Winfried Neun Herbert Melenk

Implementation of the LISP-Arbitrary Precision Arithmetic for a Vector Processor

Preprint SC 88-1 (January 1988)

Konrad-Zuse-Zentrum für Informationstechnik;
Heilbronner Straße 10; D-1000 Berlin 31

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
1000 Berlin 31
Verantwortlich: Dr. Klaus André
Umschlagsatz und Druck: Verwaltungsdruckerei Berlin

Contents

Abstract

Portable Standard LISP (PSL, Version 3.4) and REDUCE 3 were implemented for Cray 1 and Cray X-MP computers at the Konrad Zuse-Zentrum Berlin in 1986. As an special aspect of the implementation of PSL, an interface to the vector hardware of Cray processors was defined. With that interface and mostly driven by the needs of REDUCE applications (e.g. extensive calculations of Gröbner bases), the arbitrary precision integer arithmetic of PSL was rebuild using full power of the vector hardware. A modular arithmetic using vector hardware was also constructed.

1. Introduction

The Version 3.4 of Portable Standard LISP [1], a dialect of LISP developed at the University of Utah, was implemented for Cray-1 and Cray X-MP computers at the Konrad Zuse-Zentrum Berlin in 1986 for the operating system COS and in 1987 for the operating system UNICOS. This implementation was a continuation of the earlier work on PSL version 3.2 at the University of Utah, Los Alamos Laboratories and Cray Research Inc. [2]. PSL 3.4 also serves as a supporting LISP system for REDUCE, which is distributed for Cray computers by ZIB Berlin [3].

The Cray version of REDUCE 3 is one of the fastest implementations available measured by the Reduce Standard test [4]. This test does not require nonstandard integers at least on a Cray computer with its 64 bit words. With the new version, REDUCE 3.3 [5], the calculation of Gröbner bases was included and became a focus of the "Symbolic" group activities at ZIB. Some runtime analysis pointed out that the dominant part of the cpu consumption was caused by the integer arithmetic with arbitary precision, the PSL "bignums". One approach to solve "bigger" problems with Gröbner bases was to speed-up the arithmetic. It was not too difficult to get the idea of using the power of Cray vector hardware for bignum arithmetic.

Some runtime analysis found out that the bignums which occur during Gröbner bases calculations are of moderate sizes which means their representation has some hundred decimal digits, in most cases below 1000 digits. Our task is different from those problems that occur when the calculation of PI to millions of digits is attempted, i.e. the asymptotic behaviour is not our main interest.

Since we cannot expect that the Cray hardware architecture is known, we give a very short introduction, which will give some points for the later discussion on the arithmetic.

2. Cray Hardware Outline

Cray-1 processors [6] were first delivered in 1976 and are characterized and distinguished from usual computer systems by the main features: *functional units*, *instruction pipeline* and *Vector registers*. The following description is a very rough summary based on the Cray X-MP architecture, without any idea of being 100% exact or complete.

Functional Units: The "classic" cpu is divided into several specialized units (in this case 13) which are able to do one special task each. A unit can operate independently from other units in parallel, e.g. the add integer can work in parallel with the floating point add, given the fact that the destinations and sources are disjoint. There are no functional units for multiplication integer or division integer, these have to be done via float operations.

Instruction Pipeline: The computer is (potentially) able to issue one instruction each cycle (or each two cycles), which is about 100 millions per second. This is true if no jumps are taken out from the sequence of instructions and if the functional units are not blocked by earlier instructions. To sustain this instruction rate, instructions have to be buffered in cpu and a jump to a location outside the instruction buffers will cause a tremendous delay.

Vector Registers: The Cray computer has 8 vector registers which contain 64 (64 bit) words and which are operated by functional units with a rate of one per cpu cycle giving a dramatic speed-up result for some computations in FORTRAN programs. The result of a vector operation may be used directly by another vector operation before the first instruction is finished (chaining).

Note on the instruction timings: The memory access is slow in comparison with the arithmetic operations, just to give some idea: loading an integer from memory takes 13 cycles, add integer takes 3 cycles.

Examples:

a) The speed-up factor between scalar and vectorized quotient of two vectors is 5 for a modest vector length of 10 items.

b) Copying a vector from location a to location b takes about the same time as adding two vectors $a + c$ and storing the result to b, which is also the approximate time to be consumed if we multiplied a and c and stored it to b.

c) In many cases the estimation of the load time is a good estimation for the overall cpu-time and since many operations may be done in the shadow of a load.

Some impacts (programming rules) which were realized in parts automatically in Cray PSL 3.4 via a scheduling path in the PSL compiler:

- Avoid access to memory whenever possible.
- Try to write the LISP operations so that they intermix the usage of funtional units and registers.
- Avoid jumps whenever possible.

On system programming level with vectors:

- Try to keep data in registers especially in vector registers without storing in memory.

A general idea is that an analysis of an algorithm with respect to its number of operations is not the whole story on a vector processor (see example b

above). However, the simplicity of the algorithm and its data structure may be more important. It is well-known that a more sophisticated algorithm which saves operations is often hard to recode with a vector computer as a target hardware.

3. PSL Bignums

Portable Standard LISP's bignum package was developed by M. L. Griss, B. Morrison and A. C. Norman in 1982. It uses representation for its arbitrary precision integers which is very well suited for Cray vector operations: a PSL vector (which is a number of LISP items stored continuously in heap memory) with a number of bits in each item, such that the product of two items can be computed without loss of bits into one word :

[length — sign — small integer — small integer— ... — small integer]

Convention: all small integers are positive.

With Cray, the bound for the small integers is 2**21, so 43 bits in a word are lost. This is due to some curiosities in the arithmetic hardware. The multiplication of integers has to be done via the floating point format since there is no functional unit for multiplication integer or division integer. The floating point format on Cray computers is 16 bits for sign of mantissa and exponent and 48 bits for mantissa. The reason why fast multiplication and division integers are possible with at most 46 bit results which restricts the word size for the bignums. A compression and decompression "on the fly" seems to be too expensive.

A. Example for system programming with vectorized bignums

As a first simple example, let us assume that we have to add two positive bignums with the same length - less than 63. The notation is described in the Usage of Vector [7] by H. Melenk and W. Neun.

We load the integers into two different vector registers and add the registers:

```
(VsetVL actuallength)                 % length of both numbers
(Vload (vreg 1) arg1 1)               % load numbers from arg1 into V1
(Vload (Vreg 2) arg2 1)               % same with arg2 and V2
(VplusV (Vreg 0) (Vreg 1) (Vreg 2))   % V0 = v1 + v2
```

Now we have to worry about the carries:

```
(Vrshift (Vreg 3) (Vreg 0) 21)      % shift all items of a Vector
                                    % register by 21 into another one
(VandS (Vreg 4) mask (Vreg 0))      % mask out overflow bits
                                    % mask = 2**21 -1
```

Up to this point all operations were done in a chaining mode, i.e. the last item is read after the first has left the vector logic functional unit. We have now the following register contents:

V4 is the vector of sums cleared from carries.

V3 is the vector of carries which must be shifted by 64 bits into the "right" neighbor word, so that the addcarry does a true "carrying".

If the highmost digit of the carry is not zero, we have to increase size of operation by 1. This code is not presented here explicitly.

```
    (setq Vmask (VtestN (Vreg 3)))      % a mask indicating nonzero
                                        carries
                                        % if zero, we are ready
    ADDCARRY
    (while (not (eq Vmask 0)))          % until all carries eaten up
    (when (overflown?) (increase-length))
*   (Vdrshift (Vreg 5) (Vreg 3) 64)
    (VplusV (Vreg 6) (Vreg 4) (Vreg 5)) % the addcarry
*   (Vrshift (Vreg 3) (Vreg 6) 21)      % shift carries to the right
**  (setq Vmask (VtestN (Vreg 3)))      % a mask indicating
                                        % nonzero carries
**  (VandS (Vreg 4) mask (Vreg 0))      % mask out overflow bits
    (Vstore (Vreg 4) result 1)          % finally get rid of result
```

In the add-carry loop, the instructions indicated by * and ** respectively use the same functional unit in order for the second operation to wait, but multiple carry propagation seldom occurs.

When the operands length exceeds 63 items we subdivide the problem into pieces of 63 items, where each problem delivers the carry to the next subproblem. From this simple example we can see that the vector coding

produces lots of instructions and the code is not very good readable. Fortunately, it is fast: the ratio between scalar and vector addition of bignums is shown in the figure. The unsmooth parts of the curve are caused by the 64 words length of the vector registers.

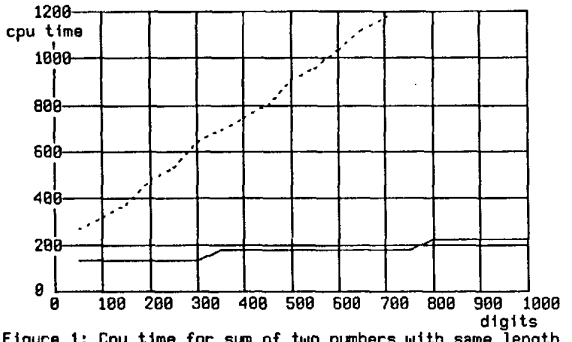

Figure 1: Cpu time for sum of two numbers with same length (nano seconds / decimal digits). Dotted line: classical version, full line: vectorized version

B. Basic algorithms for bignum arithmetic

The algorithms for multiplication of bignums have been discussed by many authors, e.g. [8]. In spite of the common recommendation to use the Fourier transform approach, we implemented for PSL on Cray the "vectorized schoolboy" method. This was done because this method is simple and therefore "easily" vectorizable. Likewise, the lengths of the bignums which are used by our REDUCE applications are not too great, that the asymptotic behaviour of the algorithms are not of interest in almost any cases. In multiplication we can get benefits from the modest bits per item size we use because we can add up the partial products "infinitely" without an overflow. There is a maximum of 64 partial products that form the final result, so we can do the carry propagation when the final result is written to memory.

Figure 2 shows the cpu consumption of the scalar and the vectorized algorithm. It can be seen that the $O(n^2)$ curve of the scalar algorithm is flattened dramatically so that the break even point between Fourier and schoolboy approach is far beyond the range of numbers in which we are interested in.

For quotient/remainder computation we use a two-fold approach: if the operands can both be held in vector registers, we use an algorithm that is based on the estimation of the result from the float quotient of the highmost digits of the bignums and keep the operands in vector registers all the

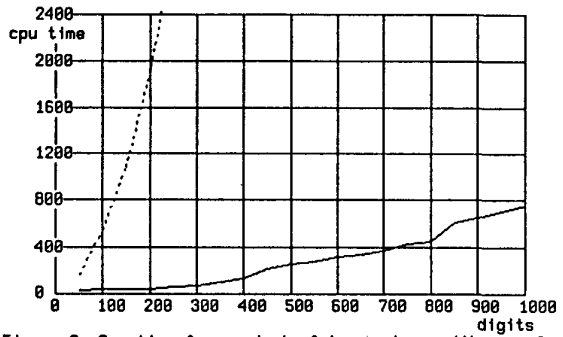

Figure 2: Cpu time for product of two numbers with same length (nano seconds / decimal digits). Dotted line: classical version, full line: vectorized version

time. In case the operands are bigger, we use a Newton iteration for the estimation of the reciprocal (shifted accordingly). This iteration naively coded had an interesting effect: the cpu time was about 4 times faster than Knuth's algorithm D [9], which is PSL bignum standard. But, in an overall summary it was a little bit slower since it used lots of intermediate results and furthermore, lots of extra garbage collections had been done. We solve that problem by explicite assignment of working memory.

Figure 3 shows the behaviour of the vectorized algorithm - Newton approach vs. Knuth's D algorithm.

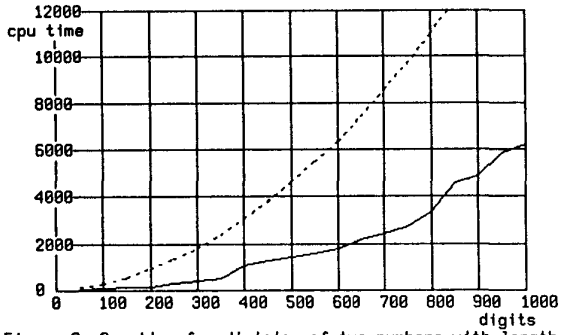

Figure 3: Cpu time for division of two numbers with length n and n/2 (nano seconds / n=decimal digits). Dotted line: classical version, full line: vectorized version

The gcd is used very often in rational arithmetic and causes heavy usage of the remainder algorithm. The gcd is computed via Euclid's algorithm. This is done mostly because it is a simple extension of the quotient/remainder

algorithm described above. This means that, for operands less than 64 words the resulting gcd is computed completely in a vector register without using extra working memory. The binary algorithm (e.g. [9]) may be valuable, but our experiences with that algorithm have shown results that there are orders of magnitude slower than Euclid's algorithm even if we used vectorized add and shift operations. This may be caused by the fact that the shift operation of a bignum vector is not so easy and the "even" test which is required is expensive because of the access time to a single word in a vector register.

4. Computations with Modular Numbers

In Knuth's book, an interesting alternative for bignum representation is outlined. Given a set of prime numbers, one can define a modular arithmetic by representing a number by their moduli relative to that set of primes applying the usual modular arithmetic for each component. By virtue of the Chinese Remainder Algorithm, one can reconstruct the number if it is less than the product of all primes.

The elementary operations are defined by:

$$\begin{aligned} [a_1, \cdots, a_n] \; \alpha \; [b_1, \cdots, b_n] &= [a_1 \; \alpha \; b_1, \cdots, a_n \; \alpha \; b_n] \\ \text{where } \alpha &\in \{+_{mod}, -_{mod}, *_{mod}\} \end{aligned}$$

The reciprocal is defined by the co-factor result of the Extended Euclidian Algorithm:

$$1/_{mod} \; a_i = c_i, \quad \text{where } c_i \text{ is the solution of}$$
$$c_i * a_i + x_i * mod_i = gcd(a_i, mod_i) = 1$$

where mod_i is the *ith* modulus and the gcd is 1 since mod_i is a prime.

This arithmetic is a very natural approach for a vector processor since we are able to do vector operations very easy and fast with many relatively small moduli. Therefore, these routines will be available in Cray PSL. This representation is adequate for very large Gröbner base calculations which use many bignum operations. There are some limitations of that method which inhibit the general usage of that representation for bignums in PSL namely:

- The representation is independent from the size of the actual number, a small number does not save any storage. For special algorithms a vector mode compression/decompression can be applied, but in general a wasting of heap space is implied.
- The modular computation per definition does not detect overflow of the numbers above the product of moduli unless they are converted back by the Chinese Remainder Algortihm and Wang's Algorithm [10].

- The conversion from modular representation to a long integer format, e.g. for printing a number is very expensive. Knuth already pointed out that this type of arithmetic is valuable only if the algorithm using it is heavy compared with the conversion.
- The modular representation format does not allow easy comparison operations like greaterp and lessp.

5. Conclusions

We presented our approach for arbitrary integer arithmetic on a special computer system. We feel that in the special environment of a vector computer some of the usual algorithm analysis do not give the correct results. Unfortunately, a correct analysis in our sense is to a large extent hardware dependent and will be obsolete with some new computer systems. Likewise, some cumbersome effects are due to our special hardware. On the other hand, a dramatic speed-up was achieved in some of our applications.

Recent improvements of the REDUCE factorizer done by J. Davenport include an interface for the computation with vectors of small integers. The implementation of that interface in Cray vector code shows a remarkable speed-up in combination with Gröbner base calculations.

References

[1] M. L. Griss, A. C. Hearn: *A Portable LISP Compiler, Software Practice and Experience.* Vol. 11 (1981).

[2] J. W. Anderson, W. F. Galway, R. R. Kessler, H. Melenk, W. Neun: *Implementing and Optimizing Lisp for the Cray*, IEEE Software, July 1987.

[3] H. Melenk, W. Neun: *REDUCE Users Guide for Cray1/X-MP Series Running COS*, Konrad Zuse-Zentrum für Informationstechnik Berlin, Technical Report TR 87-4, August 1987.

[4] J. B. Marti, A. C. Hearn: *REDUCE as a Lisp Benchmark*, ACM SIGSAM Bulletin, Vol. 19 No. 3.

[5] A. C. Hearn: *REDUCE Users Manual*, Version 3.3, The Rand Corporation (1987).

[6] *Cray Computer Systems, X-MP Series Mainframe Reference Manual*, Cray Research Inc., Mendota Heights (1983).

[7] H. Melenk, W. Neun: *Usage of Vector Hardware for LISP Processing*, Konrad Zuse-Zentrum für Informationstechnik Berlin (1986).

[8] J. D. Lipson: *Elements of Algebra and Algebraic Computing*, Benjamin/Cummings Publishing Company (1981).

[9] D. E. Knuth: *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, 2nd Edition (1981).

[10] P. S. Wang: *A p-adic Algorithm for Univariate Partial Fractions*, ACM, Proc. SYMSAC (1981).