r/lisp • 2 yr. ago
cuntymccuntlicker

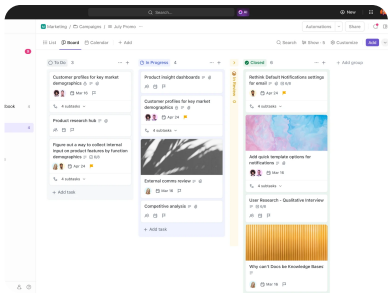
How fast can you multiply matrices using only common lisp?

Just as a matter of curiosity, I am trying to see how fast common lisp on its own can go by writing some matrix-multiplication routines. The obvious

```
(defun matrix-multiply (a b)
  "Performs matrix multiplication of two arrays."
  (assert (and (= (array-rank a) (array-rank b) 2)
               (= (array-dimension a 1) (array-dimension b 0)))
          (a b)
          "Cannot multiply ~S by ~S." a b)
```

Read more

12 27 Share



Add a Comment

Sort by: Best Search Comments

n16f • 2y ago • Edited 2y ago •

Using type declarations, e.g. asserting that elements are `SINGLE-FLOAT` or `DOUBLE-FLOAT` should massively improve performances. See the CLHS for more information [1].

Beyond that, SBCL comes with SB-SIMD [2] which, like the name says, allows you to use SIMD

[1] http://www.lispworks.com/documentation/HyperSpec/Body/d_type.htm

[2] http://sbcl.org/manual/index.html#sb_002dsimd

16 Reply

ventuspilot • 2y ago •

I tried your original `matrix-multiply` and on my laptop I ended up with

```
Evaluation took:
  50.864 seconds of real time
  50.984375 seconds of total run time (50.546875 user, 0.437500 system)
  [ Run times consist of 1.531 seconds GC time, and 49.454 seconds non-GC time. ]
  100.24% CPU
  123,051,720,362 processor cycles
  80,023,991,856 bytes consed
```

Here's what I did to make your `matrix-multiply` go brrr:

- put the function in a file
- begin the file with `(declaim (optimize (speed 3)))`
- compile it with `sbcl --eval "(compile-file \"mmult.lisp\")" --quit`

This gave a ton of compiler notes about missing optimization opportunities because of missing type information. Then I added type declarations until all the compiler notes were gone, and this was the result:

```
(declaim (optimize (speed 3)))

(defun matrix-multiply (a b)
  "Performs matrix multiplication of two arrays."
  (declare (type (simple-array double-float 2) a b))
  (assert (and (= (array-rank a) (array-rank b) 2)
               (= (array-dimension a 1) (array-dimension b 0)))
          (a b)
          "Cannot multiply ~S by ~S." a b)
  (let* ((m (first (array-dimensions a)))
         (n (second (array-dimensions b)))
         (p (second (array-dimensions a)))
         (result (make-array `(,m ,n) :element-type (array-element-type a)
                                      :adjustable nil :fill-pointer nil)))
    (declare (fixnum m n p))
```

```
        (declare (fixnum j))
        (dotimes (k p)
          (declare (fixnum k))
          (incf (aref result i j) (* (aref a i k) (aref b k j))))))
    result))

(time (matrix-multiply (make-array '(1000 1000) :element-type 'double-float
                                   :initial-element 1.0d0 :adjustable nil :fill-poin
                       (make-array '(1000 1000) :element-type 'double-float
                                   :initial-element 2.0d0 :adjustable nil :fill-poin


Evaluation took:
  4.022 seconds of real time
  4.015625 seconds of total run time (4.015625 user, 0.000000 system)
  99.85% CPU
  9,731,819,889 processor cycles
  24,000,048 bytes consed
```

I.e. a 10x speedup, also note how only the three arrays need to be allocated.

Changing `double-float` to `single-float` made it slightly faster:

```
Evaluation took:
  3.252 seconds of real time
  3.250000 seconds of total run time (3.250000 user, 0.000000 system)
  99.94% CPU
  7,867,932,005 processor cycles
  12,000,048 bytes consed
```

If you want to go faster that that maybe look into SB-SIMD or use a library as others suggested. My main point was that sbcl's compiler notes when using compile-file are pretty detailed and helpful for finding missing type declarations.

7 Reply

m518xt • 2y ago •

Maybe have a look at how magicl does this?

I did a similar exploration for a dot product recently based on conversations with u/stylewarning in r/Common_Lisp. I found SBCL was faster than numpy.

I then dug into the various elements (for my own understanding).

stylewarning • 2y ago •

These are the type combinations that are currently optimized in pure and portable Common Lisp (without BLAS): https://github.com/quil-lang/magicl/blob/master/src/high-level/matrix-functions/mult-methods.lisp#L7

5 Reply

cuntymccuntlicker OP • 2y ago •

Is it me or numcl is faster than magicl? Matrix multiplication on magicl with pure lisp backend is

```
Evaluation took:
  6.414 seconds of real time
  6.413099 seconds of total run time (6.413099 user, 0.000000 system)
  99.98% CPU
  14,775,994,330 processor cycles
  4,000,016 bytes consed
```

while with numcl I get

```
Evaluation took:
  0.423 seconds of real time
  0.422378 seconds of total run time (0.422378 user, 0.000000 system)
  99.76% CPU
  973,162,966 processor cycles
  4,000,016 bytes consed
```

For reference, numpy takes 0.0133 seconds. When I run htop I can see that numcl is only using a single core though, so maybe sb-simd can be used? Perhaps something to look into for your blog!

3 Reply

6 more replies

neil-lindquist • 2y ago •

Besides the lisp-specific optimizations mentioned elsewhere in this thread, you're missing major algorithmic optimizations to your matrix multiply.

1. You need cache-blocking. So, instead of 3 loops, you'll have 5 with the outer 2 stepping over blocks of the matrix.
2. You'll want to test different loop orders. For example, in a 3-loop mat-mul, the k-i-j ordering

3. For the second case, don't explicitly compute the transpose. Instead just access B as `(aref b j k)`. This reduces data movement, is better for SIMD, and uses the cache lines better.
4. For the multithreaded version, you maybe want to use a 2d distribution of threads. In your version, each thread accesses 1/8 of C and A but all of B (5/12 of the data). With a 4x2 distribution, each thread would only access 1/4 of C and A and half of B (4/12 of the data).

It's worth noting that the inner-kernel of numpy's GEMM is hand-coded assembly (probably written by people with Ph.D.'s in optimizing linear algebra kernels.)

3 Reply

2 more replies

sickofthisshit • 2y ago •

Your "Common Lisp on its own" is not really a useful exercise, IMO. For this kind of thing, you are going to be testing what machine code your compiler emits, and that is not part of the definition of Common Lisp, but an artifact of the implementation and whatever declarations or other hints you give the implementation to emit efficient code.

At some point, you are going to want to convince your compiler to emit some special instructions, such as SIMD operations, hopefully in cache-aware fashion, which is strongly CPU dependent. Common Lisp doesn't specify such things, so you will either have to figure out your implementation-specific magic or use a library where the library author has figured it out.

5 Reply

mm007emko • 2y ago •

This is a school-like excersise. As others said, matrix multiplications today, especially in machine learning code, has to use vector instructions (SIMD). The vast majority of CPUs nowadays support them. See either SB-SIMD if you have SBCL or use BLAS library. There are wrappers around scientific libraries available for CL if you don't want to do it on your own.

2 Reply

zyni-moe • 2y ago •

You need at least type declarations. Here is (probably incorrect, beware) version which handles arrays of good types specially, and you can add more good types if you wish.

```
(deftype array-index ()
  `(integer 0 (,array-dimension-limit)))

(defvar *matrix-multipliers* (make-hash-table :test #'equal))
```

```
            (lambda (a b c ar ac bc zero-c)
              ,@(if for-type
                    `((declare (type (simple-array ,for-type 2) a b c)
                               (optimize speed)))
                    '())
              (declare (type array-index ar ac bc))
              (when zero-c
                (let ((s (array-total-size c)))
                  (declare (type array-index s))
                  (dotimes (i s)
                    (setf (row-major-aref c i) ,(if for-type (coerce 0 for-type) 0)))
              (dotimes (i ar c)
                (declare (type array-index i))
                (dotimes (j ac)
                  (declare (type array-index j))
                  (dotimes (k bc)
                    (declare (type array-index k))
                    (incf (aref c i j) (* (aref a i k) (aref b k j))))))))
     ',for-type))

(define-matrix-multiplier)
(define-matrix-multiplier single-float)
(define-matrix-multiplier double-float)

(defun matrix-multiplier-for (type)
  (or (gethash type *matrix-multipliers*)
      (gethash nil *matrix-multipliers*)))

(defun matrix-multiply (a b &optional (c nil))
  (declare (type (array * 2) a b)
           (type (or (array * 2) null) c))
  (let ((ar (array-dimension a 0))
        (ac (array-dimension a 1))
        (br (array-dimension b 0))
        (bc (array-dimension b 1))
        (at (array-element-type a))
        (bt (array-element-type b)))
    (declare (type array-index ar ac bc))
    (unless (= ac br)
      (error "dimension"))
      (if (and (typep a 'simple-array)
               (typep b 'simple-array)
               (typep c '(or null simple-array))
               (eq at bt)
               (or (not c) (eq (array-element-type c) at)))
```

```
                    ar ac bc
                    t)
            (funcall (matrix-multiplier-for at)
                     a b (make-array (list bc ar)
                                     :element-type at
                                     :initial-element (coerce 0 at))
                     ar ac bc
                     nil))
        (if c
            (funcall (matrix-multiplier-for nil)
                     a b c
                     ar ac bc
                     t)
            (funcall (matrix-multiplier-for nil)
                     a b (make-array (list bc ar) :initial-element 0)
                     ar ac bc
                     nil)))))
```

With SBCL on M1 this will multiply two 1000x1000 double-float arrays in 3.5 seconds. Obviously you can do much better than this, but without type declarations all is lost.

2 Reply

treetrunkbranchstem • 2y ago •

Numpy calls out to C and C++, and imo as a practical matter I'd do the same with lisp.

2 Reply

ventuspilot • 2y ago •

u/n16f said "check the current best algorithms for matrix multiplication". At first I only thought of clever chache usage but there's more: IIUC the naive algorithm from the OP is $O(n^{3)}$, 1969 we went to $O(n^{2.807)}$, and since 2020 we are at $O(n^{2.373)}$.

The fastest matrix multiplication algorithm gives an intro and has more pointers in case anyone is still interested.

1 Reply

n16f • 2y ago •

Correct. There has been a ton of work to improve matrix multiplication algorithms over the years. Always improve the algorithm before having fun with micro optimization.

1 Reply

ventuspilot • 2y ago •

I did some more experimentation, currently I'm at 0.8s w/o SIMD and at 0.16 with SIMD for multiplying two 1000x1000 single-float matrices, all single-threaded. I think I'm doing the cache blocking wrong, though, as SIMD should be 8 times faster than non-SIMD, so there might be more to gain.

1 Reply

Steven1799 • 22d ago • Edited 22d ago •

u/ventuspilot, what did your final matrix code look like? I'm implementing some vector-matrix multiplication where performance is critical and am curious what you came up with when you added threading.

1 Reply

3 more replies

Fluffy_Professor_639 • 2y ago •

If OP is reporting 0.02 on numpy then you are already pretty close! Do tell what happens once you use multithreading

1 Reply

r/lisp

Celebrating 40 years of magic

298 upvotes · 18 comments

r/learnmath

When we multiply two matrices, why do we multiply a row to column of other matrix instead of simply multiplying the correspondent value of both matrices.

14 upvotes · 21 comments

r/functionalprogramming

My book Functional Design and Architecture is finally published!

294 upvotes · 38 comments

r/emacs

Company 1.0.0 released

172 upvotes · 25 comments

Why is matrix multiplication, "multiplication"?

69 upvotes · 21 comments

r/MachineLearning

[R] Multiplying Matrices Without Multiplying

394 upvotes · 69 comments

r/emacs

Does anyone else hit C-x C-s subconsciously whenever they are editing stuff?

103 upvotes · 42 comments

r/C_Programming

Language proposal: SIMD by adding/multiplying/etc arrays together

1 upvote · 15 comments

r/learnmath

Why multiply matrices? [Linear Algebra]

2 upvotes · 16 comments

r/learnmath

Why was matrix multiplication defined like that?

9 upvotes · 13 comments

r/excel

Excel formula to compare a 6 cell row against a reference row and return value that are missing

1 upvote · 11 comments

r/emacs

Emacs Docs. Had never seen these before, and thought they looked really good.

114 upvotes · 25 comments

r/learnmath

Why the multiplication between 2 matrices is so strange?

1 upvote · 13 comments

r/learnmath

3 upvotes · 5 comments

r/excel

Remove duplicates from a large amount of distinct columns

2 upvotes · 8 comments

r/C_Programming

Is there a more efficient way to write this C program?

2 upvotes · 7 comments

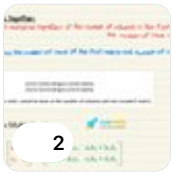
r/learnmath

is there 2 ways to multiply matrix by a vector?

2 upvotes · 16 comments

r/askmath

How does multiplying matrices work?



60 upvotes · 27 comments

r/excel

What formula can I use to pull in data from another sheet to match by both column and row?

15 upvotes · 17 comments

r/googlesheets

Sheets filled withGoogle form, want to calculate a sum on each row depending on the row above, need Arrayformula to do it automatically, but it creates a circular reference. Any idea ?

1 upvote · 4 comments

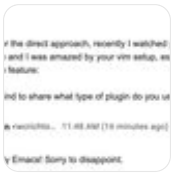
r/Clojure

Clerk: Local-First Notebooks for Clojure

158 upvotes · 14 comments

r/emacs

Amazing vim setup



My book 'Functional Design and Architecture' is finally released!

377 upvotes · 62 comments

r/PowerShell

Character comparison in PowerShell issue.

5 upvotes · 33 comments

r/haskell

A Dictionary of Single-Letter Variable Names

109 upvotes · 42 comments

TOP POSTS

Reddit

reReddit: Top posts of March 31, 2023

Reddit

reReddit: Top posts of March 2023

Reddit

reReddit: Top posts of 2023