

DRAFT: Polynomial Multiplication: Blocking to Improve Cache Performance

Ronen Gradwohl
Richard Fateman

July 10, 2002

Abstract

We search for techniques to decrease the multiplication time for large sparse polynomials in Lisp by speeding up the sequential accesses of large vectors. We do this by utilizing blocking to improve cache performance, which we show to be effective for sufficiently large problems.

1 Introduction

How can we speed up sparse polynomial multiplication? In these experiments we proposed to take advantage of blocking sections of polynomials. To the greatest extent possible we hoped to retain the normal formulation of algorithms for polynomial manipulation. We also wished to extend the (positive) results of Yozo Hida [2] who measured polynomial multiplication using C programs and Linux. Our goal was to explore whether they applied to other environments (Lisp, Windows 2000), and to larger examples.

By blocking we mean splitting up each of the polynomials p and q to be multiplied into a sum of blocks $p = p_1 + p_2 + \dots + p_n$, $q = q_1 + q_2 + \dots + q_m$, where each p_i contains (say) r terms, and each q_i contains s terms. Then performing the $m \times n$ block multiplications one block at a time increases memory locality, and should yield a decrease in computation time.

For specificity we are measuring performance on a Pentium III CPU¹ but the general notions should be relevant for other contemporary machine designs.

¹Total L1 cache size is 32Kbytes divided into two parts: 16Kbytes each for instructions and data, and a 256Kbyte L2 cache, 8-way set associative.

2 Polynomial Representation

We are using programs written in Lisp. It may seem that given this choice there would be a bias toward using linked lists for polynomials², but there are actually several plausible ways of representing a polynomial, each with its own merits and drawbacks. We could utilize lists, vectors, recursive vectors, and hashtables. Unsuccessful attempts to measure some of these more elaborate representations eventually led us to analyze a simpler problem, just to see if we could identify the effects of cache misses.

We abandoned the sparse model and chose a representation in which all polynomials were univariate. That is, the polynomial consisted of a vector of integers. For example, the polynomial $1 + x + x^2$ would be represented as a Lisp vector, denoted `#(1 1 1)`. The experiments were performed on arrays of various lengths from 100 to 128,000 terms in the polynomials to be multiplied. Also, as an additional way of isolating the effects of memory access and blocking, we short-circuited the actual arithmetic computation of the coefficients. We did, however force memory accesses to each coefficient³.

3 Measurements

In order to measure cache behavior we used Performance API (PAPI), a tool developed at the University of Tennessee[1]. PAPI facilitates access to hardware performance counters, and enabled us to record counts of L1 and L2 cache misses. The file `papi.lisp` contains the relevant functions for linking PAPI into a lisp environment. The only parts we directly use are `(start-ccm)` which initializes the L1, L2, and CPU time counters and `(read-ccm nil)` which reads out of those counters (and resets them to zero). This program returns a list of three numbers: L1 cache misses, L2 cache misses, and the total running time in milliseconds.

4 Results on Large Polynomials

When multiplying two polynomials of degree 128K, the effects of blocking were easily measured and clearly significant. Compared to unblocked access, blocking with sizes of $r = s = 100$ elements decreased the computation time from 390 to 200 seconds.

²In fact, many people do not realize there are many other choices in Lisp.

³If we had started by modeling dense polynomial arithmetic, we would have explored other techniques such as FFT, which would likely have been faster for large polynomials.

The number of L2 cache misses was 100 times smaller, and L1 cache misses 40 times smaller.

The runtimes of the computation with other block sizes or with smaller polynomials, represented a less dramatic savings.

We attempted to correlate the runtimes with measured cache misses, but it seems that small changes in the recorded counts of cache misses are not well-correlated with runtime, but large factors (100) in cache misses were predictably significant in runtime.

Some of the variation in time among the different block sizes seems inexplicable from the gathered data. We speculate that this simply reflects the external effects of other processes interspersed with ours, on cache performance.

Regardless of these issues, the speedup caused by blocking seems conclusive on large polynomials.

5 Results on Small Polynomials

On polynomials ranging from 1K to 32K terms, the effect of blocking was disappointing. Although different block sizes affected the performance slightly, *in our tests, the best performance was obtained with the simpler non-blocking computation.*

How can we explain this? First, there is some overhead in the loops to implement blocked multiplication. Specifically, the main part of that algorithm contains four nested loops, compared to only two in the non-blocking algorithm. Thus, in addition to using up more variables and more memory, the code also translates into longer machine code, which may affect its placement in the L1 instruction cache.

The second added cost to the blocking method is that it actually performs more memory references than the non-blocking method. Consider arrays x and y , with lengths k_1 and k_2 , and block sizes r and s . Also, suppose x_i is the i th term of x . In the non-blocking method, the loop accesses x_0 , and then y_0, \dots, y_{k_2-1} , then x_1 and y_0, \dots, y_{k_2-1} , and so on, totaling $k_1 \times k_2$ memory accesses. This is different from the blocking method (assume r divides k_1 and s divides k_2):

- access x_0 , then y_0, \dots, y_{s-1}
- access x_1 , then y_0, \dots, y_{s-1}
- continue until x_{r-1} , for a total of $r \times s$ memory accesses

- access x_0 , then y_s, \dots, y_{2s-1}
- access x_1 , then y_s, \dots, y_{2s-1}
- continue until x_{r-1} , for a total of $r \times s$ memory accesses
- repeat until all $\frac{k_2}{s}$ y -blocks have been multiplied by all of x_0, \dots, x_{r-1} . At this point, the y array has been accessed $r \times k_2$ times, and the x array has been accessed $r \times \frac{k_2}{s}$ times.
- continue through the rest of the r -blocks, which is $\frac{k_1}{r}$ times.

So in total, the number of memory accesses is $\frac{rk_2 + \frac{rk_2}{s}}{r} \times k_1$, which reduces to $k_1k_2 + \frac{k_1k_2}{s}$. Assuming that the current bottleneck is in the memory accesses, the $\frac{k_1k_2}{s}$ is significant for smaller s . We believe this accounts for the disappointing slowness of the blocking method compared to the non-blocking method.

6 Related Studies

While our data is clear for enormous polynomials (enormous: we figure it would take over 200 pages to print out just the input to the multiplication) the results of Yozo Hida [2] also on a Pentium III machine are definitive for smaller polynomials as well. He shows that for the problem of multiplying $(x + y + z)^{70}$ by itself, blocking can reduce the runtime by 40 to 50%. In his computation, each of the two polynomials has 2,556 terms, multiplied to produce 10,011 terms. For this size we failed to attain conclusive results. We speculate that his success can be accounted for by his use of Linux and C, as opposed to Windows and Lisp. The background processes running on Windows 2000 may have interfered with the caches enough to obliterate effects of blocking, and so for us, a larger computation was necessary in order to overcome that hurdle. At that size, however, the L1 cache effects are dominated by the L2 effects. In these tests, no Lisp garbage collection is invoked, so that is not an issue.

7 Conclusions

We conclude that for large enough polynomials, cache optimization via blocking is useful, and designs for polynomial algorithms should attempt

-bb-error = =

Figure 1: graph of time vs block size.

-bb-error = =

Figure 2: figure 2.

to accomodate to the cache sizes of the host CPU. “Large enough” polynomials need not be quite as large as given here, since a more usual polynomial would have larger-size coefficients (bignums) and worse locality. Improving the L1 cache performance, on the other hand, seems to be difficult in our computing environment.

Another problem is choosing a realistic model of representation for polynomials. The model we are using has sequences of coefficients in adjacent words in memory. Any more flexible storage model will be handicapped more-or-less by the loss of locality in representation, and thus decrease the effects of blocking. Hida’s results show that storing the polynomial as a vector but with indexing into it computed via a hashtable (essentially a super-elaborate array index) performed better than other alternatives. Introducing bignums in his tests makes blocking less effective, though we expect that if his tests approached the L2 cache size, the benefits of reducing subproblems below the L2 size would again emerge.

8 Appendix

need to add some graphs here

9 Appendix: programs

9.1 papi.lisp

```
;; The interface between PAPI and Lisp

;; load winpapi.dll from its standard place.
(load "c:/winnt/system32/winpapi.dll")
```

-bb-error = =

Figure 3: figure 3.

```

;; set up an array to store results from PAPI
(defparameter papi_values (make-array 80 :element-type '(unsigned-byte 8)
:initial-element 0))

;; PAPI uses long-long integers, not in lisp. We use an array of bytes
;; and convert to lisp bignums when necessary.
;; bytes to long long (bytes2ll) does the conversion
(defun bytes2ll(ba) ;; 8 bytes represent a long-long. convert to bignum
  ;; ba is a byte array, return array of unsigned long longs as bignums.
  (let* ((len (truncate (length ba)8))
    (a (make-array len :initial-element 0))
    (i8 0)
    (ans 0))
    (dotimes (i len a) ;return array a
      (setf i8 (1- (* (1+ i) 8)) ans 0)
      (dotimes (j 8 (setf (aref a i) ans))
        (setf ans (+ (aref ba (- i8 j))(* 256 ans)))))))

;; These declarations link the lisp programs to corresponding entry
;; points in the PAPI dll.

(ff:def-foreign-call
  (papi_start_counters "PAPI_start_counters")
  ((flags (* :int)) (len :int))
  :returning :int)

(ff:def-foreign-call
  (papi_read_counters "PAPI_read_counters")
  ((counters (* :int)) (len :int))
  :returning :int)

(ff:def-foreign-call
  (papi_stop_counters "PAPI_stop_counters")
  ((counters (* :int)) (len :int))
  :returning :int)

(defparameter *last-time* 0)

(defun start-ccm(); count L1 L2 total cache misses

```

```

    (let((ar2
(make-array 2 :element-type '(unsigned-byte 32)
      :initial-contents (vector papi_l1_tcm papi_l2_tcm))))
      (setf *last-time* (get-internal-run-time))
      (papi_start_counters ar2 2)
    ))

(defun read-ccm(printp)
  (papi_read_counters papi_values 2)
  (let* ((ans(bytes2ll papi_values))
        (newtime (get-internal-run-time))
        (diff (- newtime *last-time*)) )
    (setf *last-time* newtime)
    (if printp (format t "~% L1 cache misses = ~e, L2 cache misses=~e, runtime=~s"
(float (aref ans 0))(float (aref ans 1))diff)
      (list (aref ans 0)(aref ans 1)diff))))

```

9.2 block timing

```

;; x and y are the arrays to be multiplied
;; bfn is the name of the blocking function
;; nbfn is the name of the non-blocking function
;; filename is the file to output the data

```

```

(defun blockloop (x y rmax smax rstep sstep bfn nbfn filename)
  ; compiler optimizations:
  (declare (optimize (speed 3)(safety 0)(debug 0)))
  (start-ccm) ;; start cache counting
  (with-open-file (file filename :direction :output
    :if-does-not-exist :create :if-exists :overwrite)
    (format file "~% xlen=~s ylen=~s rmax=~s smax=~s rstep=~s sstep=~s"
      (length x)(length y) rmax smax rstep sstep)
    (let ((cache-stuff (read-ccm nil)))
      (dotimes (i 3)
        (funcall nbfn x y)
        (setf cache-stuff (read-ccm nil))
        (format file "~%--nonblocking average of 3 runs, L1, L2, RT(ms) ~s"
          (mapcar #'(lambda(r)(round r 3)) cache-stuff))
        (do

```

```

        ((i rstep (+ i rstep)))
        (> i rmax))
(do
  ((j sstep (+ j sstep)))
  (> j smax))
  (read-ccm nil) ; clear cache registers
  (funcall bfn x y i j)
  (setf cache-stuff (read-ccm nil))
  (format file "%L1[~s,~s]= ~s;" (truncate i rstep)
  (truncate j sstep)(first cache-stuff))
  (format file " L2[~s,~s]= ~s;" (truncate i rstep)
  (truncate j sstep)(second cache-stuff))
  (format file " RT[~s,~s]= ~s;" (truncate i rstep) ;; run time
  (truncate j sstep)(third cache-stuff))
  (force-output file))))))

;; The program bt does non-blocking vector multiplication.
;; (bt does not actually perform the multiplication, only
;; memory accesses)
;; x and y are the arrays to be multiplied
(defun bt (x y)
  (declare (ignore r s) (optimize (speed 3)(debug 0)(safety 0)))
  (let ((xi nil) (lenx (length x))(leny (length y))
  (res (make-array (1- (+ (length x)(length y))) :initial-element 0)))
    (declare (simple-array res x y)(fixnum r s lenx leny))
    (dotimes (i lenx res) (declare (fixnum i))
      (setf xi (svref x i))
      (dotimes (j leny)(declare (fixnum j))
        (and (svref res(+ i j))(and xi (svref y j)))))))

;; The program gbt does multiplication with blocking.
;; x and y are the vectors to be multiplied.
;; r and s are the respective block sizes.
(defun gbt(x y r s)
  (declare (fixnum r s)
  (simple-array x y)
  (optimize (speed 3)(safety 0)(debug 0)))
  (let* ((lenx (length x))

```

```

(leny (length y))
(res (make-array (+ lenx leny -1) :initial-element 0))
(xi nil)
(nx 0) (remx 0)
(ny 0) (remy 0)
(sny 0) (rnx 0))
  (declare (fixnum rii irii sjj jsjj nx remx ny remy sny rnx lenx leny
            i ii iilim j jj jjlim)
           (simple-array res))
  (multiple-value-setq
(nx remx) ; number of blocks for x, remaining blocks
  (truncate lenx r))
  (multiple-value-setq
(ny remy) ; number of blocks for y, remaining blocks
  (truncate leny s))
  (setf sny (the fixnum (* s ny)))
  (setf rnx (the fixnum (* r nx)))

  (let ()
    (do ((ii 0 (+ ii r))
        (iilim r (+ iilim r)))
      ((eql ii rnx)) ;done with nx blocks of size r
      (declare (fixnum ii iilim))
      (do ((jj 0 (+ jj s))
          (jjlim s (+ jjlim s)))
        ((eql jj sny) ) ;done with ny blocks of size s
        (declare (fixnum jj jjlim))
        ;; begin block loop
        (do ((i ii (1+ i))) ; from 0 to r-1, r to 2r-1, ... , nx times
          ((eql i iilim) )
          (declare (fixnum i))
          (setf xi (svref x i))
          (do ((j jj (1+ j))) ; from 0 to s-1, s to 2s-1, ..., ny times
            ((eql j jjlim) )
            (declare (fixnum j))
            (and (svref res (+ i j)) (and xi (svref y j)))
            ))))
        )))

  ;; if remy > 0, we output x[i]*y[s*ny+j]; all i and j=0 to remy
  (if (> remy 0)

```

```

;; there are remy more y items
(dotimes (i lenx) (declare (fixnum i));;all of x
  (setf xi (svref x i))
  (do ((j sny (1+ j));;small part of y
      ((eql leny j)
       (declare (fixnum j))
       (and (svref res (+ i j)) (and xi (svref y j))))))

      ;; if remx > 0 we output y[j]*x[i]; i from 0 to remx, j =0 to sny
      (if (> remx 0)
          (do ((i rnx (1+ i)) ;;small part of x
              ((eql i lenx)
               (declare (fixnum i))
               (setf xi (svref x i))
               (do ((j 0 (1+ j)) ;; big part of y, not all
                   ((eql j sny) ) (declare (fixnum j))
                   (and (svref res (+ i j)) (and xi (svref y j))))))
              ;; all parts done
              res)))

```

References

- [1] PAPI. <http://icl.cs.utk.edu/projects/papi/>.
- [2] Y. Hida. Data Structure and Cache Behavior of Sparse Polynomials. CS282 term paper, University of California, Berkeley. 2002.