AVX2: First Look

W. D. Cotton, August 7, 2017

***Abstract*—Some aspects of the second version of the "Advanced Vector eXtensions" (AVX2) are examined in the context of radio interferometric imaging. These are basically a strengthening of the vector instruction set. The newly included integer instructions are a big win in a public domain trigonometric vector library but the "gather" function is substantially slower than its more primitive counterpart. Two version of gcc are also compared.**

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

I. Introduction

MODERN CPUs can use operands the width of the memory bus. When the memory bus is wider than the size of a given data type, this can be used to perform parallel operations or SIMD = "Single Instruction, multiple data" essentially forming a vector engine. There were several incarnations of the SSE instructions for 128 bit buses (4 floats or 2 doubles) and the initial SIMD operations for 256 bit buses (8 float, 4 double) was "AVX" ("Advanced Vector eXtensions"). AXV2 is an enhanced version of AVX; the improvements of interest here are a stronger set of integer operations and a "gather" function to fill a vector register with data from non contiguous locations in memory. The functionality can be accessed via 1) an optimizing compiler, 2) assembly instructions or 3) "intrinsics" which are c function calls corresponding to an assembly instruction. The expensive Intel compiler is good at implementing these features but the freebe gcc lags substantially. The testing reported here used the "intrinsics".

Vectors of sine/cosine pairs are frequently used in interferometry, especially in the "DFT" approximation of a Fourier Transform. Public domain libraries using SSE/AVX/AVX2 for trigonometric functions are available (https://github.com/juj/MathGeoLib/blob/master/src/Math/sse_mathfun.h, https://github.com/reyoung/avx_mathfun). This memo evaluates using these libraries in the Obit package [1] [1]. This analysis also compares gcc-4.4.7 and gcc-4.9.1.

II. AVX2 Features

The enhancements of interest here are the improved integer support and the gather function. The new integer features are supported by the avx_mathfun.h library of trigonometric functions. Testing used the sincos function in both a simple standalone mode and as implemented in the Obit interferometry software. The gather function was evaluated in a standalone test.

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

[1]http://www.cv.nrao.edu/∼bcotton/Obit.html

TABLE I
Sin/cos timings

method	total Run time sec.	CPU time sec.
SSE	4.19	4.19
AVX	3.91	3.91
AVX2	1.75	1.75

III. Standalone Tests

These tests consist of simple c programs performing a function a large number of times and using the unix time utility to determine the run and cpu times. The tests were run on a laptop with two Intel i5-4300M cores running at 2.6 GHz and which supports SSE, AVX and AVX2 instructions. Compilation used gcc-4.9.1 with O3 optimization.

A. Sine/Cosine Tests

Usage of sines and cosines in interferometry is relatively simple and only accurate values are needed. The properties of continuity, differentiability etc. provided by the standard c libraries are overkill and the simplified version provided by (sse)avx_mathfun.h is quite adequate. The test program is given in Figure III-A.

The test program was compiled with a number of options using gcc-4.9.1. All options were tested comparing with the system library sincosf function and then a number of test timing runs were made without this comparison and the timings averaged. All comparisons gave an average difference of 5.78705e-16, and RMS difference of 2.36859e-11 a maximum error of 5.96046e-08 and sum of the sine values of 5.78705e-16. Timings are given in Table I. The AVX rate was only 7% faster than SSE while the AVX2 time is 2.4 times faster than SSE and 2.2 times faster than AVX. The poor performance of AVX appears to be the result of having to use SSE integer functions with more shuffling data around between 256 and 128 bit variables.

B. Gather Tests

The tests of the AVX2 "gather" function used the program in Figure III-B. This was compiled with either the "Manual" vector load or the "Gather" vector load sections commented out and the execution timed. The loaded vector was "stored" to keep the operation from being optimized away. Results are in Table II. While puzzling, these results appear correct; the gather function seems like a bad idea.

```
#include "ObitSinCos.h"
void sincosf(float x, float *sin, float *cos);
int main ( int argc, char **argv )
{
  ofloat sss,ccc, d, dmax;
  ofloat phase, randnorm, fazRange;
  ofloat a[1000], c[1000], s[1000];
  odouble ssum,sum, sum2, count, rms;
  olong i, j, m, n;
  gboolean doCompare=FALSE;/* Compare w/ library */
  n = 900000000;
  m = 1000;
  /*fact = 2 * 3.1415926 * 5 / (ofloat)n;*/
  sum = sum2 = count = 0.0;
  dmax = 0.0;
  fazRange = 100.0;
  randnorm = fazRange / RAND_MAX;
  /* Fill array */
  for (j=0; j<m; j++) {
    /* Random phase between fazMin, fazMax */
    phase = (rand()-RAND_MAX/2)*randnorm;
    a[j] = phase;
  }
  ssum = 0.0;
  for (i=0; i<n; i+=m) {
    /* Compute vector  */
    ObitSinCosVec (m, a, s, c);
    ssum += s[0]; /* Don't optimize away */

    /* Compare  */
    if (doCompare) {
      for (j=0; j<m; j++) {
        sincosf(a[j], &sss, &ccc);
        d = (sss-s[j]); dmax = MAX (dmax, fabs(d));
        sum += d; sum2 += d*d; count++;
        d = (ccc-c[j]); dmax = MAX (dmax, fabs(d));
        sum += d; sum2 += d*d; count++;
      }
      sum /= count; sum2 /= count;
      rms = sqrt (sum2 - sum*sum);
    }
  }
  if (doCompare)
    fprintf (stdout,"Avg difference %lg rms %lg max err=%g\n",
      sum, rms,dmax);
   fprintf (stdout,"sum=%g\n", ssum);
  return 0;
} /* end main */
```

Fig. 1.   Sine/Cosine Test Program

```
#include <immintrin.h>
typedef __m256  v8sf; // vector of 8 float (avx)
typedef __m256i v8si; // vector of 8 int   (avx)
int main ( int argc, char **argv )
{
  float a[8000], dump[800];
  long i, j, m, n;
   /* reordering - Multiply x 10 to keep in separate 256 bit mem reads*/
  long re[8] = {5*10,2*10,1*10,3*10,6*10,7*10,0*10,4*10};
  v8si order;
  v8sf vector, vector2;
  n = 9000000000;  m = 800;
  order = _mm256_set_epi32(re[0], re[1], re[2],re[3], re[4], re[5], re[6], re[7]);
  /* Fill array with random numbers */
  for (j=0; j<m*10; j++) {
    /* Random vaue between fazMin, fazMax */
    a[j] = (rand()-RAND_MAX/2)*100. / RAND_MAX;
  }
  /* loop loading in reverse order */
  for (i=0; i<n; i+=m) {
    for (j=0; j<m; j+=8) {
      /* Manual
      vector = _mm256_set_ps(a[j+re[0]], a[j+re[1]], a[j+re[2]], a[j+re[3]],
                             a[j+re[4]], a[j+re[5]], a[j+re[6]], a[j+re[7]]);
      _mm256_storeu_ps(&dump[j], vector);*/
      /* Gather */
      vector2 = _mm256_i32gather_ps(&a[j], order, 4);
      _mm256_storeu_ps(&dump[j], vector2);
    } /* end inner loop */
  } /* end outer loop */
  return 0;
} /* end main */
```

Fig. 2. Gather Test Program

TABLE II
GATHER TIMINGS

method	total Run time sec.	CPU time sec.
manual	2.77	2.77
gather	5.17	5.17

C. Interferometry Tests

Interferometry tests were performed in Obit using various combination of compiler and AVX options on the same laptop as used for the standalone tests. This machine has an SSD disk which was used for all files and 8 GByte of memory. All programs were allowed to use two threads for multi-threaded operations. Other than compiler options, the only change to the Obit was the avx_mathfun library enabling use of AVX2. Obit implements segments of code using AVX (or AVX2) using #ifdef statements allowing SSE to be used when AVX(2) is not available (SSE has been around forever). Compiling Obit software enabling AVX needs compiler options “-DHAVE_AVX=1 -mavx” and for AVX2, “-DHAVE_AVX2=1 -mavx2”.

In principle, the compiler could also use AVX2 (or AVX) functions in its optimization when this was turned on. Comparing execution times in programs that do not explicitly use AVX with different version of the compiler will test for this.

D. UVSub

Task UVSub subtracts the Fourier transform of a CLEAN model from a visibility data set. If this is done using the Cmethod=’DFT” method, the processing for a large CLEAN model is dominated by the calculation of sin/cos pairs and is a good test of the avx_mathfun library. The VLA data set used had 18,122 visibilities each with 1024 channels and dual polarization. The CLEAN model subtracted had 900 CLEAN components. Each compiler/AVX option was run several times and the average results shown in Table III. As expected from Section III-A, the AVX2 results are substantially better than for either of the AVX results, 50% faster than the gcc-4.4.7 time and 30% faster than gcc-4.9.1. The gcc-4.9.1 compiler optimizer resulted in a 15% improvement over gcc-4.4.7. The improvement is probably NOT in the avx_mathfun library as

TABLE III
UVSUB TIMINGS

method	REAL time sec.	CPU time sec.
AVX gcc-4.4.7	96.3	188.9
AVX gcc-4.9.1	84.1	165.8
AVX2 gcc-4.9.1	64.8	126.8

TABLE IV
MFIMAGE TIMINGS

method	REAL time sec.	CPU time sec.
AVX gcc-4.4.7	285.3	390.5
AVX gcc-4.9.1	218.0	282.4
AVX2 gcc-4.9.1	226.7	275.4

the timing for Section III-A showed no significant difference for the two compilers.

E. MFImage

The wide-band imager MFImage was also tested with different AVX/compiler options. This used the visibility data set used in the UVSub test and created a 3000x3000x11 image using seven facets. CLEANing proceeded for 500 CLEAN components; results for the various tests were essentially identical. This program has extensive AVX sections especially for gridding and UVSub like model calculations. Average execution times are given in Table IV.

In this test, the fraction of the time spent calculating sin/cos pairs is relatively small and the AVX2 timing is a bit (4% but outside the scatter) worse than AVX only using the same compiler. However, the gcc-4.9.1 compiled AVX version ran 31% faster than the gcc-4.4.7 version.

F. HGeom

HGeom will interpolate an image onto the celestial grid defined by another image. It is compute intensive and multi-threaded but has no explicit AVX component. Timing tests used a 3000x3000x11 image and a 5×5 convolution kernel in a Lagrangian interpolation. Average execution times are given in Table V. There were no significant differences among the various tests.

TABLE V
HGEOM TIMINGS

method	REAL time sec.	CPU time sec.
AVX gcc-4.4.7	26.5	51.7
AVX gcc-4.9.1	26.4	51.1
AVX2 gcc-4.9.1	26.0	50.6

IV. DISCUSSION

Tests were presented comparing the use of new AVX2 features with AVX. The new integer operations make a big difference in the sine/cosine calculation from the avx_mathfun library. This difference is largely due to the poorer than expected performance using only AVX. The addition of the “gather” functionality in AVX2 is very curious given that it runs at half the speed of the more “primitive” alternative.

Implementation of the AVX2 features in the avx_mathfun library into the Obit package was shown to make a sizable impact on operations dominated by the DFT Fourier transform of CLEAN models such as UVSub. In the case of the AVX intensive interferometric imager MFImage, AVX2 made it marginally slower whereas switching from gcc-4.4.7 to gcc-4.9.1 decreased the run time by 30%. The CPU intensive but AVX impoverished program HGeom showed no timing effects from either AVX/AVX2 or which compiler was used.

ACKNOWLEDGMENT

I would like to thank Tracy Halstead for help with various compiler issues.

REFERENCES

[1] W. D. Cotton, “Obit: A Development Environment for Astronomical Algorithms,” *PASP*, vol. 120, pp. 439–448, 2008.