Beating C in Scientific Computing Applications

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

Version 1.3 – July 1, 2006
Facts:

- "LISP is slow"... NOT! (it’s been 20 years)
- Image processing libraries written in C or C++ (sacrificing expressiveness for performance)
- LISP achieving 60% speed of C (recent studies)

We have to do better:

- Studying behavior and performance of LISP (part 1: full dedication)
- 4 simple image processing algorithms
- Pixel storage and access / arithmetic operations

Equivalent performance
(LISP 10% better in some cases)
# Table of Contents

1. Experimental Conditions
2. C Programs and Benchmarks
3. LISP Programs and Benchmarks
4. Type inference
Experimental conditions

- **The algorithms**: the “point-wise” class
  - Pixel assignment / addition / multiplication / division
  - Soft parameters: image size / type / storage / access
  - Hard parameters: compilers / optimization level
  - ⇒ More than 1000 individual test cases

- **The protocol**
  - Debian GNU Linux / 2.4.27-2-686 packaged kernel
  - Pentium 4 / 3GHz / 1GB RAM / 1MB level 2 cache
  - Single user mode / SMP off (no hyperthreading)
  - Measures on 200 consecutive iterations
The \textit{add} function

```c
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;
}
```

- \textbf{Gcc 4.0.3 (Debian package)}
- \textbf{Full optimization: }\texttt{-O3 \textendash{}DNDEBUG} plus inlining
- \textit{Note}: inlining should be almost negligible
Results
In terms of behavior

- 1D implementation *slightly better* (10% ⇒ 20%)
- Linear access faster (15 ⇒ 35 times)
  - Arithmetic overhead: only 4x – 6x
  - Main cause: hardware cache optimization
- Optimized code faster (60%) in linear case, irrelevant in pseudo-random access
  - Causes currently unknown
- Inlining negligible (2%)
Results
In terms of performance

Fully optimized inlined C code

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Integer Image</th>
<th>Float Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>0.29</td>
<td>0.29</td>
</tr>
<tr>
<td>Addition</td>
<td>0.48</td>
<td>0.47</td>
</tr>
<tr>
<td>Multiplication</td>
<td>0.48</td>
<td>0.46</td>
</tr>
<tr>
<td>Division</td>
<td>0.58</td>
<td>1.93</td>
</tr>
</tbody>
</table>

- Not much difference between pixel types
- **Surprise:** integer division should be costly
  - “Constant Integer Optimization” (with inlining)
  - **Do not neglect inlining!**
The **add function**

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

- **CMU-CL** (19c), **SBCL** (0.9.9), **ACL** (7.0)
- **Full optimization**: `(speed 3), 0 elsewhere`
- **Array type**: 1D, 2D
- **Array access**: `aref`, `row-major-aref`, `svref`
Comparative results
In terms of behavior

- Plain 2D implementation much slower (2.8x $\Rightarrow$ 4.5x)
- Linear access faster (30 times)
  - Same reasons, same behavior...
- Optimized code faster in linear case, irrelevant in pseudo-random access
  - Gain more important in LISP (3x $\Rightarrow$ 5x)
  - Gain more important on floating point numbers
  - In LISP, safety is costly
- Inlining negligible
  - No “Constant Integer Optimization”
  - Negative impact on performance (-15%), if any
  - Inlining still a “hot” topic (register allocation policies?)
Comparative results
In terms of performance

Pseudo-random access

- Assignment: LISP 19% faster than C
- Other: insignificant (5%)
- Exception: integer division
Comparative results
In terms of performance

- **ACL**: poor performance
- **CMU-CL, SBCL**: strictly equivalent to C
- **C**: wins on integer division, loses on floating-point one
Type inference
A weakness of COMMON-LISP ...

- **Static typing cumbersome** (source code annotations)
  - Can we provide *minimal* type declarations ...
  - ... and rely on type inference?

- **Incremental typing** by compilation log examination

- **Unfortunately:**
  - Compiler messages not necessarily ergonomic
  - Type inference systems not necessarily clever
Example of (missing) type inference

\[
\begin{align*}
&\text{multiply excerpt} \\
&;; \ldots \\
&(\text{declare (type (simple-array fixnum (*)) to from)}) \\
&(\text{declare (type fixnum val)}) \\
&;; \ldots \\
&(\text{setf (aref to i) (the fixnum (* (aref from i) val))})
\end{align*}
\]

- \((* \text{ fixnum fixnum}) \neq \text{fixnum in general, but...}\)
  - to declared as an array of \text{fixnum}'s,
  - so the multiplication \text{has to return a fixnum}
- \text{CMU-CL and SBCL ok, ACL not ok.}
  - Need for further explicit type information
  - \text{worse in ACL:}
    \text{declared-fixnums-remain-fixnums-switch}
Conclusion

- **In terms of behavior**
  - External parameters: no surprise
  - Internal parameters: differences, attenuated by optimization

- **In terms of performance**
  - Comparable results in both languages
  - Very smart LISP compilers (given language expressiveness)

- **However:**
  - Typing can be cumbersome
  - Difficult to provide both correct and minimal information (weakness of the COMMON-LISP standard)
  - Inlining is still an issue
Perspectives

- **Low level**: try other compilers / architectures (and compiler / architecture specific optimization settings)

- **Medium level**: try more sophisticated algorithms (neighborhoods, front-propagation)

- **High level**: try different levels of genericity (dynamic object orientation, static meta-programming)

- Do not restrict to image processing
Questions?