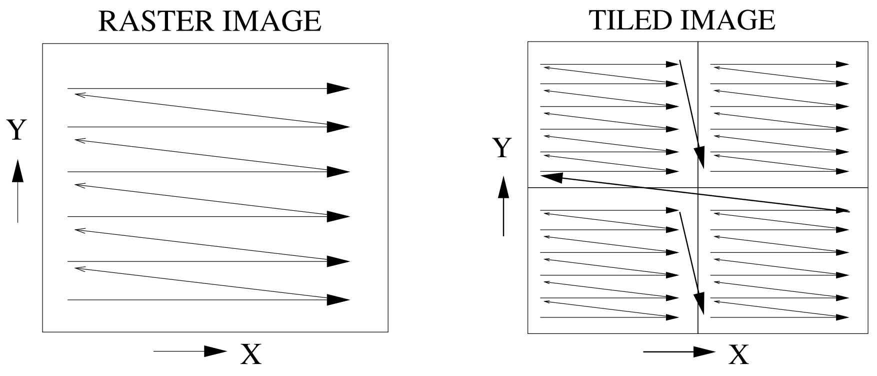
4 IMAGE REPRESENTATION AND PIXEL ACCESS

An object-oriented representation of images is essential to support the wide variety of image storage schemes needed in real world applications (Figure 2). This section demonstrates a CLOS image representation with efficient polymorphic pixel accessors using generic functions, closures, and foreign functions.

[1]Source code for the benchmarks and results are available at `http://www.ai.sri.com/~quam/Public/papers/ILC2005`.

	ImagCalc/CME 1980s	RCDE 1990s	FREEDIUS 2000s
Lisp impl.	Genera	Lucid CL	CMUCL & Allegro
HW/OS	LispM	Sparc/SunOS MIPS/Irix	PPC/MacOSX Sparc/SunOS X86/Linux X86/WinNT
Dev. tools	Zmacs	Emacs & ILisp	Emacs & SLIME
GUI		Motif & X11	Tcl/Tk & OpenGL
Image repn.	Flavors	CLOS & structs	structs
Image access	Lisp	Lisp & FFI	Lisp & FFI
Matrix fns.	Lisp	FFI	FFI

Figure 1: History of Lisp-based image understanding systems developed at SRI.



Mapped-Array-Image: Tiled or untiled.
Raster-Array-Image: 2d array storage order. Appropriate for small images.
Paged-Image: Tiles paged on demand from file. Essential for large images. Individual tiles may be compressed.
Lazy-Image: Tiles computed on demand.
Video: Frames grabbed on demand from file.

Figure 2: Image storage schemes

4.1 Skeleton Image Classes and Accessors

```
(defclass image ()
    ((xdim :initarg :xdim :reader image-xdim)
     (ydim :initarg :ydim :reader image-ydim)
     (element-type :initform '(unsigned-byte 8) :initarg :element-type)
     xmap ymap pixels))

(deftype image-map-element-type () '(signed-byte 32))
(deftype image-map () `(simple-array image-map-element-type (*)))

(defmethod initialize-instance :after ((image image))
  (with-slots (xdim ydim xmap ymap element-type pixels) image
    (setf ymap (make-array ydim :element-type image-map-element-type)
          xmap (make-array xdim :element-type image-map-element-type)
          pixels (make-array (* ydim xdim) :element-type element-type))
    ;; Default maps for raster storage order
    (loop for x fixnum from 0 below xdim do (setf (aref xmap x) x))
    (loop for y fixnum from 0 below ydim do (setf (aref ymap y) (* y xdim)))))

(defclass array-image-unsigned-byte-8 (image) ())
```

IREF* is presented here to support the benchmark showing the poor performance of generic functions for single pixel access.

```
(defmacro fix+(x y) `(the fixnum (+ (the fixnum ,x) (the fixnum ,y))))

(defmethod iref* ((image array-image-unsigned-byte-8) x y)
  (with-slots (xmap ymap pixels) image
    (declare (type (simple-array (unsigned-byte 8) (* *)) pixels)
             (type image-map xmap ymap))
    (aref pixels (fix+ (aref xmap x) (aref ymap y)))))
```

A (SETF IREF*) accessor is similarly defined.

4.2 IREF-Based Image Add Operators

The IMAGE-ADD operators defined next are presented for the purpose of benchmarking the performance of the single pixel accessors. Image operators that can be written in terms of row-at-a-time operations should be implemented using the GETROW/PUTROW accessors as shown later.

```
(defun iref*-image-add (a b c)
  (loop with xdim fixnum = (image-xdim a)
        with ydim fixnum = (image-ydim a)
        for y fixnum from 0 below ydim
        do (loop for x fixnum from 0 below xdim
                 do (setf (iref* c x y) (fix+ (iref* a x y) (iref* b x y)))))
  c)
```

For comparison purposes, we also define a similar operator 2D-ARRAY-ADD using AREF and 2d-arrays.

Figure 3 clearly shows that IREF* is very slow compared to 2D-AREF. Careful examination of disassembled code shows that most of the time is spent in generic function dispatch. This is addressed in the next refinement of the image class definitions and IREF accessors.

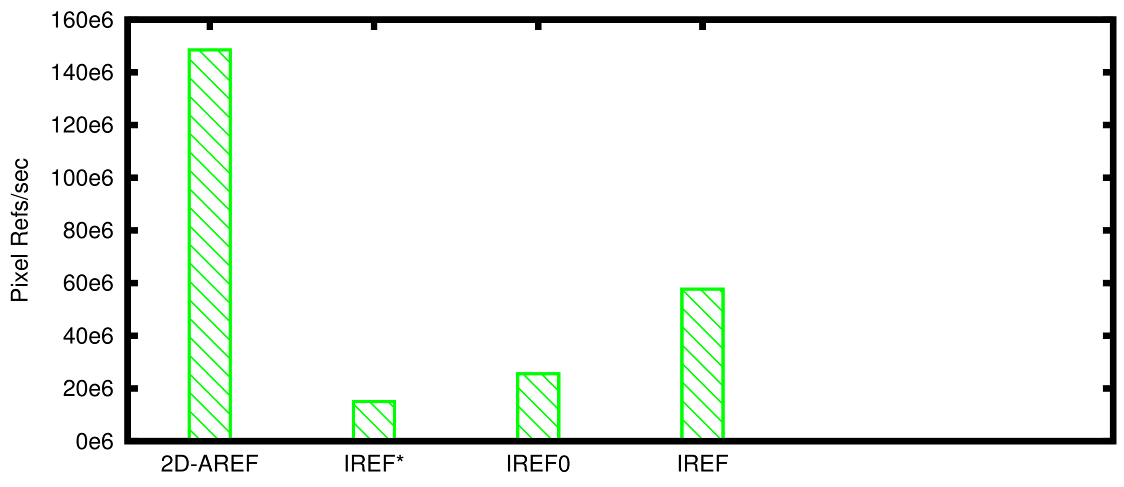


Figure 3: Single pixel access performance for CMUCL/AthlonXP2800+/Linux .

4.3 Refined Image Class and Accessors

```
(defclass image ()
    ((xdim :initarg :xdim :reader image-xdim)
     (ydim :initarg :ydim :reader image-ydim)
     (element-type :initform '(unsigned-byte 8) :initarg :element-type)
     (iref-fn :reader image-iref-fn)
     (iset-fn :reader image-iset-fn)
     xmap ymap pixels))
```

By defining IREF-FN inside a closure, we convert standard-object slot accesses into closure binding accesses.

```
(defmethod initialize-instance :after ((image array-image-unsigned-byte-8))
  (with-slots (iref-fn xmap ymap pixels)
    (let ((xmap xmap) (ymap ymap) (pixels pixels))
      ;; *****  This is the lexical scope of the IREF-FN  ***********
     (flet ((iref-fn (x y)
               (declare (type fixnum x y))
               (aref (the (simple-array (unsigned-byte 8) (*)) pixels)
                     (fix+ (aref (the image-map xmap) x) (aref (the image-map ymap) y)))))
        (setf iref-fn #'iref-fn)))))
```

We replace calls to the IREF* generic function with calls to the IREF-FN of the image instance.

```
(defmacro iref (image x y)
  `(funcall (the function (image-iref-fn ,image) ,x ,y)))

(defun iref-image-add0 (a b c)
  (loop with xdim fixnum = (image-xdim a)
        with ydim fixnum = (image-ydim a)
        for y fixnum from 0 below ydim
        do (loop for x fixnum from 0 below xdim
                 do (setf (iref c x y) (fix+ (iref a x y) (iref b x y)))))
  c)
```

To eliminate the relatively slow calls to the IMAGE-IREF-FN accessor inside the loops, another macro, WITH-IMAGE-ELEMENTS, is implemented in a manner that changes the macroexpansion of IREF as shown in the macroexpanded version of IREF-IMAGE-ADD.

```
(defun iref-image-add (a b c)
  (with-image-elements (a b c)
    (loop with xdim fixnum = (image-xdim a)
          with ydim fixnum = (image-ydim a)
          for y fixnum from 0 below ydim
          do (loop for x fixnum from 0 below xdim
                   do (setf (iref c x y) (fix+ (iref a x y) (iref b x y))))))
  c)

;;; MACROEXPANDED
(defun iref-image-add (a b c)
  (let ((a-iref-fn (image-iref-fn a))
        (b-iref-fn (image-iref-fn b))
        (c-iset-fn (image-iset-fn c))
    (declare (type function a-iref-fn b-iref-fn c-iset-fn))
    (loop with xdim fixnum = (image-xdim a)
          with ydim fixnum = (image-ydim a)
          for y fixnum from 0 below ydim
          do (loop for x fixnum from 0 below xdim
                   do (setf (funcall c-iref-fn c x y)
                            (fix+ (funcall a-iref-fn x y) (funcall b-iref-fn x y))))))
  c))
```

As is shown in Figure 3, these IREF optimizations have steadily increased the performance of out-of-line IREF to 40% the speed of inlined 2D-AREF, which is the best that can be expected from a polymorphic implementation of IREF.

In order to support a variety of image storage schemes, the multi-pixel accessors GETROW and PUTROW are implemented using generic functions and methods. For the sake of brevity, the PUTROW method, the class definitions for the RASTER-ARRAY-IMAGE class, and its GETROW and PUTROW methods are omitted from this paper.

```
(defmethod image-getrow ((image array-image-unsigned-byte-8) buf x y &optional n)
  (declare (fixnum x y) (type (or null fixnum) n) (type array buf))
  (with-slots (xdim pixels ymap) image
    ;; Simplified version permitting only fixnum buffer type.
    (copy-mapped-u8-to-ibuf pixels buf (or n xdim) (aref (the image-map ymap) y) xmap x))))

(defun copy-mapped-u8-to-ibuf (pixels ibuf n pixels-off map map-idx)
  (declare (type (simple-array (unsigned-byte 8) (*)) pixels))
  (declare (type (simple-array integer-buffer-element-type (*)) ibuf))
  (declare (type image-map map))
  (declare (fixnum pixels-off n map-idx))
  (loop for i fixnum from 0 below n
        do (setf (aref ibuf i) (aref pixels (fix+ (aref map map-idx) pixels-off)))
           (incf map-idx)))
```

COPY-MAPPED-U8-TO-IBUF can be replaced by a foreign function for better performance as shown in Figure 4.

4.4 IMAGE-ADD using GETROW/PUTROW

```
(defun image-add (a b c)
  (let* ((abuf (make-row-buffer a))
         (bbuf (make-row-buffer b))
         (cbuf (make-row-buffer c)))
    (declare (type integer-buffer abuf bbuf cbuf))
    (loop with xdim fixnum = (image-xdim a)
```

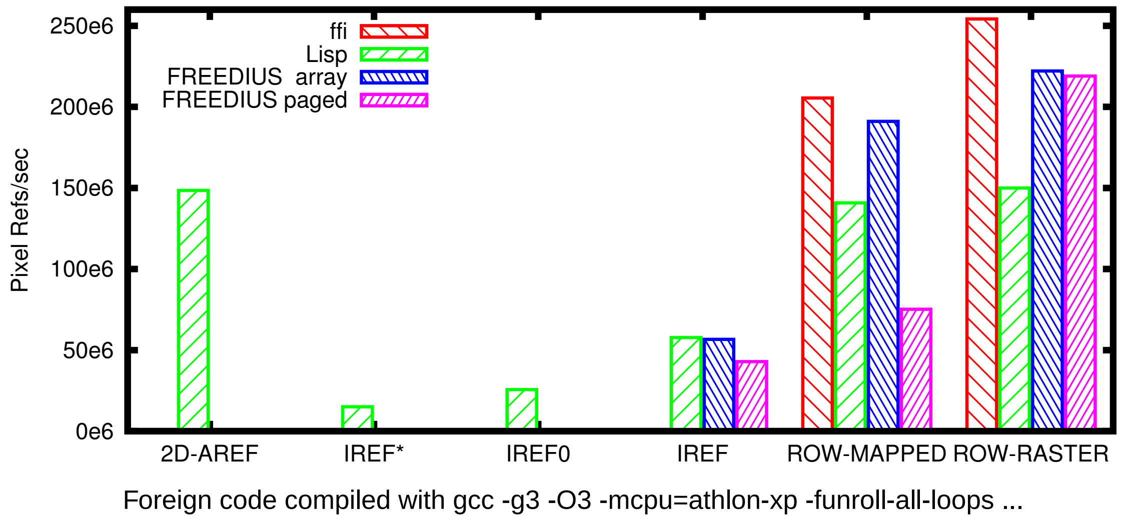


Figure 4: IMAGE-ADD Performance for CMUCL/AthlonXP2800+/Linux including FREEDIUS benchmarks.

```
         with ydim fixnum = (image-ydim a)
         for y fixnum from 0 below ydim
         do (image-getrow a abuf 0 y)
            (image-getrow b bbuf 0 y)
            (loop for x fixnum from 0 below xdim
                  do (setf (aref cbuf x) (fix+ (aref abuf x) (aref bbuf x))))
            (image-putrow c cbuf 0 y))
   c))
```

Figure 4 shows the performance of IMAGE-ADD for the four combinations of LISP vs FFI versions of GETROW and PUTROW combined with mapped-image vs raster-image. The performance of both Lisp versions of GETROW and PUTROW are about the same as 2D-AREF. The FFI versions are significantly faster, with the raster version being 1.7 times the speed of the 2D-AREF version. The figure also shows that the IREF and GETROW performance of this CLOS implementation of images compares well with the struct-based implementation used in FREEDIUS. The GETROW FFI performance shown is better than that of FREEDIUS because some suboptimalities in the FREEDIUS implementation.

5 MATRIX MULTIPLY

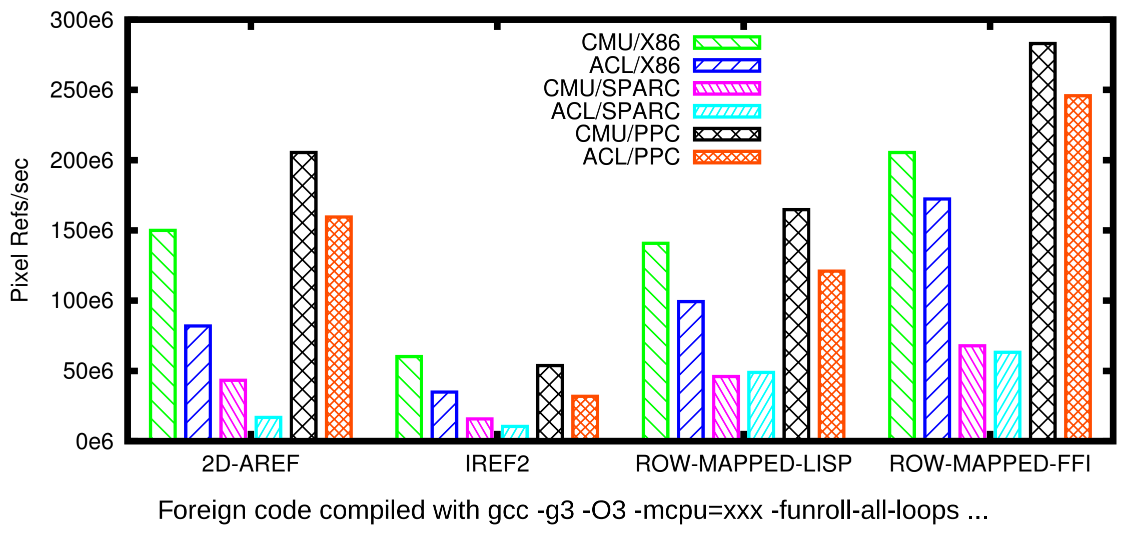
Double-float matrix multiply provides a useful benchmark of Lisp compilation of loops involving AREF and double-float multiply and add, as well as the integer operations needed for loop control and indexing. We define it as follows:

$$C_{n_i \times n_j} = A_{n_i \times n_k} \cdot B_{n_k \times n_j}$$

$$c_{i,j} = \sum_k a_{i,k} * b_{k,j}$$

Loop order is not specified

As trivial as the definition may look, there are a many ways to implement it based on rearrangements of loop order and blocking. The LAPACK [2] and ATLAS [7, 1] linear albegra packages provide highly efficient Fortran and C implementations of linear algebra operations.



Processor/Operating System	CMUCL-19a	Allegro CL 7.0
AMD Athlon XP2800+/Linux	x	x
dual 1200MHz Sun-Blade-1000/SunOS	x	x
dual G5 PowerPC/MacOSX	x	x

Figure 5: Image Access Performance for all Tested Architectures

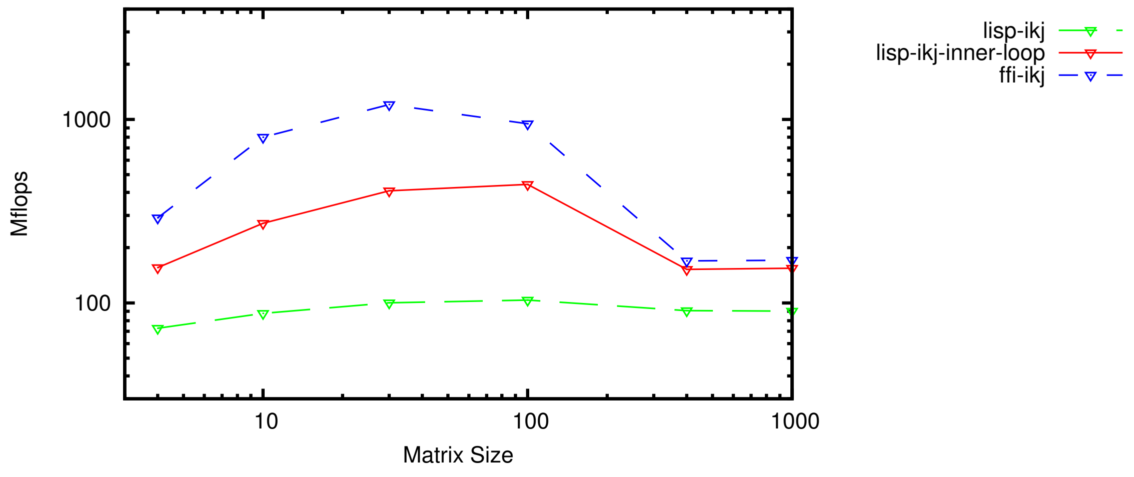
5.1 Matrix Multiply Performance Issues

Listed in approximate order of importance:

- Register allocation - key problem for Lisp compilers
- Use of architecture submodel instruction set
- Loop unrolling
- Memory reference patterns
  - Loop order IJK, IKJ, JIK, ...
  - Block algorithms are essential for huge matrices
- L1 cache size
- Instruction pipeline issues

5.2 IKJ Loop Order Matrix Multiply

```
// IKJ loop order matrix multiply based on LAPACK.
void matmul_ikj (int ni, int nk, int nj, double *A, double *B, double *C)
{
  double *Arowi = A, *Crowi = C, *Browk = B;
  for (int i=0; i < ni; i++, Arowi += nk, Crowi += nj) {
    for (int j=0; j < nj; j++) Crowi[j] = 0;
    for (int k=0; k < nk; k++, Browk += nj) {
      double aik = Arowi[k];
      for (int j=0; j < nj; j++)
```



Foreign code compiled with gcc -g3 -O3 -mcpu=athlon-xp

Figure 6: IKJ Matrix Multiply Performance for CMUCL/AthlonXP2800+/Linux

```
        // compute C[i][j] += A[i][k]*B[k][j];
        Crowi[j] += aik * Browk[j];
 }}}
```

```
;;; Unoptimized Lisp IKJ Loop Order Matrix Multiply using 2D AREF
(defun multiply-matrices-ikj (a b)
  (let* ((ni (array-dimension a 0))
         (nk (array-dimension a 1))
         (nj (array-dimension b 1))
         (c (make-array (list ni nj) :initial-element 0)))
    (loop for i from 0 below ni
          do (loop for k from 0 below nk
                   for aik = (aref a i k)
                   do (loop for j from 0 below nj
                            do (incf (aref c i j) (* aik (aref b k j))))))
    c))
```

Optimized Lisp IKJ Loop Order Matrix Multiply using 1D AREF. “Reduction in Strength” has been applied to optimize the loop index calculations.

```
(defun matmul-1d-ikj (a b &optional c)
  (declare (optimize (speed 3) (safety 0)))
  (let* ((ni (array-dimension a 0))
         (nk (array-dimension a 1))
         (nj (array-dimension b 1))
         (c (or c (make-dmatrix ni nj t)))
         (a1d (underlying-simple-vector a))
         (b1d (underlying-simple-vector b))
         (c1d (underlying-simple-vector c)))
    (declare (type (simple-array double-float (*)) a1d b1d c1d))
    (declare (fixnum ni nk nj))
    (loop for i fixnum from 0 below ni
          for i*nk fixnum from 0 by nk
          for i*nj fixnum from 0 by nj
          for i*nj+nj fixnum from nj by nj
          do (loop for k fixnum from i*nj below i*nj+nj
                   do (setf (aref c1d k) 0.0d0))
```

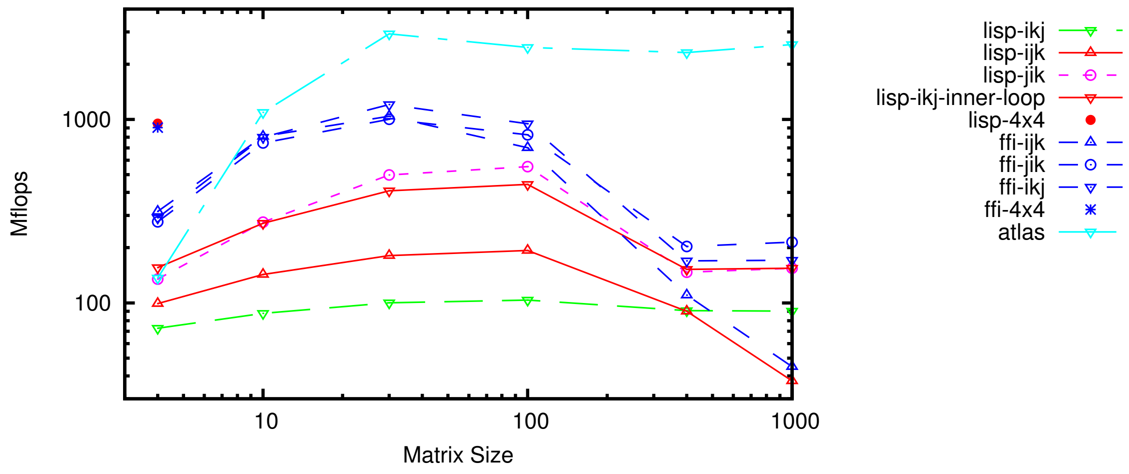


Figure 7: Atlas, IJK, JIK and IKJ Matrix Multiply Performance for both Lisp and FFI code. The lisp-jik performance is about 60% of the performance of the best of the FFI versions. Also note the lisp-4x4 performance is slightly better than the ffi-4x4 performance, at close to 1 GFLOP! The ATLAS performance at nearly 3 GFLOP approaches the limits of the Athlon XP2800+, and scales well to much larger matrices than are shown.

```
                (loop for k fixnum from 0 below nk
                      for k*nj fixnum from 0 by nj
                      for i*nk+k fixnum from i*nk
                      for aik double-float = (aref a1d i*nk+k)
                      do (loop for k*nj+j fixnum from k*nj
                               for i*nj+j fixnum from i*nj below i*nj+nj
                               do (incf (aref c1d i*nj+j) (* aik (aref b1d k*nj+j))))))
   c))
```

In CMUCL/X86, moving the inner loop of MATMUL-1D-IKJ into a separate function achieves “optimal” register allocation resulting in a 4.4 times speedup.

6 FOREIGN CODE CAVEATS

The biggest problem I have encountered with the use of foreign function interfaces is the possibility of the Lisp garbage collector moving objects in memory while foreign code is retaining a pointer to the object, leading to possible memory corruption and the probable death of the Lisp process. There are two primary ways this can occur:

- During the execution of foreign code, a callback to Lisp allows code to run that triggers a garbage collection. Versions of CMUCL with “generational conservative garbage collection” (GENCGC) prevent objects pointed to by registers or the stack from moving during garbage collection. Non-conservative versions of CMUCL and all of the Allegro versions do not protect against this problem.

- During the execution of foreign code, an interrupt or a multiprocessor thread allows Lisp to run code that triggers a garbage collection resulting in the moving of the Lisp object. A complete discussion of FFI issues related to interrupts, threads, and Lisp processes is beyond the scope of this paper.

Here are some rules for solving this problem:

- Do not allow foreign code to retain pointers to Lisp objects beyond the lifetime of the call to the foreign function. With GENCGC versions of CMUCL, this is the only rule needed.

- For non-GENCGC CMUCL or Allegro:

  - Prevent interrupts, foreign code callbacks, and multi-processor threads while running foreign functions.

    OR

  - Allocate stationary arrays: Allegro supports this well. In CMUCL, FREEDIUS implements MAKE-FOREIGN-VECTOR, which allocates an array in (stationary) malloc-space which appears to Lisp like an ordinary Lisp array. The difficultly with this approach is that the foreign-vectors must be explicitly freed since the garbage collector cannot reclaim unused objects in malloc-space.

7 RECOMMENDED ENHANCEMENTS TO COMMON LISP

Here are some enhancements to Common Lisp implementations that would make it easier to achieve better performance.

- Lisp Compiler
  - Better type inference (CMUCL is good here) would simplify the writing of efficient code.
  - Better register allocation. AMD64 and PowerPC should allow for major improvements here.
  - Machine submodel instruction support. Neither CMUCL nor Allegro support newer X86 instructions.
  - Open-code AREF for an array of type (AND ARRAY (NOT (SIMPLE-ARRAY 1))). This would facilitate fast access AREF to malloced foreign arrays.
- CLOS
  - Fast slot access (CMUCL supports inside methods).
- Generic Functions
  - Faster dispatching.
  - Optimized effective methods.
  - Eliminate dispatching when a generic function has only one method. The generic function becomes the code for the single method.
  - Allow inlining when a “sealed” generic function has only one method.
- Foreign Function Interface
  - Direct support for array arguments. Allegro supports; easy to do in CMUCL. This is a major shortcoming of UFFI.
  - Callbacks to Lisp: Both Allegro and CMUCL support this. This is a major shortcoming of UFFI.
- Arrays
  - UNDERLYING-SIMPLE-VECTOR to get the 1 dimensional vector “underlying” a multi-dimensional array.
  - MAKE-STATIONARY-ARRAY to allocate a Lisp GCed array that will not move in memory.
  - MAKE-FOREIGN-ARRAY to construct a Lisp accessable array which is displaced to foreign allocated memory. Example: (make-foreign-array <dims> :displaced-to <foreign-address>)
- Garbage Collector
  - Generational GC for performance. Allegro supports on all architectures and CMUCL on X86 and Sparc.
  - Stationary GC area (Allegro supports), and/or Conservative GC (CMUCL supports on X86 and Sparc) for safely passing arrays into foreign code without worrying about callbacks, threads and interrupts.

8 CONCLUSION

I have shown examples where, contrary to popular myth, Common Lisp can compile very efficient code for image access and matrix operations, with additional performance possible using the foreign function interface.

I selected the title of the paper based on my surprise to discover a few examples where the CMUCL X86 compiler, due to exceptionally good register usage, produces exceptional numerical code: inline 4x4 matrix multiply at nearly 1 GFLOP and JIK loop order matrix multiply at .55 GFLOP. The image access benchmarks in Figure 5 show that compiled Lisp code can perform at about 60% the speed of maximally optimized, loop-unrolled C code compiled for the specific submodel instruction sets. This gives me hope that the performance of Lisp compilers will improve further on the newer architectures, such as the AMD64 and PowerPC, which have more registers. I am also encouraged by the recent CLOS performance enhancements to CMUCL, which make slot access access within methods as fast as inline struct slot access.

That is *performance beyond expectations!*

REFERENCES

[1] Atlas homepage. `http://math-atlas.sourceforge.net/`.

[2] Lapack homepage. `http://www.netlib.org/lapack`.

[3] A.J. Hanson, A.P. Pentland, and L.H. Quam. Design of a prototype interactive cartographic display and analysis environment. In *DARPA Image Understanding Workshop*, pages 475–482. Morgan Kaufman, 1987.

[4] A.J. Hanson and L.H. Quam. Overview of the SRI Cartographic Modeling Environment. In *DARPA Image Understanding Workshop*, pages 576–582. Morgan Kaufman, 1988.

[5] A. J. Heller, P. Fua, C. Connolly, and J. Sargent. The Site-Model Construction Component of the RADIUS Testbed System. In *ARPA Image Understanding Workshop*, pages 345–355, 1996. Also available from `http://www.ai.sri.com/ajh/papers/smc2-corrected.pdf`.

[6] Aaron J. Heller and Lynn H. Quam. The RADIUS Common Development Environment. In Oscar Firschein and Tom Strat, editors, *RADIUS: Image Understanding for Imagery Intelligence*. Morgan Kaufmann, San Mateo (CA), 1997. Also available from `http://www.ai.sri.com/ajh/papers/rcde-corrected.pdf`.

[7] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. Available from `http://www.netlib.org/lapack/lawns/lawn131.ps`.