# Beating C in Scientific Computing Applications

## On the Behavior and Performance of LISP, Part 1

Didier Verna

EPITA Research and Development Laboratory
14–16 rue Voltaire, F-94276 Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

## Abstract

This paper presents an ongoing research on the behavior and performance of LISP with respect to C in the context of scientific numerical computing. Several simple image processing algorithms are used to evaluate the performance of pixel access and arithmetic operations in both languages. We demonstrate that the behavior of equivalent LISP and C code is similar with respect to the choice of data structures and types, and also to external parameters such as hardware optimization. We further demonstrate that properly typed and optimized LISP code runs as fast as the equivalent C code, or even faster in some cases.

***Keywords*** C, LISP, Image Processing, Scientific Numerical Calculus, Performance

## 1. Introduction

More than 15 years after the standardization process of COMMON-LISP [1] (Steele, 1990), and more than 20 years after people really started to care about performance (Gabriel, 1985; Fateman et al., 1995; Reid, 1996), LISP still suffers from the legend that it is a slow language.

As a matter of fact, it is very rare to find efficiency demanding applications written in LISP. To take a single example from a scientific numerical calculus application domain, image processing libraries are mostly written in C (Froment, 2000) or in C++ with various degrees of genericity (bib, 2002; Ibanez et al., 2003; Duret-Lutz, 2000). Most of the time, programmers are simply unaware of LISP, and in the best case, they falsely believe that sacrificing expressiveness will get them better performance.

We must admit however that this point of view is not totally unjustified. Recent studies (Neuss, 2003; Quam, 2005) on various numerical computation algorithms find that LISP code compiled with CMU-CL can run at 60% of the speed of equivalent C code. People having already made the choice of LISP for other reasons might call this "reasonable performance" (Boreczky and Rowe, 1994), but people coming from C or C++ will not: if you consider an image processing chain that requires 12 hours to complete for instance (and this is a real case), running at a 60% speed means that you have just lost a day. This is unacceptable.

Hence, we have to do better: we have to show people that they would lose *nothing* in performance by using LISP, the corollary being that they would gain *a lot* in expressiveness.

This article presents the first part of an experimental study on the behavior and performance of LISP in the context of numerical scientific computing; more precisely in the field of image processing. This first part deals with fully dedicated (or "specialized") code: programs built with full knowledge of the algorithms and the data types to which they apply. The second and third part will be devoted to studying genericity, respectively through dynamic object-orientation and static meta-programming. The rationale here is that adding support for genericity would at best give equal performance, but more probably entail a loss of efficiency. So if dedicated LISP code is already unsatisfactory with respect to C versions, it would be useless to try and go further.

In this paper, we demonstrate that given the current state of the art in COMMON-LISP compiler technology (most notably, open-coded arithmetics on unboxed numbers and efficient array types (Fateman et al., 1995, sec. 4, p.13), the performances of equivalent C and LISP programs are comparable; sometimes better with LISP.

Section 2 describes the experimental conditions in which our performance measures were obtained. In section 3 on the next page, we study the behavior and performance of several specialized C programs involving two fundamental operations of image processing: pixel access and arithmetic processing. Section 4 on page 5 introduces the equivalent LISP code and studies its behavior comparatively to C. Finally, section 5 on page 10 summarizes the benchmarking results we obtained with both languages and several compilers.

## 2. Experimental Conditions

### 2.1 The Protocol

Our experiments are based on 4 very simple algorithms: pixel assignment (initializing an image with a constant value), and pixel addition / multiplication / division (filling a destination image with the addition / multiplication / division of every pixel from a source image by a constant value). These algorithms involve two fundamental atomic operations of image processing: pixel (array element) access and arithmetic processing.

In order to avoid benchmarking any program initialization side-effect (initial page faults *etc.*), the performances have been measured on 200 consecutive iterations of each algorithm.

Additionally, the behavior of the algorithms is controlled by the following parameters:

**Image size** Unless otherwise specified, the benchmarks presented in this paper are for $800 * 800$ images. However, we also ran the tests on smaller images with an increased iterations number. Some interesting comparisons between the two image sizes are presented in sections 3.5 on page 4 and 4.6 on page 9.

**Data type** We have benchmarked both integer and floating-point images, with the corresponding arithmetic operations (operand of the same type, no type conversion needed). Additionally, no arithmetic overflow occurs in our programs.

**Array types** We have benchmarked 2D images represented either directly as 2D arrays, or as 1D arrays of consecutive lines

[1] COMMON-LISP is defined by the X3.226-1994 (R1999) ANSI standard.

(see also 2.2). Some interesting comparisons between these two storage layouts are presented in sections 3.1 on the next page and 4.3 on page 6.

**Access type** We have benchmarked both linear and pseudo-random image traversal. By "linear", we mean that the images are traversed as they are represented in memory: from the first pixel to the last, line after line. By "pseudo-random", we mean that the images are traversed by examining in turn pixels distant from a constant offset in the image memory representation (see also section 2.3).

**Optimization** We have benchmarked our algorithms with 3 different optimization levels: unoptimized, optimized, and also with inlining of each algorithm's function into the iterations loop. The exact meaning of these 3 optimization levels will be detailed later. For simplicity, we will now simply say "inlined" to refer to the optimized *and* inlined versions.

The combination of all these parameters amounts to a total of 48 actual test cases for each algorithm, for a total of 192 individual tests. Not all combinations are interesting; we present only comparative results where we think there is a point to be made.

The benchmarks have been generated on a Debian GNU/Linux[2] system running a packaged `2.4.27-2-686` kernel version on a Pentium 4, 3GHz, with 1GB RAM and 1MB level 2 cache. In order to avoid non deterministic operating system or hardware side-effects as much as possible, the PC was freshly rebooted in single-user mode, and the kernel used was compiled without symmetric multiprocessing support (the CPU's hyperthreading facility was turned off).

Given the inherent fuzzy and non-deterministic nature of the art of benchmarking, we are reluctant to provide precise numerical values. However such values are not really needed since the global shape of the comparative charts presented in this paper are usually more than sufficient to make some behavior perfectly clear. Nevertheless, people interested in the precise benchmarks we obtained can find the complete source code and results of our experiments at the author's website[3]. The reader should be aware that during the development phase of our experiments, several consecutive trial runs have demonstrated that timing differences of less than 10% are not significant of anything.

### 2.2 Note on array types

In the case of a 2D image represented as a 1D array in memory, we used a single index to traverse the whole image linearly. One might argue that this is unfair to the 2D representation because since the image really is a 2D one, we should use 2D coordinates (hence a double loop) and use additional arithmetics to compute the resulting 1D position of the corresponding pixel in the array. Our answer to this is no, for the following two reasons.

1. Firstly, remember that we are in the case of dedicated code, with full knowledge of data types and implementation choices. In such a case, a real world program will obviously choose the fastest solution and choose a single loop implementation for linear pixel access.
2. Secondly, remember that our main goal is to compare the performances of "equivalent" code in C and LISP. Comparing the respective merits of 1D or 2D implementation is only second-order priority at best. Hence, as long as the tested code is equivalent in both languages, our comparisons hold.

[2] `http://www.debian.org`

[3] `http://www.lrde.epita.fr/~didier/comp/research/publi.php`

### 2.3 Note on access types

It is important to consider pseudo-random access in the experiments because not all image processing algorithms work linearly. For instance, many morphological operators, like those based on "front propagation" algorithms (d'Ornellas, 2001) access pixels in an order that, while not being random, is however completely unrelated to the image's in-memory storage layout.

In order to simulate non-linear pixel access, we chose a scheme involving only arithmetic operations (the ones we were interested in benchmarking anyway): we avoided using real randomization because that would have introduced a bias in the benchmarks, and worse, it would have been impossible to compare results from C and LISP without a precise knowledge of the respective cost of the randomization function in each language or compiler.

The chosen scheme is to traverse the image by steps of a constant value, unrelated to the image geometry. In order to be sure that all pixels are accessed, the image size and the step size have to be relatively prime. 509 was chosen as the constant offset for this experiment, because it is prime relatively to 800 (our chosen image size) and is also small enough to avoid arithmetic overflow for a complete image traversal.

## 3. C Programs and Benchmarks

In this section, we establish the ground for comparison by studying the behavior and performance of specialized C implementations of the algorithms described earlier.

For benchmarking the C programs, we used the GNU C compiler, GCC [4], version 4.0.3 (Debian package version 4.0.3-1). Full optimization is obtained with the `-O3` and `-DNDEBUG` flags. To prevent the compiler from inlining the algorithm functions, the `noinline` GCC attribute (GNU C specific extension) is used. On the contrary, to force inlining of these functions, they are declared as `static inline`.

For the sake of clarity, two sample programs (details removed) are presented in listings 1 and 2: the addition algorithm for `float` images, both in linear and pseudo-random access mode.

```
void add (image *to, image *from, float val)
{
  int i;
  const int n = ima->n;

  for (i = 0; i < n; ++i)
    to->data[i] = from->data[i] + val;
}
```

**Listing 1.** Linear `float` Pixel Addition, C Version

```
void add (image *to, image *from, float val)
{
  int i, pos, offset = 0;
  const int n = ima->n;

  for (i = 0; i < n; ++i)
    {
      offset += OFFSET;
      pos = offset % n;
      to->data[pos] = from->data[pos] + val;
    }
}
```

**Listing 2.** Pseudo-Random `float` Pixel Addition, C Version

[4] `http://gcc.gnu.org`

### 3.1 Array types

Although 1D array implementation would probably be the usual choice (for questions of locality, and perhaps also genericity with respect to the image ranks), it is interesting to see how 1D or 2D array implementations behave in C, and compare this behavior with the equivalent LISP implementations.

In the 2D implementation, the image data is an array of pointers to individually `malloc`'ed lines (in sequential order), and a double line / column loop is used to access the pixels linearly.

Chart 1 shows the timings for all linear optimized algorithms with a 2D (rear) or 1D (front) array implementation.

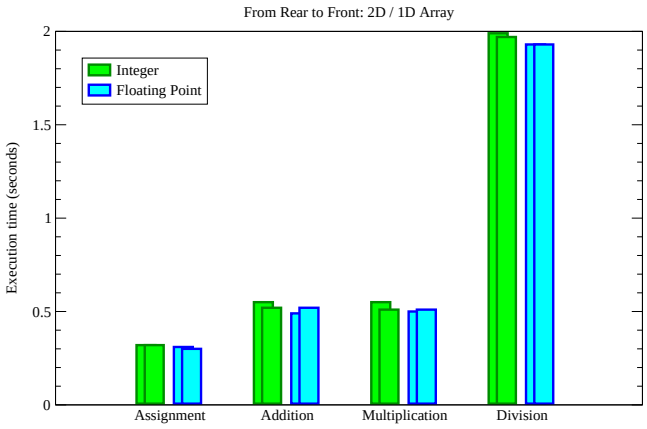


**Chart 1.** Linear Optimized Algorithms, 2D or 1D C Versions

This chart indicates that the 1D implementation ran 10% faster than the 2D one at best for integer arithmetics. This is hardly significant, and even less in the case of floating point treatment. The same experiments (not presented here) with inlined algorithms confirm this, with the case of the integer division more pronounced, as it runs 25% faster in the 1D implementation (the performance of integer division will be discussed in section 3.4 on the following page).

Chart 2 shows the timings for all pseudo-random optimized algorithms with a 2D (rear) or 1D (front) array implementation.

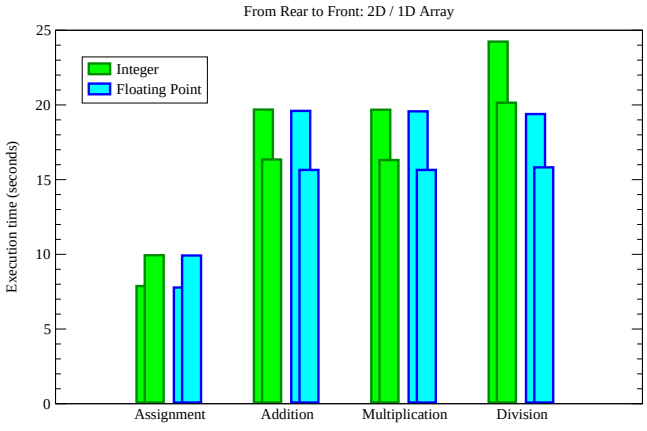


**Chart 2.** Pseudo-Random Optimized Algorithms, 2D or 1D C Versions

In that case, the differences between the two implementations are somewhat deeper: the 1D implementation almost 20% faster than the 2D one on the 3 algorithms invoking arithmetics, and this is the other way around for pixel assignment. Again, the same experiments (not presented here) with inlined algorithms confirm this.

In the remainder of this section, we will consider 1D storage layout exclusively.

### 3.2 The Assignment algorithm

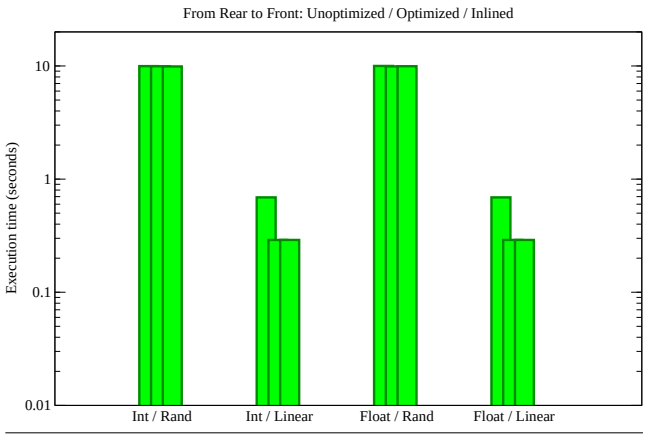


**Chart 3.** 1D Pixel Assignment, C Versions

Chart 3 presents the results obtained for the 1D assignment algorithm. Benchmarks are grouped by data type / access type combinations (this makes 4 groups). In each group, the timings (from rear to front) for the unoptimized, optimized and inlined versions are displayed. In order to make small timings more distinct, a logarithmic scale is used on the Y axis.

#### 3.2.1 Access Types

The first thing we remark on this chart is the huge difference in the performance of pseudo-random and linear access. In numbers, the linear versions run between 15 and 35 times faster than their pseudo-random counterpart, depending on the optimization level. An explanation for this behavior is provided in section 3.5 on the next page.

#### 3.2.2 Optimization levels

Chart 3 also demonstrates that while optimization is insignificant on the pseudo-random versions, the influence is non-negligible in the linear case: the optimized versions of the pixel assignment algorithm run approximately 60% faster than the unoptimized ones. To explain this, two hypotheses can be raised:

1. the huge cost of pseudo-random access completely hides the effects of optimization,
2. and / or special optimization techniques related to sequential memory addressing are at work in the linear case.

These two hypotheses have not been investigated further yet.

As real world applications *need* non linear pixel access sometimes, it is important to remember, especially during program development, that turning on optimization is much less rewarding in that case.

### 3.3 Addition and Multiplication

Charts similar to chart 3 were made for the addition and multiplication algorithms. They are very similar in shape so only the addition one is presented (see chart 4 on the following page). This chart entails 3 remarks:

- The difference in performance between pseudo-random and linear access is confirmed: the linear versions run approximately

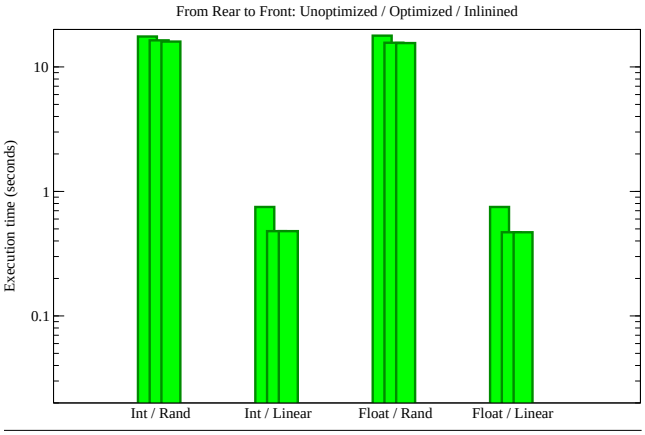


**Chart 4.** 1D Pixel Addition, C Versions

between 25 and 35 times faster than the pseudo-random access ones (15 – 35 for the assignment algorithm), depending on the optimization level. This increased performance gap in the unoptimized case is unlikely to be due to the arithmetic operations involved. A more probable cause is the fact that two images are involved, hence doubling the number of memory accesses. Again, refer to section 3.5 for an explanation.

- As in the previous algorithm (although less visible on this chart), optimization is more rewarding in the case of linear access: whereas the optimized versions of pseudo-random access algorithms run 10% faster than their unoptimized counterpart, the gain is of 36% for linear access. However, it should be noted that this ratio is inferior to that obtained with the assignment algorithm (60%) in which no arithmetics was involved (apart from the *same* looping code). This, again, suggests that optimization has more to do with sequential memory addressing than with arithmetic operations.
- Inlining the algorithms functions doesn't help much (this was also visible on chart 3 on the preceding page): the gain in performance is insignificant. It fluctuates between - and +2% which does not permit to draw any conclusion. This observation is not surprising because the cost of the function calls is negligible compared to the time needed to execute the functions themselves (*i.e.* the time needed to traverse the whole images).

### 3.4 Division

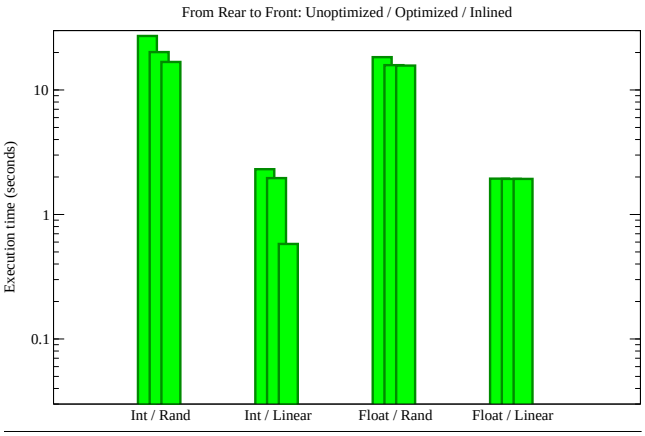


**Chart 5.** 1D Pixel Division, C Versions

The division algorithm, however, comes with a little surprise. As shown in chart 5, the division chart globally has the same shape as that of chart 4, but inlining *does* have a non-negligible impact on integer division: in pseudo-random access mode, the inlined version runs 1.2 times faster than the optimized one, and the ratio amounts to almost 3.5 in the linear case. In fact, inlining reduces the execution time to something closer to that of the multiplication algorithm.

Careful disassembly of the code (shown in listings 3 and 4) reveals that the division instruction (`idiv`) is indeed missing from the inlined version, and replaced by a multiplication (`imull`) and some other additional (cheaper) arithmetics.

```
0x08048970 <div_sb+48>:  lea    0x0(,%ecx,4),%eax
0x08048977 <div_sb+55>:  inc    %ecx
0x08048978 <div_sb+56>:  mov    (%esi,%eax,1),%edx
0x0804897b <div_sb+59>:  mov    %eax,0xfffffff8(%ebp)
0x0804897e <div_sb+62>:  mov    %edx,%eax
0x08048980 <div_sb+64>:  mov    %edx,0xfffffff4(%ebp)
0x08048983 <div_sb+67>:  cltd
0x08048984 <div_sb+68>:  idiv   %ebx
0x08048986 <div_sb+70>:  mov    0xfffffff8(%ebp),%edx
0x08048989 <div_sb+73>:  cmp    %ecx,0xfffffff0(%ebp)
0x0804898c <div_sb+76>:  mov    %eax,0xfffffff0(%ebp)
0x0804898f <div_sb+79>:  mov    %eax,(%edi,%edx,1)
0x08048992 <div_sb+82>:  jne    0x8048970 <div_sb+48>
```

**Listing 3.** Assembly Integer Division Excerpt

```
0x080489d0 <main+640>:  lea    0x0(,%esi,4),%ecx
0x080489d7 <main+647>:  mov    $0x92492493,%eax
0x080489dc <main+652>:  imull  (%ebx,%ecx,1)
0x080489df <main+655>:  inc    %esi
0x080489e0 <main+656>:  mov    (%ebx,%ecx,1),%eax
0x080489e3 <main+659>:  mov    %edx,0xffffff7c(%ebp)
0x080489e9 <main+665>:  add    %eax,%edx
0x080489eb <main+667>:  mov    (%ebx,%ecx,1),%eax
0x080489ee <main+670>:  sar    $0x2,%edx
0x080489f1 <main+673>:  sar    $0x1f,%eax
0x080489f4 <main+676>:  sub    %eax,%edx
0x080489f6 <main+678>:  cmp    %esi,0xffffffa8(%ebp)
0x080489f9 <main+681>:  mov    %edx,(%edi,%ecx,1)
0x080489fc <main+684>:  jne    0x80489d0 <main+640>
```

**Listing 4.** Assembly Inlined Integer Division Excerpt

This is an indication of a constant integer division optimization (Warren, 2002, Chap. 10). In fact, the dividend in our program is indeed a constant, and GCC is able to propagate this knowledge within the inlined version of the division function. Replacing this constant by a call to `atoi ("7")`, for instance, in the source code suffices to neutralize the optimization.

Although the case of integer division is somewhat peculiar, if not trivial, it demonstrates that inlining is always worth trying, even when the cost of the function call itself is negligible, because one never knows which next optimization such or such compiler will be able to perform.

Finally, notice that division is more costly in general: the presence of division arithmetics here, flattens the performance gap between pseudo-access and linear access. Whereas the other algorithms featured a factor between 25 and 35, linear floating-point division runs "only" 8.1 – 9.5 times faster than the pseudo-access versions.

### 3.5 Caching

As we just saw, the gain from pseudo-random to linear access is 15 – 35 for pixel assignment, addition and multiplication, and 8.1 – 9.5 for (floating point) pixel division. The arithmetic operations needed to implement pseudo-random access involve one addition

and one (integer division) remainder per pixel access, as apparent on listing 2 on page 2. These supplemental arithmetic operations alone cannot explain this performance gap.

In order to convince ourselves of this, we benchmarked alternative versions of the algorithms in which pseudo-random access related computations *are* being performed, but the image is still accessed linearly. An example of such code is shown in listing 5.

```
void assign (image *ima, float val)
{
  int i, offset = 0;
  const int n = ima->n;

  for (i = 0; i < n; ++i)
    {
      offset += OFFSET;
      ima->data[i] = offset % n;
    }
}
```

**Listing 5.** Fake Pseudo-Random `float` Pixel Assignment, C Version

Comparing the performances of these versions and the real linear ones shows that the performances degrade only by a factor 4 – 6; not 25 – 35.

What is really going on here is that pseudo-random access defeats the CPU's level 2 caching optimization. This can be demonstrated by using smaller images (that fit entirely into the cache) and a larger number of iterations in order to maintain the same amount of arithmetic operations. The initial experiments were run on $800*800$ images and 200 iterations. Such images are 2.44 times bigger than our level 2 cache. We ran the same tests on $400 * 400$ images and 800 iterations. This gives us almost the same number of arithmetic operations, but this time, it is almost possible to fit two images into the cache.

Charts 6 and 7 present all algorithms side by side, respectively for optimized linear and pseudo-random access. From rear to front, benchmarks for big and small images are superimposed. Timings are displayed on a linear scale this time.

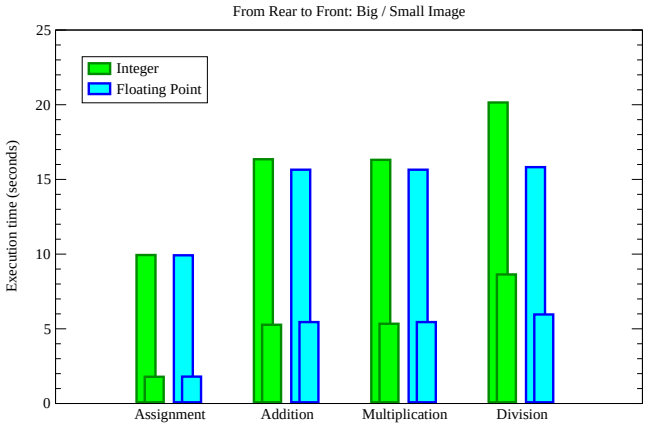


**Chart 6.** 1D Pseudo-Random Optimized Algorithms, C Versions

For pseudo-random access, the assignment algorithm goes approximately 5.5 times faster for the small image, and the other algorithms gain roughly a factor of 3, despite the same amount of arithmetics. In the linear case however, the assignment algorithm gains 2.6 (instead of 5.5), the addition and multiplication ones hardly reach 1.3 (instead of 3), and the division one gains practically nothing.

In section 2.3 on page 2, we emphasized on the importance of considering pseudo-random access in our tests. Here, we further

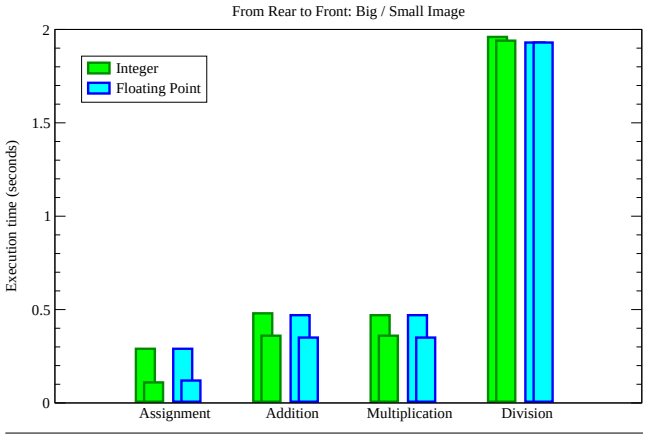


**Chart 7.** 1D Linear Optimized Algorithms, C Versions

demonstrate the importance of hardware considerations like the size of the level 2 cache: one has to realize that given the size of our images ($800 * 800$) and the chosen offset for pseudo-random access (509), we hardly jump from one image line to the next, and it already suffices to degrade the performance by a factor of 30.

This consideration is all the more important that it is not *only* a technical matter: for instance, Lesage et al. (2006) observe that for a certain class of morphological operators implemented as queue-based algorithms, there is a relation between the image entropy (for instance the number of grey levels) and the amount of cache misses.

## 4. LISP Programs and Benchmarks

In this section, we study the behavior and performance of LISP implementations of our algorithms, and establish a first batch of comparisons with the C code previously examined.

For testing LISP code, we used the experimental conditions described in section 2 on page 1, with some LISP specificities described below.

### 4.1 Experimental Conditions

**Compilers** We took the opportunity to try several LISP compilers. We tested CMU-CL [5] (version 19c), SBCL [6] (version 0.9.9) and ACL [7] (Allegro 7.0 trial version). Our desire to add LispWorks[8] to the benchmarks was foiled by the fact that the personal edition lacks the usual `load` and `eval` command line options used with the other compilers to automate the tests.

**Array Types** Both 1D and 2D array types were tested, but with a total of 7 access methods: apart from the usual `aref` method for both 1D and 2D arrays, we also experimented `row-major-aref` access for 2D arrays and `simple-vector` / `svref` storage / access for fixnums. See section 4.3 on the next page for further discussion on this.

These new parameters increased the number of tests to 252 per algorithm, making a total of 1008 individual test cases. Again, only especially interesting comparisons are shown.

Also, following the advice of Anderson and Rettig (1994), care has been taken to use `truncate` instead of `/` for fixnum division in order to avoid unwanted ratio manipulation.

[5] `http://www.cons.org/cmucl`

[6] `http://www.sbcl.org`

[7] `http://www.franz.com`

[8] `http://www.lispworks.com`

Finally, it should be noted that none of the tested programs triggered any garbage collection (GC) during timing measurement, apart from a few unoptimized (hence unimportant) cases, and a "special occasion" with ACL described later. This is a good thing because it demonstrates that for low-level operations such as the ones we are studying, the language design does not get in the way of performance, and the compilers are able to avoid costly memroy management when it is theoretically possible. See also section 7.3 on page 11 for more remarks on GC.

### 4.2 LISP code tuning

For the reader unfamiliar with LISP, it should be mentionned that requesting optimization is not achieved by passing flags to the compiler as in C, but by "annotating" the source code directly with so-called *declarations*, specifying the expected type of variables, the level of required optimization, and even compiler-specific information. Examples are given below.

Unoptimized code is obtained with `(safety 3)` and all other qualities set to 0, while fully optimized code is obtained with `(speed 3)` and all other qualities set to 0. Providing the type declarations needed by LISP compilers in order to optimize was more difficult than expected however. The process and the difficulties are explained below.

First, we compiled untyped LISP code with full optimization and examined the compilation logs. The Python compiler (MacLachlan, 1992) from CMU-CL (or SBCL for that matter) has a very nice way to throw a compilation note each time it is unable to optimize an operation. This makes it very easy to provide it with the strictly required type declarations without cluttering the code too much.

After that, we used the somewhat less ergonomic `(:explain :boxing)` declaration of ACL in order to check that these minimal type declarations were also enough for the Allegro compiler to avoid number consing. This is not however sufficient to avoid suboptimal type declaration in ACL, because even without any signalled number boxing, the compiler might be unable to open-code some of its internal functions. Thus, we had to use yet another ACL-specific declaration to get that information; namely `(:explain :calls :types)`. Some typing issues with ACL are still remaining; they will be described in section 4.3.

As a result of this typing process, two code examples are provided in listings 6 and 7. These are the LISP equivalent of the C examples shown in listings 1 and 2 on page 2.

```
(defun add (to from op)
  (declare (type (simple-array single-float (*)) to))
  (declare (type (simple-array single-float (*)) from))
  (declare (type single-float op))
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i) (+ (aref from i) op)))))
```

**Listing 6.** Linear `single-float` Pixel Addition, LISP Version

These two code samples should also be credited for illustrating one particular typing annoyance: note that adding two fixnums might produce something bigger than a fixnum, hence the risk of number boxing. This means that in principle, we should have written the addition like this:

```
(the fixnum (+ (aref from i) op))
```

or used a macro to do so. Surprisingly however, none of the tested compilers complained about this missing declaration. After some investigation, it appears that they don't need this declaration, but for different reasons:

```
(defun add (to from op)
  (declare (type (simple-array single-float (*)) to))
  (declare (type (simple-array single-float (*)) from))
  (declare (type single-float op))
  (let ((size (array-dimension to 0))
        (offset 0)
        pos)
    (declare (type fixnum offset))
    (dotimes (i size)
      (setf offset (+ offset +offset+)
            pos (rem offset size)
            (aref to pos) (+ (aref from pos) op)))))
```

**Listing 7.** Pseudo-Random `single-float` Pixel Addition, LISP Version

- Python's type inference system is relatively clever: if you `setf` the result of an addition of two fixnums to a variable also declared as a fixnum (the `to` array in our case), then it is deduced that no arithmetic overflow is expected in the code, hence it is not necessary to declare (neither to check) the type the addition's result. Note that this is also true for multiplication for instance.
- On the other hand, ACL's type inference system does not go that far, but has a "hidden" switch automatically turned on in our full optimization settings, and which *assumes* the result of fixnum addition to remain a fixnum. This is not true for multiplication however. Hence, the same code skeleton might (surprisingly) behave differently, depending on the arithmetic operation involved.

As a last example of typing difficulty, CMU-CL's type inference (or compiler note generation) system seems to have problems with `dotimes` loops: declaring the type of a `dotimes` loop index is normally not needed to get inlined arithmetics (and SBCL doesn't require it either). However, when two `dotimes` loops are nested, CMU-CL issues a note requesting an explicit declaration for the *first* index. Macro-expanding the resulting code shows that two consecutive declarations are actually issued: the one present in the `dotimes` macro, and the user-provided one. On the other hand, no declaration is requested for the second loop's index. The reason for this behavior is currently unknown (even by the CMU-CL maintainers we contacted). As a consequence, it is difficult to figure out if we can really trust the compilation notes or if there is a missed optimization opportunity: we had to disassemble the code in question in order to make sure that inlined fixnum arithmetics was indeed used.

All of this demonstrates that trying to provide the strictly necessary and correct type declarations *only* is almost impossible because compilers may behave quite differently with respect to type inference, and the typing process can lead to surprising behaviors, unless you have been "wading through many pages of technical documentation", to put it like Fischbacher (2003), and finally acquired an in-depth knowledge of each compiler specificities.

### 4.3 Array types

In LISP just as in C, one could envision to represent 2D images as 1D or 2D arrays. In section 3.1 on page 3, we showed that the chosen C representation does not make much difference for optimized code.

In LISP, the choice is actually larger than just 1 or 2D: in addition to choosing between single or multidimensional array, the following opportunities also arise:

- It is possible to treat multi-dimensional arrays as linear ones thanks to the standard function `row-major-aref`.

- At some point in LISP history (Anderson and Rettig, 1994), it was considered a better choice to use an unspecialized `simple-vector` for storing fixnums along with its accompanying accessor `svref`, because using a specialized `simple-array` might involve shifting in order to represent them as machine integers. On the other hand, fixnums are immediate objects so they are stored as-is in a `simple-vector`.

The combination of element type / array type / access method amounts to a total of 7 cases that are worth studying.

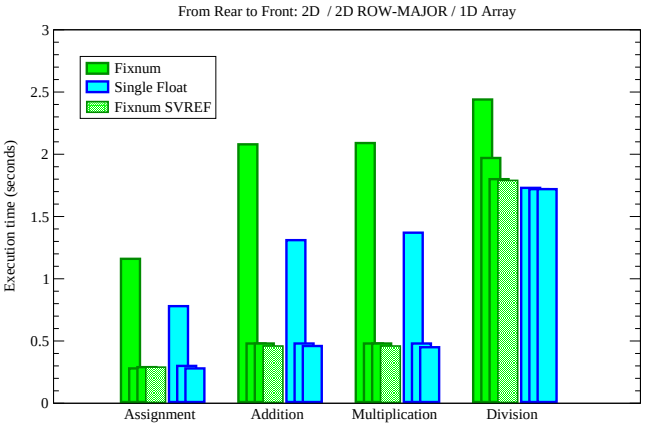


**Chart 8.** Linear Optimized Algorithms, CMU-CL Versions

Chart 8 shows the CMU-CL timings for all linear optimized algorithms, with (from rear to front), 2D, 2D `row-major` accessed, 1D, and fixnum `simple-vector` implementations. This chart entails several remarks:

- Contrary to C, using a plain 2D access method gives poor performance: the algorithms perform between 2.8 (for single-float images) and 4.5 (for fixnum images) times slower than the 1D version. This performance loss is much less important on the integer division though: the 1D fixnum version performs "only" 1.35 times faster than the 2D one, and the single-float versions perform at exactly the same speed.
- The performance gap is more pronounced for fixnum images than for single-float ones: while the performance for fixnum and single-float is equivalent in the case of 1D access, fixnum algorithms are roughly 1.5 times slower than their single-float counterpart in plain 2D access.
- Altogether, the timings for 2D `row-major` access, and 1D `aref` access methods are practically the same, although there is a small noticeable difference for integer division. This brings an interesting remark: it seems that while 2D access optimization is performed at the *compiler* level in C, it is available at the *language* level in COMMON-LISP, through the use of `row-major-aref`.
- Finally, using of a simple-vector to store fixnum images does not bring anything compared to the traditional simple-array approach. This is in contradiction to Anderson and Rettig (1994), but the reason is that nowadays, LISP compilers use efficient representations for properly specialized arrays, so for instance, fixnums are stored unshifted, just as in a simple-vector. Actually, if one wants machine integers, the `(signed-byte 32)` type is available.

A similar chart for the inlined versions (not presented here) teaches us that inlining has a bad influence on the 2D `row-major` access versions: they take about twice the time to execute, which is a strong point in favor of choosing a 1D implementation.

Similar chars (not presented here) were also made for pseudo-random access (both in optimized and inlined versions). There is no clear distinction between the different implementation choices however. The plain 2D method is behind the others in most cases (but not all), the 2D `row-major` choice would seem to be better most of the time, but the performance differences hardly reach 10% which is not enough to lead to any definitive conclusion.

The case of SBCL is very similar to that of CMU-CL, hence will not be discussed further. Let us just mention that some differences are noticed on inlined versions, which seems to indicate that inlining is still a "hot" topic in LISP compilers (probably because of its tight relationship to register allocation policies).

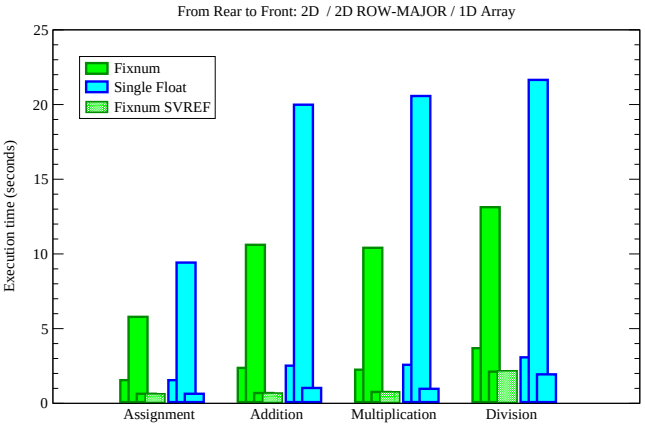


**Chart 9.** Linear Optimized Algorithms, ACL Versions

The case of ACL is peculiar enough to be presented here however (see chart 9). The most surprising behavior is the incredibly poor performance of the `row-major` access method in general, and for single-float images in particular (note that the timing goes up to 22 seconds, not even including GC time, whereas CMU-CL hardly reaches 2.5 seconds). A verbose compilation log examination leads us to the following assumptions:

- ACL indicates that it is unable to open-code the calls to `array-total-size` and, more importantly, `row-major-aref`, so it has to use a generic (polymorphic ?) one. This might explain why the performance is so poor.
- In the case of floating point images, ACL also indicates that it is generating a single-float box for every arithmetic operation. This might further explain why floating point images perform even poorer than fixnum ones.

As of now, we are still unable to explain this behavior, so it is unknown whether we missed an optimization opportunity (that would have required additional code trickery or intimate knowledge of the compiler anyway), or whether we are facing an inherent limitation in ACL.

***Final remark on plain 2D representation*** In our experiments, the size of the images are not known at compile time, so arrays are declared as follows in the 2D case:

```
(simple-array <type> (* *))
```

Out of curiosity, and motivated by the fact that ACL is unable to inline calls to `array-dimension` in this situation, we wanted to know the impact of a more precise type declaration on the performance.

To that aim, we ran experiments with hard-wired image sizes in the declarations. Arrays were hence declared as follows:

```
(simple-array <type> (800 800))
```

We observed a performance gain of approximately 30% (for CMU-CL) and 20% (for ACL) in plain 2D access (still worse than the 1D version), and no gain at all for `row-major` access. This is interesting because even when using the full expressiveness of the language (in that case, generating and compiling dedicated functions with full knowledge of the array types on the fly), we gain nothing compared to using a standardized access facility.

To conclude, the choice between 2D (even with `row-major` access because of ACL) and 1D array representation does matter a bit more than in C, but also tends towards the 1D choice. In the remainder of this section, we will consider only 1D simple array.

### 4.4 The Assignment Algorithm

Chart 10 shows comparative results for the 3 tested compilers on the optimized versions of the 1D pixel assignment algorithm. Additionally, the inlined versions are shown on the right side of each bar group for CMU-CL and SBCL (note that ACL doesn't support user function inlining). Timings are displayed on a logarithmic scale. Following are several observations worth mentioning from this chart.

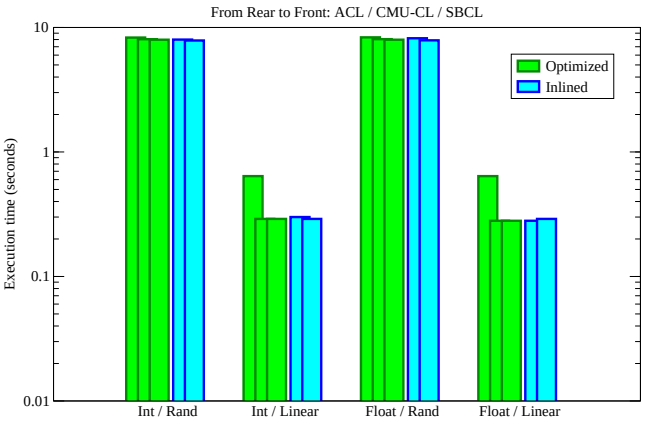


**Chart 10.** Pixel Assignment, Optimized LISP Versions

#### 4.4.1 Lisp Compilers

In pseudo-random access, all compilers are in competition, although ACL seems a bit slower. However, this is not significant enough to draw any conclusion. On the other hand, the difference in performance is significant for linear access: both CMU-CL and SBCL run twice as fast.

#### 4.4.2 Access Types

The impact of the access type on performance is exactly the same as in the C case: both CMU-CL and SBCL perform 30 times faster in linear access mode. For ACL, this factor falls down to 13 though. Further investigation on the impact of the additional arithmetics only is provided in section 4.6 on the facing page.

#### 4.4.3 Optimization levels

It would not make any sense to compare unoptimized versions of LISP code and C code (neither across LISP compilers actually) because the semantics of "unoptimized" may vary across languages and compilers: for instance, Python compilers may treat type declarations as assertions and perform type verification for unoptimized code, which never happens in C. We are not interested in benchmarking this process.

However, it can be interesting, for development process, to figure out the performance gain from non to fully optimized LISP code, and also to see its respective impact on pseudo-random and linear access, just as we did for C.

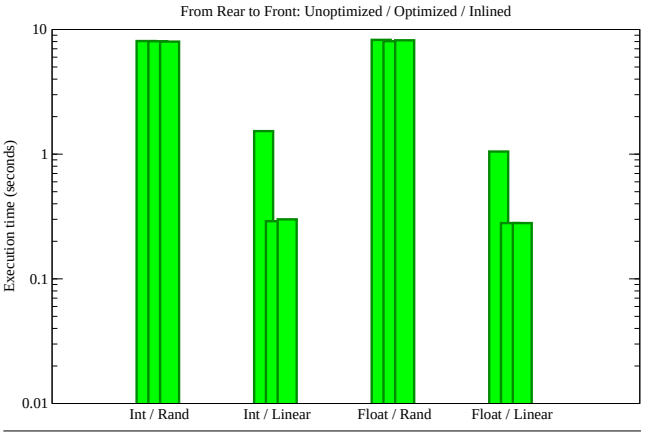


**Chart 11.** CMU-CL Pixel Assignment Performance

Chart 11 shows the performance of CMU-CL on the assignment algorithm for code fully safe (GC time removed), fully optimized, and inlined (from rear to front), in the case of 1D `aref` access. Timings are displayed on a logarithmic scale. We observe a behavior very similar to that of C (chart 3 on page 3): optimization is insignificant for pseudo-random access, but quite important for linear access. This gain is also more important here than in C: while optimized C versions ran approximately twice as fast as their unoptimized counterpart, the LISP versions run between 3.8 and 5.3 times faster. In fact, the gain is more important, not because optimized versions are faster, but because unoptimized ones are slower: in LISP, *safety* is costly.

Although not presented here, similar charts were made for SBCL and ACL. The behavior of SBCL is almost identical to that of CMU-CL, so we will say no more about it. ACL has some differences, due to the fact that the unoptimized versions are much slower (sometimes twice as slow) than Python generated code. Again, it would not be meaningful to compare unoptimized code, even between LISP compilers. But this just leads to the consequence that the effect of optimization is significant *even* in pseudo-random access mode for Allegro.

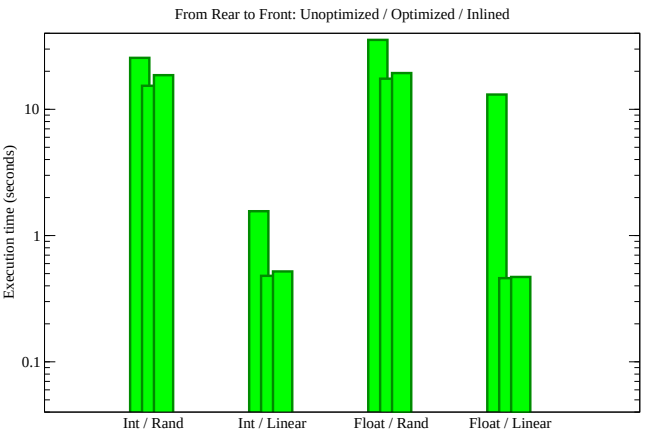


**Chart 12.** CMU-CL Pixel Addition Performance

With some anticipation on the following section, chart 12 shows the impact of optimization on the addition algorithm for CMU-CL (logarithmic scale), where we learn something new: it appears that contrary to C, the impact of optimization is much more important

on floating point calculations. More precisely, it is the cost of safety which is much greater for floating point processing than for integer calculation: optimized versions run at approximately the same speed, regardless of the pixel type. However, unoptimized fixnum assignment goes 1.4 times faster than the floating point version in the pseudo-random access case, and 8.5 times faster in linear access mode. This explains why optimization has a more important impact on the floating point versions.

Similar charts were made for the other algorithms, and the other compilers. They all confirm the preceding observations. There is one issue specific to CMU-CL however: as visible on chart 12 on the facing page, inlining actually degrades the performances, which is particularly visible on the pseudo-random access versions.

### 4.5 Other Algorithms

Performance charts were made for the addition, multiplication and division algorithms. They are very similar in shape so only the results for the division one is presented on chart 13 (the division was chosen because it has some specificities). As in chart 10 on the facing page, the performance of the 3 tested compilers is presented for the optimized versions, along with the inlined versions of CMU-CL and SBCL.

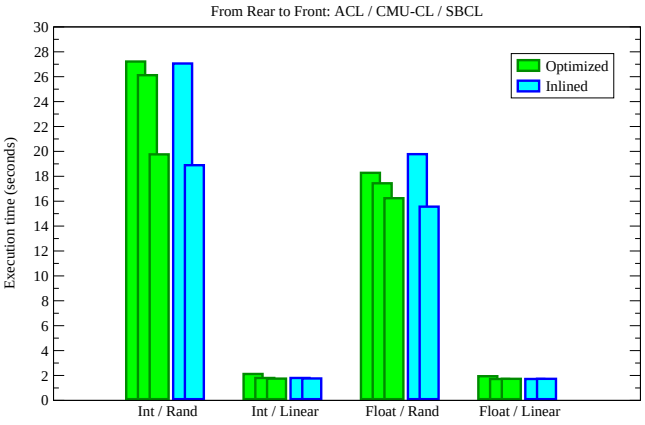


**Chart 13.** Pixel Division, Optimized LISP Versions

The following points can be noticed on this chart:

- Globally, we can classify ACL, CMU-CL and finally SBCL by order of efficiency (from the slowest to the fastest). This is also the case for the other algorithms.
- Specifically to pseudo-random integer division, SBCL seems quite better than the two others. The reason for this particularity is currently unknown.
- The impact of the access type on performance here is exactly the same as in the C case: the division, being more costly reduces the gap between linear and pseudo-random access to a factor of 9 – 11 (the other algorithms feature a factor of 30, just like in C).
- This chart also confirms that inlining is insignificant at best. However, CMU-CL seems to have a specific problem with pseudo-random access (this is true for all algorithms): inlining degrades the performances by a factor up to 15% in some cases. Although not presented here, we found at that SBCL suffers from inlining too, but mostly for linear access. All of this suggests that inlining is still a "hot" topic in LISP compilers, and can have a serious impact on locality and on register allocation policies.
- Finally, the global insignificance of inlining in this very case of integer division demonstrates that none of the tested compilers seem to be able to perform the constant integer division optimization brought to light with C. This hypothesis was confirmed by compiling and disassembling a short function of one argument performing a constant division of this argument: for both CMU-CL and SBCL, the division is optimized *only* when the dividend is a power of 2, whereas GCC is able to optimize *any* constant dividend.

### 4.6 Caching

As in the C case, we noticed a factor of 30 in performance between linear a pseudo-random access. In order to evaluate the impact of the additional arithmetic operations only, we ran tests equivalent to those described in section 3.5 on page 4 (linear access, but still performing pseudo-random access related arithmetics). Listing 8 shows the alternative assignment algorithm for single-float images used in this aim.

```
(defun assign (image value)
  (declare (type (simple-array fixnum (*)) image))
  (declare (type fixnum value))
  (let ((size (array-dimension image 0))
        (offset 0))
    (declare (type fixnum offset))
    (dotimes (i size)
      (setf offset (+ offset +offset+)
            (aref image i) (rem offset size)))))
```

**Listing 8.** Alternative `single-float` Pixel Assignment, LISP Version

Here again, comparing the performances of these versions and the real linear ones shows that the performances degrade only by approximately a factor of 5 (just like in C); not 30. Still as in the C case, comparing performances on big and small images with the same amount of arithmetic operations leads to charts 14 and 15 on the next page for CMU-CL.

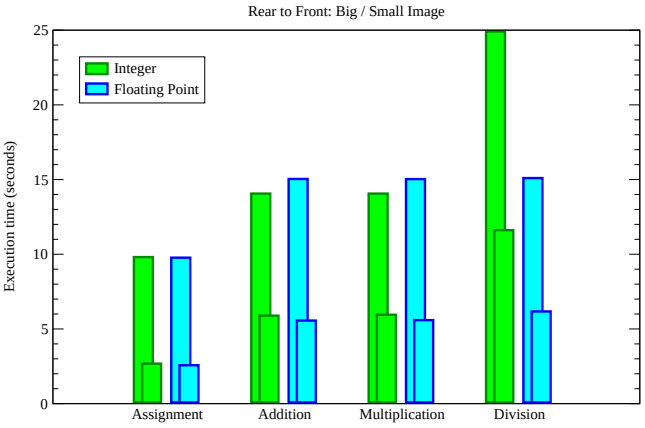


**Chart 14.** Pseudo-Random Optimized Algorithms, CMU-CL Versions

These charts are very similar to charts 6 on page 5 and 7 on page 5 for C: for pseudo-random access, the assignment algorithm goes 3.8 times faster for the small image (5.5 for C), and the other algorithms gain roughly a factor of 2.3 (3 for C). In the linear case, the assignment algorithm gains roughly a factor of 3 (2.6 for C) instead of 3.8, the addition and multiplication algorithms hardly reach 1.2 (1.3 n the case of C), and the division algorithm gains nothing, exactly as in C.

Although not presented here, similar charts were made for SBCL and, as expected, they exhibit the same behavior as CMU-CL. The case of ACL is also very similar, although the gain in performance

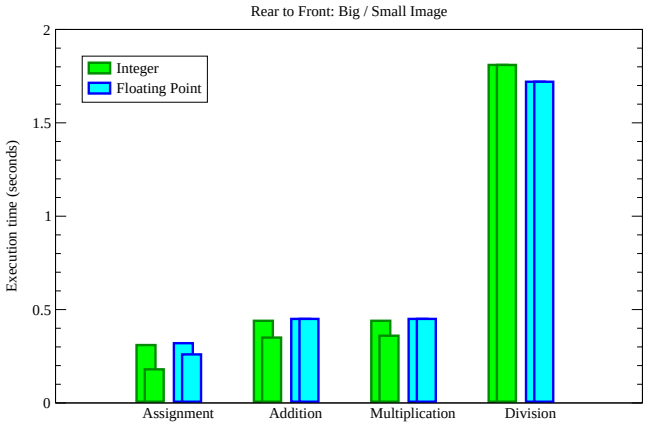


**Chart 15.** Linear Optimized Algorithms, CMU-CL Versions

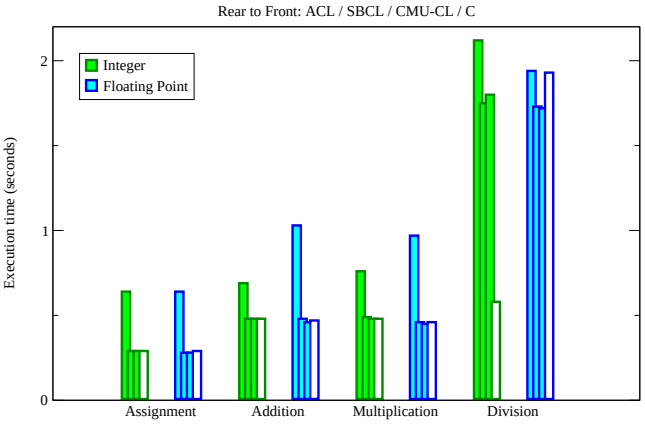


**Chart 17.** Linear Algorithms, Fastest Versions

from big to small image have a tendency to be globally smaller than for CMU-CL and SBCL.: in pseudo-random access, the assignment algorithm runs less than twice as fast on the small image (compare this to 3.8 in the case of CMU-CL) and the ratio for the other algorithms is rather of 2.2, almost identical to the case of CMU-CL. In linear access, the gain is practically nothing however, even for the assignment algorithm which is surprising since we got a factor of 3 with Python generated code.

## 5. Final Comparative Performance

In this section, we draw a final comparison of absolute performance between the C and LISP versions of our algorithms.

Charts 16 and 17 show all the algorithms, respectively in pseudo-random and linear access mode. From rear to front, execution times for ACL, SBCL, CMU-CL and C are presented.

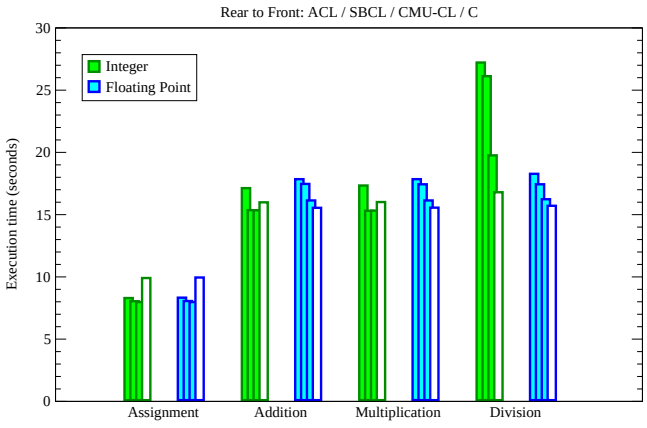


**Chart 16.** Pseudo-Random Algorithms, Fastest Versions

Note that these charts compare what we found to be globally the best implementation solution for performance in every case. This means that we use a 1D representation in both camps, and we actually compare inlined C versions with optimized (but not inlined) LISP versions because inlining makes things worse in LISP most of the time. Hence, strictly speaking, we are not comparing exactly the same things.

This is justified by the position we are adopting: that of a programmer interested in getting the best in each language while not being an expert in low level optimization. Only general optimization flags (`-O` in C, standard qualities declarations in LISP) and inlining are used. No compiler-specific local trick has been tested, either in C or in LISP.

And here comes an enjoyable little surprise:

- For pseudo-random access, the assignment algorithm runs about 19% faster in LISP than in C. For pixel addition and multiplication, the performance gap hardly reaches 5%, and is sometimes in favor of LISP, sometimes in favor of C. So the differences are completely insignificant.
- The only exception to that is the case of integer division where C is significantly faster, but notice however that even without knowledge of the constant integer division optimization, SBCL manages to lose only by 17%.

The linear case also teaches us several things:

- ACL is noticeably slower than both CMU-CL and SBCL: it runs between 1.4 and 2.2 times slower than its competitors. The case of division is less pronounced though.
- C code gives performances *strictly* equivalent to that of CMU-CL or SBCL, with the exception of division.
- In that case, it clearly wins with integers (because of the optimization already mentioned), but lose by 10% with floating point images.

Similar charts were made for small images, fitting entirely into the cache, and are of similar shape, so they are not presented here. The only different behavior they exhibit is that of ACL for which the performances are even poorer, especially in the linear case. This suggests that apart maybe from ACL, low-level hardware optimization has globally the same impact on LISP and C code. In other words, the aspects of locality (both of instructions and data) are topologically very close.

## 6. Conclusion

In this paper, we described an ongoing research on the behavior and performance of LISP in the context of scientific numerical computing. We tested a few simple image processing algorithms both in C and LISP, and also took the opportunity to examine three different LISP compilers. With this study, we demonstrated the following points:

- With respect to parameters external to the languages, such as hardware optimization via caching, the behavior of equivalent LISP and C code is globally the same. This is comforting for people considering switching language, because they should

not be expecting any unpredictable behavior from anything else than language transition.

- With respect to parameters internal to the languages, we saw that there can be surprising differences in behavior between C and LISP. For instance, we showed that plain 2D array representations behave quite differently (slow in LISP), and that performance can depend on the data type in LISP (which does not happen in C). However, as soon as optimization is at work, and is the main goal, those differences vanish: we are led to choose equivalent data structures (namely 1D (simple) arrays), and regain homogeneity on the treatment of different data types. This is also comforting for considering language transition.
- Since Fateman et al. (1995); Reid (1996), considerable work has been done in the LISP compilers in the fields of efficient numerical computation and data structures, to the point that equivalent LISP and C code entails *strictly* equivalent performance, or even *better* performance in LISP sometimes. This should finally help getting C or C++ people's attention.
- We should also emphasize on the fact that these performance results are obtained without intimate knowledge of either the languages or the compilers: only standard and / or portable optimization settings were used.

To be fair, we should mention that with such simple algorithms as the ones we tested, we are comparing *compilers* performance as well as *languages* performance, but compiler performance is a critical part of the production chain anyway. Moreover, to be even fairer, we should also mention that when we speak of "equivalent C and LISP" code, this is actually quite inaccurate. For instance, we are comparing a *language construct* (`for`) with a *programmer macro* (`dotimes`); we are comparing sealed function calls in C with calls to functions that may be dynamically redefined in LISP *etc.*. This means that it is actually impossible to compare exclusively either language, or compiler performance. This also means that given the inherent expressiveness of LISP, compilers have to be even smarter to reach the efficiency level of C, and this is really good news for the LISP community.

LISP still has some weaknesses though. As we saw in section 4.2 on page 6 it is not completely trivial to type LISP code both correctly *and* minimally (without cluttering the code), and *a fortiori* portably. Compilers may behave very differently with respect to type declarations, may provide type inference systems of various quality, and who knows which more or less advertised optimization flag of their own (this happens also in C however). The yet-to-be-explained problem we faced with ACL's `row-major` access is the perfect example. Maybe the COMMON-LISP standard leaves too much freedom to the compilers in this area.

Another current weakness of LISP compilers seems to be inlining technology. ACL simply does not support user function inlining yet. The poor performance of CMU-CL in this area seems to indicate that inlining defeats its register allocation strategy. It is also regrettable that none of the tested compilers are aware of the integer constant division optimization brought to light with C, and from which inlining makes a big difference.

## 7. Perspectives

The perspectives of this work are numerous. We would like to emphasize on the ones we think are the most important.

### 7.1 Further Investigation

The research described in this paper is still ongoing. In particular, we have outlined several currently unexplained behaviors with respect to typing, performance, or optimization of such or such LISP compiler. These oddities should be further analyzed, as they probably would reveal room for improvement in the compilers.

### 7.2 Vertical Complexity

Benchmarking these very simple algorithms was necessary to isolate the parameters we wanted to test (namely pixel access and arithmetic operations). These algorithms can actually be considered as the kernel of a special class of image processing ones called "point-wise algorithms", involving pixels one by one. It is likely that our study will remain relevant for all algorithms in this class, but nevertheless, the same experiments should be conducted on more complex point-wise treatments, involving more local variables, function calls *etc.* for instance in order to spot potential weaknesses in the register allocation policies of LISP compilers, as the inlining problem tends to suggest, or in order to evaluate the cost of function calls.

This is also the place where it would be interesting to measure the impact of compiler-specific optimization capabilities, including architecture-aware ones like the presence of SIMD (Single Instruction, Multiple Data) instruction sets. One should note however that this leads to comparing compilers more than languages, and that the performance gain from such optimizations would be very dependant on the algorithms under experimentation (thus, it would be difficult to draw a general conclusion).

### 7.3 Horizontal complexity

Besides point-wise algorithms, there are two other important algorithm classes in image processing: algorithms working with "neighborhoods" (requiring several pixels at once) and algorithms like front-propagation ones where treatments may begin at several random places in the image. Because of their very different nature in pixel access patterns, it is very likely that their behavior would be quite different from what we discussed in this paper, so it would be interesting to examine them too.

Also, because of their inherent complexity, this is where the differences in language expressiveness would be taken into account, and in particular where it would become important to include GC timings in the comparative tests, because it is part of the language design (not to mention the fact that different compilers use different GC techniques which adds even more parameters to compare).

### 7.4 Benchmarking other compilers / architectures

The benchmarks presented in this paper were obtained on a specific platform, with specific compilers. It would be interesting to measure the behavior and performance of the same code on other platforms, and also with other compilers. The automated benchmarking infrastructure provided in the source code should make this process easier for people willing to do so.

### 7.5 From dedication to genericity

Benchmarking low-level, fully dedicated code was the first step in order to get C or C++ people's attention. However, most image treaters want some degree of genericity: genericity on the image types (RGB, Gray level *etc.*), image representation (integers, unsigned, floats, 16bits, 32bits *etc.*), and why not on the algorithms themselves. To this aim, the object oriented approach is a natural way to go. Image processing libraries with a variable degree of genericity exist both in C++ and in LISP, using CLOS (Keene, 1989) . As a next step of our research, a study on the cost of dynamic genericity in the same context as that of this paper is at work already. However, given the great flexibility and expressiveness of CLOS that C++ people might not even feel the need for (classes can be redefined on the fly, the dispatch mechanism is customizable *etc.*), the results are not expected to be in favor of LISP this time.

### 7.6 From dynamic to static genericity

Even in the C++ community, some people feel that the cost of dynamic genericity is too high. Provided with enough expertise on the template system and on meta-programming, it is now possible to write image processing algorithms in an object oriented fashion, but in such a way that all generic dispatches are resolved at compile time (Burrus et al., 2003). Reaching this level of expressiveness in C++ is a very costly task, however, because template programming is cumbersome (awkward syntax, obfuscated compiler error messages *etc.*). There, the situation is expected to turn dramatically in favor of LISP. Indeed, given the power of the LISP macro system (the whole language is available in macros), the ability to generate function code *and* compile it on-the-fly, we should be able to automatically produce dedicated hence optimal code in a much easier way. There, the level of expressiveness of each language becomes of a capital importance.

Finally, it should be noted that one important aspect of static generic programming is that the cost of abstraction resides in the compilation process; not in the execution anymore. In other words, it will be interesting to compare the performances of LISP and C++ not in terms of execution times, but compilation times.

### 7.7 From Image Processing to ...

In this paper we put ourselves in the context of image processing to provide a concrete background to the tests. However, one should note that the performance results we got do not reduce to this application domain in particular. Any kind of application which involves numerical processing on large sets of contiguous data might be interested to know that LISP has caught up with performance.

## Acknowledgments

The author would like to thank Jérôme Darbon, Nicolas Pierron, Sylvain Peyronnet, Thierry Géraud and many people on `comp.lang.lisp` and some compiler-specific mailing lists for their help or insight.

## References

(2002). *Horus User Guide*. University of Amsterdam, The Netherlands.

Anderson, K. R. and Rettig, D. (1994). Performing LISP: Analysis of the fannkuch benchmark. *ACM* SIGPLAN LISP *Pointers*, VII(4):2–12. Downloadable version at `http://www.apl.jhu.edu/~hall/text/Papers/Lisp-Benchmarking-and-Fannkuch.ps`.

Boreczky, J. and Rowe, L. A. (1994). Building COMMON-LISP applications with reasonable performance. `http://bmrc.berkeley.edu/research/publications/1993/125/Lisp.html`.

Burrus, N., Duret-Lutz, A., Géraud, T., Lesage, D., and Poss, R. (2003). A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL)*, Anaheim, CA, USA.

d'Ornellas, M. (2001). *Algorithmic Pattern for Morphological Image Processing*. PhD thesis, University of Amsterdam.

Duret-Lutz, A. (2000). Olena: a component-based platform for image processing, mixing generic, generative and OO, programming. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—-Young Researchers Workshop; published in "Net.ObjectDays2000"*, pages 653–659, Erfurt, Germany. `http://olena.lrde.epita.fr`.

Fateman, R. J., Broughan, K. A., Willcock, D. K., and Rettig, D. (1995). Fast floating-point processing in COMMON-LISP. *ACM Transactions on Mathematical Software*, 21(1):26–62. Downloadable version at `http://openmap.bbn.com/~kanderso/performance/postscript/lispfloat.ps`.

Fischbacher, T. (2003). Making LISP fast. `http://www.cip.physik.uni-muenchen.de/~tf/lambda/aei/lisp.html`.

Froment, J. (2000). *MegaWave2 System Library*. CMLA, École Normale Supérieure de Cachan, Cachan, France. `http://www.cmla.ens-cachan.fr/Cmla/Megawave`.

Gabriel, R. P. (1985). *Performance and Evaluation of* LISP *Systems*. MIT Press.

Ibanez, L., Schroeder, W., Ng, L., and Cates, J. (2003). *The ITK Software Guide: The Insight Segmentation and Registration Toolkit*. Kitware Inc. `http://www.itk.org`.

Keene, S. E. (1989). *Object-Oriented Programming in* COMMON-LISP*: a Programmer's Guide to* CLOS. Addison-Wesley.

Lesage, D., Darbon, J., and Akgul, C. B. (2006). An efficient algorithm for connected attribute thinnings and thickenings. In *Proceedings of the International Conference on Pattern Recognition*, Hong-Kong, China.

MacLachlan, R. A. (1992). The python compiler for CMU-CL. In *ACM Conference on* LISP *and Functional Programming*, pages 235–246. Downloadable version at `http://www-2.cs.cmu.edu/~ram/pub/lfp.ps`.

Neuss, N. (2003). On using COMMON-LISP for scientific computing. In *CISC Conference, LNCSE*. Springer-Verlag. Downloadable version at `http://www.iwr.uni-heidelberg.de/groups/techsim/people/neuss/publications.html`.

Quam, L. H. (2005). Performance beyond expectations. In of LISP Users, T. A., editor, *International* LISP *Conference*, pages 305–315, Stanford University, Stanford, CA. The Association of LISP Users. Downloadable version at `http://www.ai.sri.com/~quam/Public/papers/ILC2005/`.

Reid, J. (1996). Remark on "fast floating-point processing in COMMON-LISP". In *ACM Transactions on Mathematical Software*, volume 22, pages 496–497. ACM Press.

Steele, G. L. (1990). COMMON-LISP *the Language, 2nd edition*. Digital Press. Online and downloadable version at `http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`.

Warren, H. S. (2002). *Hacker's Delight*. Addison Wesley Professional. `http://www.hackersdelight.org`.