Fast Sine/Cosine Routine: Revenge of the Vector Processors

W. D. Cotton, January 17, 2014

***Abstract*—The calculation of sine/cosine pairs is a common and relatively expensive operation in radio interferometry. The very flexible “DFT” calculation of the interferometer response to a sky model makes heavy use of this operation and the cost of sine and cosine calculations can dominate the run time of a process using this technique. A previous memo described an approximate sine/cosine routine including a vector implementation based on the SSE extensions which achieved a substantial performance gain over using the c library routines. In this memo, the technique is extended to use the Advanced Vector Extensions (AVX) using the 256 bit memory bus. The ability to process 8 floats at a time combined with a more powerful instruction set leads to a performance enhancement of a factor of 3 over the SSE implementation and a factor of 10 over the c library sincosf routine.**

***Index Terms*—interferometry, performance**

I. Introduction

CALCULATING the response of an interferometer to a sky model is a common operation in radio interferometry and for complex sky models can be one of the more computationally expensive operations. One generic type of sky model calculation is the so called Direct Fourier Transform (AKA “DFT”) technique wherein the response to each component of a sky model (e.g. CLEAN component) is evaluated for each complex correlation in the data set. The real and imaginary parts of such model calculations are evaluated by sine and cosine functions of the component phase. This can result in VERY LARGE numbers of calls to sin and cos routines which dominate the cost of this operation.

A previous memo [1] explored an approximate vector sine/cosine routine based on the Intel Streaming SIMD Extensions technology (SSE) to obtain a substantial improvement over using the c library routines. SSE uses the 128 bit memory bus to operate on 4 values simultaneously. This memo extends this approach to the Intel Advanced Vector Extensions technology (AVX). AVX is an extension of SSE to 8 parallel operations on floats using the 256 bit memory bus and a stronger set of instructions.

II. Sines and Cosines

Sines and cosines are periodic functions that repeat every 2π radians of their argument. Thus, while the range of arguments to these functions is $\pm\infty$, they can be fully described by the interval $[0, 2\pi]$ and replacing the argument with its value modulo 2π. This facilitates a table look-up scheme.

National Radio Astronomy Observatory, 520 Edgemont Rd., Charlottesville, VA, 22903 USA email: bcotton@nrao.edu

While an arbitrary precision can be obtained by a simple table look-up, adequate resolution can require quite large tables. Further improvements in the precision of a table look-up can be obtained using either an interpolation in the table or a series expansion about the tabulated value. A Taylor’s series expansion is given by:

$$f(x) \ = \ f(a) \ + \ f'(a)\frac{x-a}{1!} \ + \ f''(a)\frac{(x-a)^2}{2!} + ...$$

Since the derivatives of sines and cosines are the cosines and sines of the same angles, evaluation of such a series is straightforward. The one term expansions for sine and cosines of angle x about tabulated value a are:

$$sin(x) \ = \ sin(a) \ + \ cos(a)[x-a]$$

$$cos(x) \ = \ cos(a) \ - \ sin(a)[x-a]$$

III. Implementation

A fast sine/cosine routine, ObitSinCosCalc, to calculate a sine/cosine pair was implemented in Obit utility module ObitSinCos using a 1381 element (1 1/4 turn) table look-up followed by the single term expansion given above. This routine will initialize the tables on the first call. Once initialized, this function should be thread-safe; a single call prior to initializing threading using this routine will make usage thread-safe.

A. Scalar

A single look-up table can be used for sine and cosine by noting that cos(phase) = sin(phase + 1/4 turn). Further reductions in the size of the table needed can be had by using the symmetries in the sine/cosine functions but at a cost of increased logic and computation. This implementation is unchanged from [1].

B. Vector

It is possible to go one step further. The overhead of function calls can be reduced by collecting the phases for which the sines and cosines are desired into an array and doing the calculations in a single function call. A “vector” version of ObitSinCosCalc is implemented in routine ObitSinCosVec. Organizing data into vectors also improves the cache hit ratio of the code.

C. AXV

Intel Advanced Vector Extensions technology (AVX) is an extension of the SIMD SSE architecture operating on length 8 vectors of floats (32 bit) using the 256 bit memory bus of current processors. In addition, the instruction set in AVX is more powerful than that in SSE allowing more of the processing to be done in parallel. In particular, AVX has load and store operations allowing the 256 bit registers to be loaded from, and copied to, contiguous blocks of memory.

The source code for the updated ObitSinCos package is given in the appendix. #ifdefs are used to select AVX if available and if not then SSE if available, and if that is not available a vanilla version in c is used. Note, in the current gcc compiler, the "-march native" argument is needed to enable AVX.

IV. Testing

The following give the results of various precision and timing tests.

A. Precision

In order to evaluate the precision of this technique the comparison of [1] between the approximate routine and the standard library sinf/cosf routines was repeated, a test program was employed that used 10^8 angles randomly spaced from -100 to 100 turns and compared the results of ObitSinCosCalc and ObitSinCosVec with the c library sinf and cosf routines. The average difference in this test was -1.6×10^{-7}, the rms difference was 7.3×10^{-6} and the maximum difference was 2.2×10^{-5}. When the test was restricted to the range -1 to +1 turn, the maximum error was 4.6×10^{-6}, the average error was 1.82×10^{-7} and the RMS error was 1.8×10^{-6}. The values of the sine and cosine range between -1.0 and +1.0.

B. Sin/cos Timing

Timing was based on the Threaded DFT time test program described in [2]. This test is dominated by the time to compute 1.3×10^{11}, sine/cosine pairs. The test was modified to use a variety of routines and implementations or none at all ("none" test) to determine the compute time of the remainder of the test. The test was run using one thread for each of the 16 cores on the target machine (smeagle Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz, cache size=20480 KB). Threading was based on gthreads thread pools. More details of the test program are given in [2].

The vector length used for the ObitSinCosVec tests was 128. Several runs were made in each test and the results averaged. The results are given in Table I and summarized in Figure 1. The "ratio" column in Table I and the values plotted in Figure 1 are the ratios of the test times minus the "none" time of the sinf/cosf results to the other tests.

V. Discussion

The cost of "DFT" interferometer model calculations is strongly dominated by the cost of calculating sines and cosines. The new sine/cosine AVX based routine described above appears to be >10 times faster than the c library sinf and cosf versions and over 3 times faster than the SSE implementation. The improvement of the AVX version over the SSE version is from a combination of the larger register size (8 v. 4 floats) and the stronger instruction set allowing more of the application to use the vector primitives. The AVX vector implementation is a great improvement over SSE although still not to the level of the glory days of vector processing of Crays and Convexes. The performance of the c library routines and the compiler appear to have improved since the tests reported in [1].

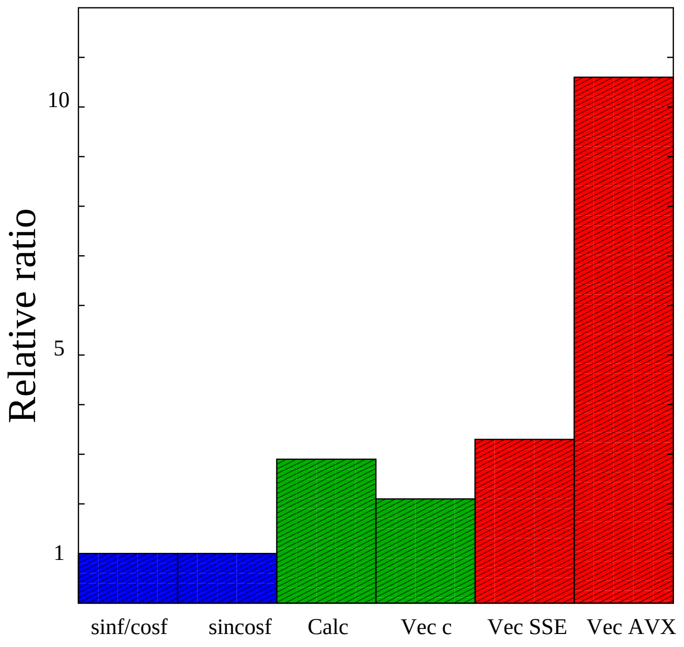

Fig. 1. The relatives speeds of various sine/cosine routines from the results in Table I. Bars in blue are c library routines, in green are c compiled version and in red are SSE or AVX intrinsics.

TABLE I
Sin/cos timings

method	total CPU time sec.	ratio
c sinf/cosf	239.26	1.0
c sincosf	237.33	1.0
ObitSinCosCalc	95.76	2.9
ObitSinCosVec c	122.80	2.1
ObitSinCosVec SSE	85.65	3.3
ObitSinCosVec AVX	39.84	10.6
none	19.44	

Appendix

Text of the Obit utility ObitSinCos.c follows.

```
/* Utility routine for fast sine/cosine calculation                   */
#include "ObitSinCos.h"
#include <math.h>

#define OBITSINCOSNTAB  1024  /* tabulated points per turn */
#define OBITSINCOSNTAB4 256   /* 1/4 size of table */
/** Is initialized? */
gboolean isInit = FALSE;
/** Sine lookup table covering 1 1/4 turn of phase */
gfloat sincostab[OBITSINCOSNTAB+OBITSINCOSNTAB4+1];
/** Angle spacing (radian) in table */
gfloat delta;
/** 1/2pi */
gfloat itwopi = 1.0/ (2.0 * G_PI);
/** 2pi */
gfloat twopi = (2.0 * G_PI);

/** AVX implementation 8 floats in parallel */
#if HAVE_AVX==1
#include <immintrin.h>

typedef __m256  v8sf;
typedef __m256i v4si;

/* gcc or icc */
# define ALIGN32_BEG
# define ALIGN32_END __attribute__((aligned(32)))

/* Union allowing c interface */
typedef ALIGN32_BEG union {
  float f[8];
  int   i[8];
  v8sf   v;
} ALIGN32_END V8SF;

/* Union allowing c interface */
typedef ALIGN32_BEG union {
  int  i[8];
  v4si      v;
} ALIGN32_END V4SI;

/* Constants */
#define _OBIT_TWOPI  (2.0 * G_PI)           /* 2pi */
#define _OBIT_ITWOPI 1.0/ (2.0 * G_PI)      /* 1/2pi */
#define _OBIT_DELTA  0.0061359231515425647  /* table spacing = 2pi/Obit_NTAB */
#define _OBIT_NTAB   OBITSINCOSNTAB         /* size of table -1 */
#define _OBIT_NTAB4  OBITSINCOSNTAB4
static const v8sf _half = {0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5}; /* 0.5 vector */
static const v8sf _ntab = { _OBIT_NTAB, _OBIT_NTAB, _OBIT_NTAB, _OBIT_NTAB,
    _OBIT_NTAB, _OBIT_NTAB, _OBIT_NTAB, _OBIT_NTAB};
static const v8sf _i2pi = { _OBIT_ITWOPI, _OBIT_ITWOPI, _OBIT_ITWOPI, _OBIT_ITWOPI,
    _OBIT_ITWOPI, _OBIT_ITWOPI, _OBIT_ITWOPI, _OBIT_ITWOPI};
static const v8sf _toopi= { _OBIT_TWOPI, _OBIT_TWOPI, _OBIT_TWOPI, _OBIT_TWOPI,
    _OBIT_TWOPI, _OBIT_TWOPI, _OBIT_TWOPI, _OBIT_TWOPI};
static const v8sf _dlta = { _OBIT_DELTA, _OBIT_DELTA, _OBIT_DELTA, _OBIT_DELTA,
    _OBIT_DELTA, _OBIT_DELTA, _OBIT_DELTA, _OBIT_DELTA};
```

```
/**
 * Fast AVX (8) vector sine/cosine of angle
 * Approximate sine/cosine, no range or value checking
 * \param angle  angle in radians
 * \param table  lookup table
 * \param s      [out] sine(angle)
 * \param c      [out] cosine(angle)
*/
void fast_sincos_ps(v8sf angle, float *table, v8sf *s, v8sf *c) {
  v8sf anglet, temp, ft, cell, it, sine, cosine, d;
  V4SI addr;

  /* get angle in turns */
  anglet = _mm256_mul_ps(angle, _i2pi);

  /* truncate to [0,1] turns */
  temp   = _mm256_floor_ps(anglet);        /* next lowest (signed) integer  */
  ft     = _mm256_sub_ps (anglet, temp);   /* Fractional turn */

  /* Table lookup, cos(phase) = sin(phase + 1/4 turn)*/
  it        = _mm256_mul_ps(ft, _ntab);    /* To cells in table */
  it        = _mm256_add_ps(it, _half);    /* add half */
  cell      = _mm256_floor_ps(it);         /* round to nearest cell */
  addr.v    = _mm256_cvtps_epi32(cell);    /* to integers */
  /* use union to load sine and cosine values */
  sine      = _mm256_set_ps(table[addr.i[7]],
    table[addr.i[6]],
    table[addr.i[5]],
    table[addr.i[4]],
    table[addr.i[3]],
    table[addr.i[2]],
    table[addr.i[1]],
    table[addr.i[0]]);
  cosine    = _mm256_set_ps(table[addr.i[7]+_OBIT_NTAB4],
    table[addr.i[6]+_OBIT_NTAB4],
    table[addr.i[5]+_OBIT_NTAB4],
    table[addr.i[4]+_OBIT_NTAB4],
    table[addr.i[3]+_OBIT_NTAB4],
    table[addr.i[2]+_OBIT_NTAB4],
    table[addr.i[1]+_OBIT_NTAB4],
    table[addr.i[0]+_OBIT_NTAB4]);

  /* One term Taylor series  */
  anglet = _mm256_mul_ps(ft, _toopi);       /* Now angle in radians [0,2 pi] */

  d      = _mm256_mul_ps (cell, _dlta);     /* tabulated phase = cell*delta_phase */
  d      = _mm256_sub_ps (anglet, d);       /* actual-tabulated phase */
  /* Cosine */
  temp   = _mm256_mul_ps (sine,d);
  *c     = _mm256_sub_ps (cosine, temp);
  /* Sine */
  temp   = _mm256_mul_ps (cosine,d);
  *s     = _mm256_add_ps (sine, temp);

  _mm_empty();  /* wait for operations to finish */
  return ;
} /* end AVX fast_sincos_ps */
```

```
/* end HAVE_AVX */

/** SSE implementation 4 floats in parallel */
#elif HAVE_SSE==1
#include <xmmintrin.h>

typedef __m128 v4sf;
typedef __m64 v2si;

/* gcc or icc */
# define ALIGN16_BEG
# define ALIGN16_END __attribute__((aligned(16)))

/* Union allowing c interface */
typedef ALIGN16_BEG union {
  float f[4];
  int   i[4];
  v4sf  v;
} ALIGN16_END V4SF;

/* Union allowing c interface */
typedef ALIGN16_BEG union {
  int   i[2];
  v2si  v;
} ALIGN16_END V2SI;

/* Constants */
#define _OBIT_TWOPI  6.2831853071795862     /* 2pi */
#define _OBIT_ITWOPI 0.15915494309189535    /* 1/2pi */
#define _OBIT_DELTA  0.0061359231515425647  /* table spacing = 2pi/Obit_NTAB */
#define _OBIT_NTAB   1024.0                 /* size of table -1 */
#define _OBIT_NTAB4  256
static const v4sf _half = {0.5, 0.5, 0.5, 0.5}; /* 0.5 vector */
static const v4sf _ntab = { _OBIT_NTAB, _OBIT_NTAB, _OBIT_NTAB, _OBIT_NTAB};
static const v4sf _i2pi = { _OBIT_ITWOPI, _OBIT_ITWOPI, _OBIT_ITWOPI, _OBIT_ITWOPI};
static const v4sf _toopi= { _OBIT_TWOPI, _OBIT_TWOPI, _OBIT_TWOPI, _OBIT_TWOPI};
static const v4sf _dlta = { _OBIT_DELTA, _OBIT_DELTA, _OBIT_DELTA, _OBIT_DELTA};

/**
 * Fast SSE (4) vector sine/cosine of angle
 * Approximate sine/cosine, no range or value checking
 * \param angle  angle in radians
 * \param table  lookup table
 * \param s      [out] sine(angle)
 * \param c      [out] cosine(angle)
*/
void fast_sincos_ps(v4sf angle, float *table, v4sf *s, v4sf *c) {
  v4sf anglet, temp, it, zero, mask, one, sine, cosine, d;
  v2si itLo, itHi;
  V2SI iaddrLo, iaddrHi;

  /* angle in turns */
  anglet = _mm_mul_ps(angle, _i2pi);

  /* truncate to [0,1] turns */
  /* Get full turns */
  itLo   = _mm_cvttps_pi32 (anglet);       /* first two truncated */
  temp   = _mm_movehl_ps (anglet,anglet);  /* upper two values into lower */
  itHi   = _mm_cvttps_pi32 (temp);         /* second two truncated */
  it     = _mm_cvtpi32_ps (temp, itHi);    /* float upper values */
  temp   = _mm_movelh_ps (it, it);         /* swap */
  it     = _mm_cvtpi32_ps (temp, itLo);    /* float lower values */

  /* If anglet negative, decrement it */
  zero   = _mm_setzero_ps ();              /* Zeros */
  mask   = _mm_cmplt_ps (anglet,zero);     /* Comparison to mask */
  one    = _mm_set_ps1 (1.0);              /* ones */
  one    = _mm_and_ps(one, mask);          /* mask out positive values */
  it     = _mm_sub_ps (it, one);
  anglet = _mm_sub_ps (anglet, it);        /* fold to [0,2pi] */

  /* Table lookup, cos(phase) = sin(phase + 1/4 turn)*/
  it        = _mm_mul_ps(anglet, _ntab);           /* To cells in table */
  it        = _mm_add_ps(it, _half);               /* To cells in table */
  iaddrLo.v = _mm_cvttps_pi32 (it);
  temp      = _mm_movehl_ps (it,it);               /* Round */
  iaddrHi.v = _mm_cvttps_pi32 (temp);
  sine      = _mm_setr_ps(table[iaddrLo.i[0]],table[iaddrLo.i[1]],
  table[iaddrHi.i[0]],table[iaddrHi.i[1]]);
  cosine    = _mm_setr_ps(table[iaddrLo.i[0]+_OBIT_NTAB4],
  table[iaddrLo.i[1]+_OBIT_NTAB4],
  table[iaddrHi.i[0]+_OBIT_NTAB4],
  table[iaddrHi.i[1]+_OBIT_NTAB4]);

  /* One term Taylor series  */
  anglet = _mm_mul_ps(anglet, _toopi);          /* Now angle in radians */
  temp   = _mm_cvtpi32_ps (it, iaddrHi.v);      /* float upper values */
  it     = _mm_movelh_ps (temp,temp);           /* swap */
  it     = _mm_cvtpi32_ps (it, iaddrLo.v);      /* float lower values */
  d      = _mm_mul_ps (it, _dlta);              /* tabulated phase */
  d      = _mm_sub_ps (anglet, d);              /* actual-tabulated phase */
  /* Cosine */
  temp   = _mm_mul_ps (sine,d);
  *c     = _mm_sub_ps (cosine, temp);
  /* Sine */
  temp   = _mm_mul_ps (cosine,d);
  *s     = _mm_add_ps (sine, temp);

  _mm_empty();  /* wait for operations to finish */
  return ;
} /* end AVX fast_sincos_ps */

#endif  /* HAVE_AVX */

/**
 * Initialization
 */
 void ObitSinCosInit(void)
{
  glong i;
  float angle;

  isInit = TRUE;  /* Now initialized */
  delta = ((2.0 *G_PI)/OBITSINCOSNTAB);

  for (i=0; i<(OBITSINCOSNTAB+OBITSINCOSNTAB4+1); i++) {
    angle = delta * i;
    sincostab[i] = (gfloat)sinf(angle);
  }
} /* end ObitSinCosInit */

/**
 * Calculate sine/cosine of angle
 * Lookup table initialized on first call
 * \param angle  angle in radians
 * \param sin    [out] sine(angle)
 * \param cos    [out] cosine(angle)
*/
void ObitSinCosCalc(gfloat angle, gfloat *sin, gfloat *cos)
{
  glong it, itt;
  gfloat anglet, ss, cc, d;

  /* Initialize? */
  if (!isInit) ObitSinCosInit();

  /* angle in turns */
  anglet = angle*itwopi;

  /* truncate to [0,1] turns */
  it = (glong)anglet;
  if (anglet<0.0) it--;   /* fold to positive */
  anglet -= it;

  /* Lookup, cos(phase) = sin(phase + 1/4 turn) */
  itt = (glong)(0.5 + anglet*OBITSINCOSNTAB);
  ss  = sincostab[itt];
  cc  = sincostab[itt+OBITSINCOSNTAB4];

  /* One term Taylor series  */
  d = anglet*twopi - delta*itt;
  *sin = ss + cc * d;
  *cos = cc - ss * d;
} /* end ObitSinCosCalc */

/**
 * Calculate sine/cosine of vector of angles, uses AVX or SSE implementation if available
 * Lookup table initialized on first call
 * \param n      Number of elements to process
 * \param angle  array of angles in radians
 * \param sin    [out] sine(angle)
 * \param cos    [out] cosine(angle)
*/
void ObitSinCosVec(glong n, gfloat *angle, gfloat *sin, gfloat *cos)
{
  glong i, nleft, it, itt;
  gfloat anglet, ss, cc, d;
  /** SSE implementation */
#if   HAVE_AVX==1
  glong ndo;
  v8sf vanglet, vss, vcc;
#elif HAVE_SSE==1
  glong ndo;
  V4SF vanglet, vss, vcc;
```

```
#endif /* HAVE_SSE */

  /* Initialize? */
  if (!isInit) ObitSinCosInit();

  nleft = n;   /* Number left to do */
  i     = 0;   /* None done yet */

 /** avx implementation */
#if HAVE_AVX==1
  /* Loop in groups of 8 */
  ndo = nleft - nleft%8;  /* Only full groups of 8 */
  for (i=0; i<ndo; i+=8) {
    vanglet = _mm256_loadu_ps(angle); angle += 8;

    fast_sincos_ps(vanglet, sincostab, &vss, &vcc);
    _mm256_storeu_ps(sin, vss); sin += 8;
    _mm256_storeu_ps(cos, vcc); cos += 8;
 } /* end AVX loop */
 /** SSE implementation */
#elif HAVE_SSE==1
  /* Loop in groups of 4 */
  ndo = nleft - nleft%4;  /* Only full groups of 4 */
  for (i=0; i<ndo; i+=4) {
    vanglet.f[0] = *angle++;
    vanglet.f[1] = *angle++;
    vanglet.f[2] = *angle++;
    vanglet.f[3] = *angle++;
    fast_sincos_ps(vanglet.v, sincostab, &vss.v, &vcc.v);
    *sin++ = vss.f[0];
    *sin++ = vss.f[1];
    *sin++ = vss.f[2];
    *sin++ = vss.f[3];
    *cos++ = vcc.f[0];
    *cos++ = vcc.f[1];
    *cos++ = vcc.f[2];
    *cos++ = vcc.f[3];
  } /* end SSE loop */
#endif /* HAVE_SSE */

  nleft = n-i;  /* How many left? */

 /* Loop doing any elements not done in AVX/SSE loop */
  for (i=0; i<nleft; i++) {
    /* angle in turns */
    anglet = (*angle++)*itwopi;

    /* truncate to [0,1] turns */
    it = (glong)anglet;
    if (anglet<0.0) it--;   /* fold to positive */
    anglet -= it;

    /* Lookup, cos(phase) = sin(phase + 1/4 turn) */
    itt = (glong)(0.5 + anglet*OBITSINCOSNTAB);
    ss  = sincostab[itt];
    cc  = sincostab[itt+OBITSINCOSNTAB4];

    /* One term Taylor series  */
    d = anglet*twopi - delta*itt;
    *sin++ = ss + cc * d;
    *cos++ = cc - ss * d;
  } /* end loop over vector */
} /* end ObitSinCosVec */
```

References

[1] W. D. Cotton, “A Fast Sine/Cosine Routine,” *Obit Development Memo Series*, vol. 14, pp. 1–9, 2009.

[2] ——, “Comparison of gpu and multithreading for interferometric dft model calculation,” *Obit Development Memo Series*, vol. 35, pp. 1–5, 2013.