Vector implementation of multiprecision arithmetic

Richard Crandall and Jason Klivington
Advanced Computation Group, Apple Computer

Abstract. We describe herein an implementation of arbitrary precision arithmetic using the new PowerPC Velocity-Engine (G4) vector instructions. We first define essential digit size of a multiprecision integer to be 128 bits, in view of the Velocity Engine architecture. By designing a general purpose vector library comprised of multiplication, square, bit shift and bitwise comparison, we have achieved significant speedups over PowerPC G3 scalar (base-2^{16}) implementation, or even hand-tuned assembly (base-2^{32}) packages. In particular, vector multiplications enjoy performance improvements ranging from 3:1 to 10:1 over their scalar counterpart, depending on operand bit size. An application for general purpose integer factoring was written and is able to factor large integers with speedups factors roughly in the aforementioned range. A second, large convolution application was written to settle a certain research problem; in this instance a length-2^{15} convolution of 512-bit elements is performed. On a 300 MHz max!-based prototype Macintosh, one such operation consumes 10 seconds, currently the fastest known time for integer convolution relevant to the motivating problem.

25 October 1999

1. Multiprecision elements

The various motives for multiprecision implementation are well known. There are cryptography and computational number theory, which two fields are closely related. In such fields there is never enough speed, in the sense that practical limits for computation have cultural importance; for example factoring limits determine worldwide security standards, and so on. But there is also a need for multiprecision floating point, and it is usually the case that one builds this up from multiprecision *integer* routines. Moreover, multiprecision implementations test very well a vector-architecture machine, and this motive is important in regard to the new G4 processor. These brief motivational remarks having been made, we concentrate hereafter on theory and detail of our Velocity Engine multiprecision integer implementation, reporting results of both low-level and high-level (application) timing.

In the world of multiprecision arithmetic it is well known that the most common, and therefore from a performance standpoint the most important, composite operation is of the form:

$$(x * y) \bmod N,$$

and furthermore the most common scenario has each of x, y the same bit-size as N. For example, in cryptography and computational number theory in general, we are interested in primality testing, integer factoring, and so on; and in such studies the above composite operation is usually performed for a stable N, or for just a few distinct values of N. This "mul-mod" operation appears to involve multiplication and division, but the so-called Barrett method for modular reduction uses multiplications alone. Specifically, if one defines a "steady-state reciprocal" r of N, via:

$$r = \lfloor 4^{B(N-1)}/N \rfloor,$$

where $B(x)$ denotes the bit-length of any nonnegative x, with $B(0) = 0,\ B(1) = 1,\ B(15) = 4$ and so on, then modular reduction can be effected using this reciprocal and multiply/add routines alone. It turns out that r itself can be evaluated via a Newton iteration—involving itself no explicit division–with the result that modular reduction can be performed via:

$$z \bmod N = z - N * \lfloor (z * r) >> s \rfloor - \epsilon N,$$

where $s = 2(B(r) - 1)$, and ϵ is a very small error that is then removed via a small number (if any) of subtractions of N. Thus z mod N can be effected without explicit division, and if N is indeed "steady-state" we can recycle the reciprocal r *ad infinitum*. Casual inspection of this method shows that a mod (or div) operation can be achieved via 3 size-N multiplies in the steady state. But one can intervene into the multiplication process for $z * r$ and ignore some bits, etc. to further reduce the work. It turns out that mod (or div) can be brought down to about 1 multiply, with extra work in the detailed loop operations. We did not go that distance, but partially reduced the work in favor of reasonably simple code, so that our mod (or div) is roughly 2 multiplies. These machinations show that a multiprecision Velocity Engine mod (or div) can be effected in 2 Velocity Engine multiplies, and this is without any classical long division of any kind.

Being as so much work comes down to multiplication, we concentrated heavily—although not exclusively—on Velocity Engine multiply routines. If one represents x in digit form

$$x = \{x_0, ..., x_{D-1}\},$$

where each x_i is an element of $[0, ..., b-1]$ where b is the base, then multiplication of x, y having D digits each is the acyclic convolution

$$x * y = \sum_{m=0}^{2D-1} (\sum_{i+j=m} x_i y_j) b^m.$$

The inner sum (over $i+j$) is manifest in sofware as a standard, "grammar-school" multiply loop, and that is where Velocity Engine excels. For actual Velocity Engine implementation, we chose digit size

$$b = 2^{128},$$

so that the basic quantization of large integers into digits is based on 128-bit segments, and the detailed multiplication steps are based on vectors of that length (see Section 3).

For squaring *per se* the acyclic summation can be simplified due to redundancy; i.e. the fact that x, y have identical digits cuts the multiplication work asymptotically in half. This having been said about the fundamental operation $(x * y) \bmod N$, we briefly overview other algorithms on which Velocity Engine advantage was focused.

Outside of the domain of multiplication and squaring, there are yet more Velocity Engine advantages. It is well know that for certain moduli N, in particular when

$$N = 2^q \pm k,$$

with k small, the mod operation can be effected via additions (subtractions) and shifts alone. This is the basis for Apple Computer's proprietary crypto system FEE, in which one chooses fields over primes of the form $2^q \pm k$, yielding the fastest known elliptic curve system at given bit-depth. Similarly, many modern factoring experiments use Mersenne numbers $N = 2^q - 1$ or Fermat numbers $N = 2^q + 1$, and so enjoy Velocity Engine acceleration. While the Velocity Engine is not best suited to the task of addition and subtraction *per se* of multiprecision values, its vector-shift options provide significant speed enhancements.

2. Summary of Velocity Engine performance

Figures 1,2,3 show the performance of Velocity Engine vs. a scalar (base $b = 2^{16}$) implementation. All timings (for both scalar and vector implementations) were performed on a 300 MHz max!-based prototype Macintosh. Note the natural quantization of performance on 128-bit boundaries. As for the performance itself, let us write the time for multiplication in the general form:

$$T_n = an^2 + bn + c,$$

where a represents the difficulty of multiplication of digits, b is the difficulty of addition *per se*, and c is overhead. The reason that Figure 2 is "less parabolic" than Figure 1 is

that, for the Velocity Engine implementation, the constant a is so very small (i.e., the vector multiplication is so very fast). Speedup factors in the range 3:1 to 10:1 are evident in Figure 3.

We observed that for the Velocity Engine code, a more modern multiplication algorithm — meaning a method more efficient asymptotically than grammar-school multiply — should be used beyond the 3000-bit region. The Karatsuba algorithm breaks each operand into two (roughly) equal-length bit strings, and uses the identity:

$$x * y = (a + bW) * (c + dW) = \frac{t+u}{2} - v + \frac{t-u}{2}W + vW^2,$$

where

$$t = (a + b) * (c + d), \quad u = (a - b) * (c - d), \quad v = b * d.$$

If W is a convenient power $2^e \sim \sqrt{x} \sim \sqrt{y}$, these last three multiplies to get t, u, v, together with some shifts by multiples of e bits, are all that is required to obtain $x * y$. In this way the work of 4 size-W multiplies is done in 3 of same, leading, upon recursive calling of this Karatsuba scheme, to a complexity

$$T = O(N^{\log 3/ \log 2})$$

for multiplication of very large integers. This complexity reduection over grammar-school multiply is substantial, especially when one gets to 10000 or 100000 bit operands, e.g. for a 100000-bit multiply the Karatsuba scheme on the Velocity Engine gains a full factor of 4 over grammar-school Velocity Engine multiply. Figure 4 shows the result of our Velocity Engine implementation of Karatsuba, which as we have said takes over at 3000-bit operands threshold. The slope, being close to $\log 3/ \log 2 \sim 1.6$ is in good agreement with theory. Incidentally the Velocity Engine breakover at 3000 bits is relatively high; most systems require a Karatsuba breakover much less than this. The reason is, of course, the fast fundamental 128-bit multiplication which drives the threshold upward.

When the above multiplication routines, together with the other routines such as shifting, are used in an actual application, the Velocity Engine speedups come down somewhat, depending of course on the application overhead that is not relegated to Velocity Engine code. See Section 5 for timing information on an actual factoring application,

3. Implementation details

Since the fundamental calculation for all general-case multiplications that we perform is the aformentioned sum

$$\sum_{i+j=m} x_i y_j,$$

with each x, y being a 128-bit digit, we shall first discuss the Velocity Engine implementation of this operation.

First we examine a typical product $x_i y_j$. If we break each 128-bit value down into 16-bit elements, then $x_i y_j$ can in turn be represented more finely as

$$x_i * y_j = \sum_{m=0}^{14} (\sum_{k+l=m} a_k b_l) 16^m.$$

Representing this visually, we have the familiar parallelogram:

$$\begin{array}{cccccc} & & a_7b_0 & \cdots & a_1b_0 & a_0b_0 \\ & a_7b_1 & \cdots & a_1b_1 & a_0b_1 & \\ & \vdots & & \vdots & & \\ a_7b_7 & \cdots & a_1b_7 & a_0b_7 & & \end{array}$$

which we add vertically with carry, the result being—in general—nearly a 256-bit value. To effect all of this, we establish eight 32-bit vectors $S_0 ... S_7$, each of which initialized to zero. Next, using the Velocity Engine merge and shift operations, we create four vectors, as follows:

$$A_3 = \{0, 0, 0, 0, 0, 0, 0, 0\}$$

$$A_2 = \{0, 0, 0, 0, 0, 0, 0, a_7\}$$

$$A_1 = \{a_7, a_6, a_6, a_5, a_5, a_4, a_4, a_3\}$$

$$A_0 = \{a_3, a_2, a_2, a_1, a_1, a_0, a_0, 0\}$$

As a third initialization step, we create a vector:

$$B = \{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$$

Now, using the Velocity Engine `perm` (permute) operation, we create two vectors of 8-bit elements, from elements selected out of B:

$$C_0 = \{0, b_{1l}, 0, b_{0l}, 0, b_{1l}, 0, b_{0l}, 0, b_{1l}, 0, b_{0l}, 0, b_{1l}, 0, b_{0l}\}$$

$$C_1 = \{0, b_{1h}, 0, b_{0h}, 0, b_{1h}, 0, b_{0h}, 0, b_{1h}, 0, b_{0h}, 0, b_{1h}, 0, b_{0h}\}$$

with $b_{\#l}$, $b_{\#h}$ being respectively the low/high 8 bits of a 16-bit value $b_\#$. At the core of our multiply calculation is the Velocity Engine's powerful `msum` operation. This operation takes three inputs: two 16-bit-element vectors r, s, and a 32-bit-element vector t, and produces a 32-bit-element vector u. The `msum` opcode takes the two vectors r, s, and multiplies their corresponding 16-bit elements together to yield 32-bit results. Adjacent even and odd 32-bit results are then summed to yield one 32-bit-element vector, whose elements are then added to the corresponding 32-bit elements of vector t to produce the resulting 32-bit-element vector u (all of this accomplished in four clock cycles!).

Taking $r = A_0$, $s = C_0$, and $t = S_0$, if we perform the operation

$$S_0 = msum(A_0, C_0, S_0)$$

we obtain the vector

$$S_O = \{a_3b_{1l} + a_2b_{0l}, \quad a_2b_{1l} + a_1b_{0l}, \quad a_1b_{1l} + a_0b_{0l}, \quad a_0b_{1l} + 0\}.$$

Note that each 8-bit by 16-bit result produces a 24-bit result that is added with another 24-bit value. If we were to perform a full 16-bit by 16-bit multiply, we would have 32-bit products, which when added together would possibly overflow our 32-bit vector elements. By restricting ourselves to 24-bit results, we eliminate the need for keeping track of carries, and can perform several `msum` operations before our result vectors are saturated. If we perform a similar calculation for S_1, namely

$$S_1 = msum(A_0, C_1, S_1)$$

then we get:

$$S_1 = \{a_3 b_{1h} + a_2 b_{0h}, \quad a_2 b_{1h} + a_1 b_{0h}, \quad a_1 b_{1h} + a_0 b_{0h}, \quad a_0 b_{1h} + 0\}.$$

Note that if we take the vector S_1 and shift all of its elements left by eight bits, then add S_0 and S_1, the resulting four 32-bit elements correspond to the sums of the four rightmost elements of the top two rows of the paralellogram. We continue by calculating $S_2 ... S_7$:

$$S_2 = msum(A_1, C_0, S_2)$$

$$S_3 = msum(A_1, C_1, S_3)$$

$$S_4 = msum(A_2, C_0, S_4)$$

$$S_5 = msum(A_2, C_1, S_5)$$

$$S_6 = msum(A_3, C_0, S_6)$$

$$S_7 = msum(A_3, C_1, S_7)$$

At this point, we have calculated all partial products for the first two rows of the parallelogram. If we now use the Velocity Engine `perm` operator, we can shift all of $A_3 ... A_0$ left by 4 elements, to obtain:

$$A_3 = \{0, 0, 0, 0, 0, 0, 0, 0\}$$

$$A_2 = \{0, 0, 0, a_7, a_7, a_6, a_6, a_5\}$$

$$A_1 = \{a_5, a_4, a_4, a_3, a_3, a_2, a_2, a_1\}$$

$$A_0 = \{a_1, a_0, a_0, 0, 0, 0, 0, 0\}$$

Next, we shift right the vector B by 2 elements, yielding:

$$B = \{0, 0, b_7, b_6, b_5, b_4, b_3, b_2\}.$$

This allows the same permute operation used before to regenerate the vectors C_0, C_1, with

$$C_0 = \{0, b_{3l}, 0, b_{2l}, 0, b_{3l}, 0, b_{2l}, 0, b_{3l}, 0, b_{2l}, 0, b_{3l}, 0, b_{2l}\}$$

$$C_1 = \{0, b_{3h}, 0, b_{2h}, 0, b_{3h}, 0, b_{2h}, 0, b_{3h}, 0, b_{2h}, 0, b_{3h}, 0, b_{2h}\}$$

Using these the new A and C vectors, we again calculate $S_0...S_7$ with the same `msum` operations. In doing so we have now calculated all partial products for the first four rows of the parallelogram. It is clear that, in repeating this shift, permute, and `msum` cycle two more times, we will have calculated all partial products for the convolution of the two 128-bit multiplicands.

What remains is for us to take our eight result vectors $S_0...S_7$ and manipulate them to yield a single 256-bit result. To do this, we need to extract the sums of partial products from these vectors and add them in the appropriate bit position of the result. Beginning with the lower (least significant) half of the result, we have four result vectors $S_0...S_3$ that contain our sums. For example, the vector S_0 contains sums of partial products for four columns, but the vector elements are 32 bits wide, and need to overlap by 16 bits, since each element is the sum of partial products of 16-bit elements.

To accomplish this, we permute the elements of these result vectors so that they can be added in the correct bit positions. We begin by generating the vectors $T_0...T_3$ as unions of the S_{ij} (defined as the j-th component of vector S_i), as follows:

$$T_0 = S_{21}S_{23}S_{01}S_{03}$$

$$T_1 = S_{20}S_{22}S_{00}S_{02}$$

$$T_2 = S_{31}S_{33}S_{11}S_{13}$$

$$T_3 = S_{30}S_{32}S_{10}S_{12}$$

We then add these to our result vector R_0 like so:

$$R_0 = T_0 + (T_1 << 8) + (T_2 << 16) + (T_3 << 24).$$

The portions of the vectors that are shifted past the vector boundary are added to R_1, the upper 128 bits of our result. We again generate $T_0...T_3$, this time using $S_4...S_7$ in place of $S_0...S_3$, and add them to R_1:

$$R_1 = R_1 + T_0 + (T_1 << 8) + (T_2 << 16) + (T_3 << 24).$$

Having done this, we obtain two 128-bit vectors, R_0 and R_1, which together contain our 256-bit result.

4. Software routines—Release 1.0

After writing a library of Velocity Engine-specific functions, we modified an established (public domain) large-integer library to call those Velocity Engine-based functions. (The public domain code actually orginated in the 1980s at NeXT Software, Inc.). In this way we were able to port to Velocity Engine some pre-existing code for factoring and convolution.

Release 1.0 of this library includes the following source and header files:

```
factor.c
giantsdebug.h
giantsstack.c
```

```
giantsstack.h
vecarith.c
vecarith.h
vecarith.s
vgiants.c
vgiants.h
```

Also included is a Metrowerks project for building the factoring application (vfactor.mcp), and a document containing the required commands to build and execute the application using MPW and the MrC compiler (build.mpw). The Metrowerks project was built using a prerelease version of the CodeWarrior IDE (3.3.1b1) and C++ compiler (2.2.5b1). Minor project modifications may be necessary to build the project on subsequent CodeWarrior releases.

To build the MPW project, open the buildvfactor.mpw file, and execute the commands listed therein. (This process will require the source files specified above, as well as the libraries and object files in the MPW libs folder.) This will produce an MPW tool called "vfactor". Note that building with MrC requires a version that is Velocity Engine-aware. (This project was built with MrC compiler version 4.1a4c1.)

To run the factor program, either run the "factor app" application (produced by the CodeWarrior project) or type "vfactor" within MPW. Typing in a number and hitting enter will start the factoring application. Here is a sample transcript:

```
vfactor
342934934294239324934293249342934292921332112312
Sieving...
2 * 2 * 2 * 7 * 59 * 263 * 6863 *
Commencing Pollard rho...
.....................................................................
.....................................................................
.............
Commencing Pollard (p-1).............
37837124287441
*
Commencing ECM...
Choosing curve 1, with s = 1438858426, B = 1000, C = 50000:
..
Commencing second stage, curve 1...
....
Choosing curve 2, with s = 1438858427, B = 1000, C = 50000:
..
Commencing second stage, curve 2...
....
Choosing curve 3, with s = 1438858428, B = 1000, C = 50000:
..
Commencing second stage, curve 3...
```

```
....
20432578927
* 743807641141
```

This result shows a factoring of the number

$$342934934294239324934293249342934292 1332112312$$

into the factors

$$2 * 2 * 2 * 7 * 59 * 263 * 6863 * 37837124287441 * 20432578927 * 743807641141.$$

5. Test applications: factoring and large convolution

When we forged a factoring application, it was found that the overall performance fell roughly into this aforementioned range 3:1 to 10:1, more precisely 2:1 to 7:1. This slightly reduced range for the application is because, of course, a factoring implementation is only "mostly multiply-mod:, not 100 per cent so. Still, the performance enhancement was excellent. Here is one typical example, the factorization of the 620-bit number

$$N = 435108243740245651358922388150549211158687501789812601675356530 31$$

$$917099368240626644680118263857543017865140367343593561428407 0$$

$$336919814680301062476748028243020739149015352838320127815841 7$$

(into the factor 281474976726667 and a (500+)-bit prime), using an elliptic curve method (ECM) factoring program linked to our Velocity Engine multiprecision library. The entire run consumed 200 seconds on using the Velocity Engine, which in itself has no real meaning because ECM is statistical in nature. However a deterministically identical run on scalar (base-2^{16}) implementation of the same program consumed 1140 seconds, for a speedup of about 6. Incidentally, though a G3 hand-tuned assembly version of the factoring program was not available, it would be expected to yield a gain of perhaps 1.5 to 2, because again not all of the work is in multiply-mod. This means for the factoring application the Velocity Engine-over-assembler gain would still be about 3 or 4.

A second application program was written to exploit the fast Velocity Engine multiply in a (very) large convolution problem. Namely, the problem of determining the character (prime or composite) of the twenty-fourth Fermat number F_{24} requires convolutions of staggering length. We chose 512-bit convolution elements, so that the required convolution length was therefore $D = 2^{15} = 32768$. This length-32768 convolution (technically, what is called a negacyclic convolution because of the nature of Fermat arithmetic) of 512-bit elements was effected via the so-called Nussbaumer method. We were able to bring the convolution work down to a total time of about 10 seconds, making this the fastest known integer convolution (mod F_{24}). A research effort is currently underway, using Velocity Engine machinery to settle the F_{24} problem in rigorous fashion.

Acknowledgments

The authors are indebted to G. Miranker, D. Mitchell, A. Sazegari, and L. Slotnick for insight and support relevant to this research.

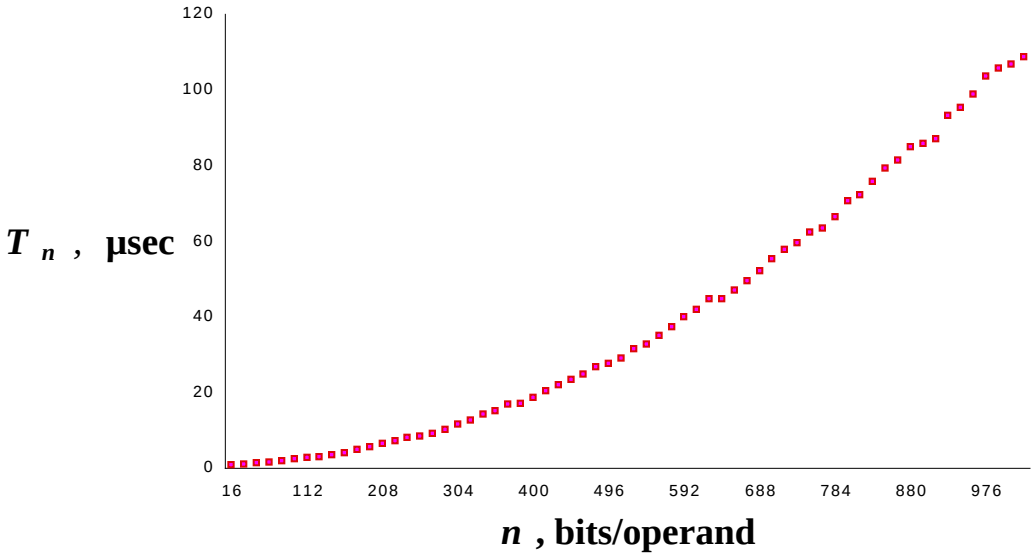

Figure 1: Standard multiply timings. The time Tn for base-2^{16} generic multiply (on Apple G3, 300MHz) is plotted against n, the bit size of each operand. So, for example, T_{256} = 8.4 µsec, T_{1024} = 108µsec. The timing is parabolic (i.e. as n^2).

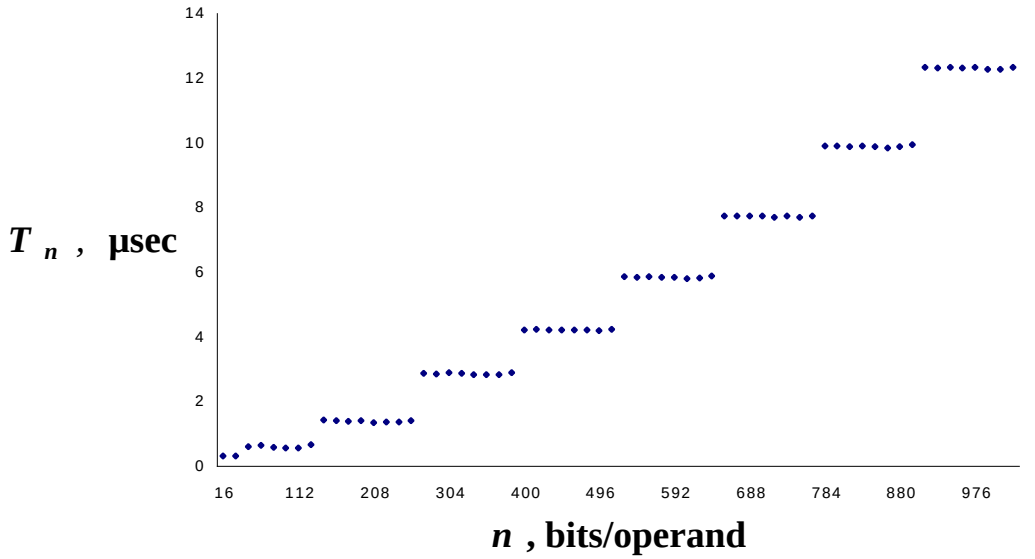

Figure 2: Altivec multiply timings. The time T_n is plotted against n, the operand bit size. Nartural, 128-bit boundaries are evident. Note the timing is not purely parabolic, rather has some linearity over this range of n, as explained in text.

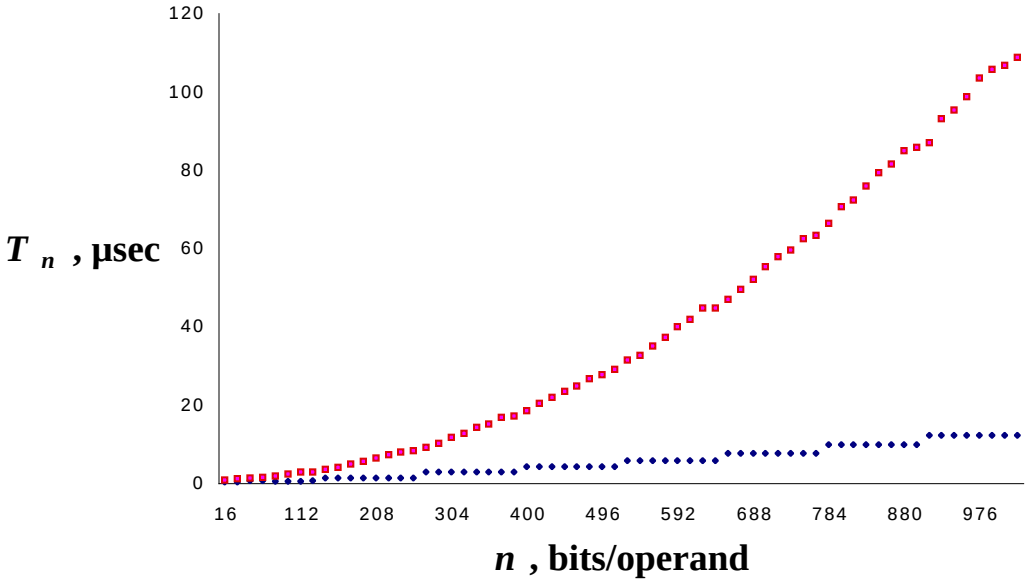

Figure 3: Comparison of generic base-2^{16} (upper curve) vs. Altivec (lower curve) multiply timings. Note the speedup factor is about 10:1 for the higher bit sizes n . Experiments showed that the factor 10:1 is essentially the asymptotic speedup.

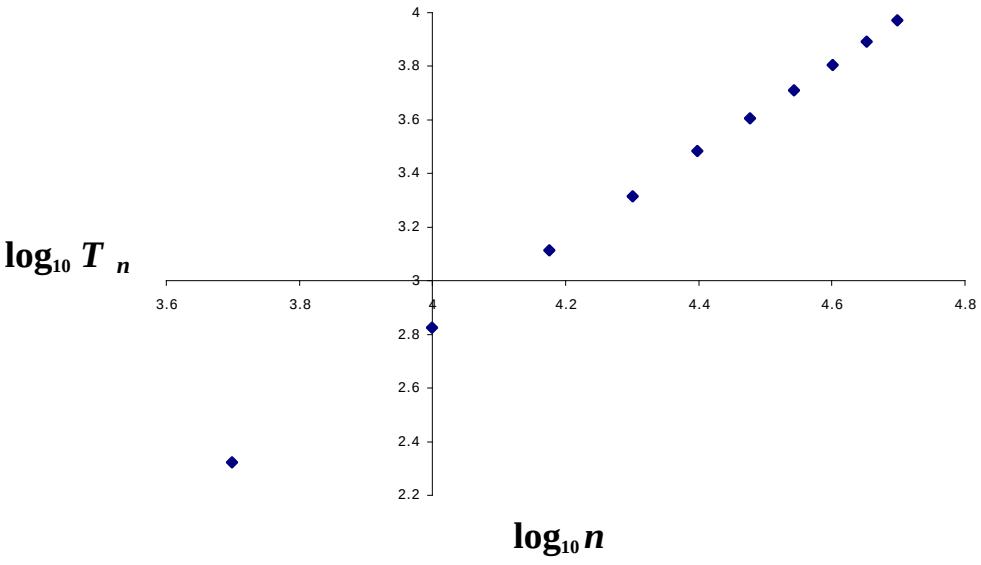

Figure 4: Altivec timings beyond the 3000-bit region. The log-log plot for Karatsuba multiply shows the theoretically expected growth of $T \sim n^{1.6}$. The plot shows, for example, that for 50000-bit operands the Altivec multiply time is about 10 millisec.