Comparison of GPU and Multithreading for Interferometric DFT Model Calculation

W. D. Cotton, version 2, January 17, 2014

***Abstract*—Calculation of sky models for interferometric image processing using a direct Fourier transform (DFT) allows a relatively arbitrary model to be computed. This flexibility comes at the price of a potentially large computing cost as the response to each sky model component must be computed for each visibility measurement. Fortunately, such calculations can be highly parallel and a multi–threading method has been implemented in Obit. This memo compares the performance of a multi–threaded implementation with that using a GPU. A substantial improvement (33X) is seen on large problems from using a GPU over multi–threading plus AVX and both implementations are far faster than a CPU single thread.**

***Index Terms*—interferometry, computation efficiency, multi–threading, GPU**

I. Introduction

CALCULATION of sky models for interferometric image processing using a direct Fourier transform (DFT) allows a relatively arbitrary model to be computed enabling a large variety of corrections. This flexibility comes with a potentially large computing cost as the response to each sky model component must be computed for each visibility measurement. This version of the memo uses the __sincosf function in the CUDA routine rather than sincosf which leads to a substantial improvement as well as the use of AVX intrinsics for the sine/cosine calculation in the threaded CPU case.

II. DFT Sky Model Calculation

The model of the celestial emission is frequently composed of discrete components derived from a CLEAN deconvolution. These may be a collection of either delta functions or extended components such as Gaussians. The following will consider the simple case of a set (k) of delta functions described by a flux density (a_k) and a position (x_k, y_k and z_k=$1-\sqrt{x_k^2+y_k^2}$). Each visibility measurement (V_j) is a complex value and is described by its location in the aperture plane as (u_{j0}, v_{j0}, w_{j0}) which is a function of frequency (ν) and the 0 subscripts indicate the (u, v, w) coordinate at the reference frequency ν_0. For multi–frequency data, the (u, v, w) coordinates at a given frequency are given by:

$$\begin{aligned} u_{j\nu} &= u_{j0}\ \frac{\nu}{\nu_0} \\ v_{j\nu} &= v_{j0}\ \frac{\nu}{\nu_0} \\ w_{j\nu} &= w_{j0}\ \frac{\nu}{\nu_0} \end{aligned}$$

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

The response of an interferometer to a sky model composed of n flat spectrum delta functions is then:

$$V_{i,\nu}\ = \sum_{k=0}^{n}\ a_k\ e^{-2i\pi \frac{\nu}{\nu_0}(u_{j0}x_k+v_{j0}y_k+w_{j0}z_k)} \qquad (1)$$

where i = $\sqrt{-1}$.

Evaluation of the complex exponential in Eq. 1 is done by taking the sine and cosine of the argument. A fast implementation of sine and cosine functions is described in [1].

III. Parallel Implementations

Modern large data-sets contain many (millions) of visibility records each with many frequency channels. The evaluation of Eq. 1 can be performed independently for visibility and frequency and is therefore “embarrassingly parallel”. Two current schemes for parallel computing are using multiple cores in a CPU via multi–threading and the using attached graphics processors generically called GPUs.

Large data-sets can be larger than will fit into computer memory so the technique of “strip mining”, processing sequential segments of the data, is frequently employed. Each segment of data can be split in any of a number of ways and the different parts processed in parallel.

A. Multi–threading

Modern CPUs contain multiple cores with a common address space; multiple cores can be used simultaneously by means of multiple threads of execution. A means of creating and controlling multiple threads in an application program is described in [2].

The implementation of multi–threaded DFT model calculation described here splits the data into groups of visibilities each of which is processed by a separate core. A sample thread routine is given in the appendix which uses the fast Sine/Cosine routine described in [1]. This version of the sin/cos vector routines are based on AVX intrinsics.

B. GPU

GPUs are attached processors with separate memories and address spaces; data must be copied to and from the device from the host. GPUs are optimized for graphics operations and are not as flexible as CPU cores but have sufficient capability for our needs, in particular, 32 bit floating arithmetic and math functions. A sample GPU kernel in CUDA is given in the appendix. GPUs contain a large number of processors similar to CPU cores but generally slower. In the implementation given in the appendix, the problem is split by visibility and frequency channel and each kernel evaluates Eq. 1 for a given visibility and frequency. Higher performance GPUs allow overlapping data transfers and execution which enhances the overall performance. Note: the CUDA code executes substantially faster using the __sincosf intrinsic function rather than sincosf.

IV. Timing Tests

A set of tests were performed on a workstation with both multiple cores and several GPU devices. The CPU contained 6 Intel(R) Xeon(R) CPU E5-1650 3.2 GHz cores hyper-threaded, a Tesla K20c GPU with 13 MPs x 192 (Cores/MP) = 2496 (Cores), and a GeForce GTX 780 GPU with 12 MPs x 192 (Cores/MP) = 2304 (Cores). The GTX 780 gave better performance and was used for the tests reported here.

Two implements were compared, one based on multi-threading using all 12 hyper threads with a fast sine/cosine routine based on AVX intrinsics and the other used the GTX 780 GPU. The threading version used the c language and the GPU, CUDA. Each was performed on the same simulated data-set consisting of 10,000 visibilities each with 512 frequency channels. Sky models with various numbers of point components were tested. Each test was repeated 100 times and the wall clock execution times measured with the UNIX time utility. The computation routines used are given in the appendix.

The GPU implementation used 4 streams to overlap data transfer and execution. The multi-threaded implementation used the AVX based fast sine/cosine routines from [1].

Each implementation was run using sky models of a varying number of components and the results given in Table I. The time as a function of number of components is plotted in Figure 1 and the ratio of the run times in Figure 2. For reference, a single threaded run of the 1024 component test took 2198 sec or, a factor of 6.0 longer than the multi–threaded test or 190 times longer than the GPU test. The speed of the threaded version scales with the number of CPU cores; in this case 6.

V. Discussion

The results given in Table I and Figures 1 and 2 show that for a small number of sky model components, the data transfer dominated the time used in the GPU implementation; there is little increase in the run time for up to 128 components. The speed of the GPU is still sufficient that it beats the multi–threaded implementation for all but the smallest test. The ratio of multi–threaded to GPU times increases rapidly up to about 512 components at which point the GPU computation time becomes substantially larger than the data transfer time. The ratio peaks at 33 at 4096 components and declines slowly for higher numbers. This decline in the ratio is likely due to the increasing efficiency of the multi–threaded implementation; for larger problems the scalar overhead of multiple threads is increasingly hidden behind the parallel computing. Both the multi–threaded and GPU implementations showed large performance enhancements over a single thread test.

This test suggests there are big gains to be had, factors of 10s improvement in performance using GPUs in radio interferometry imaging software. Other compute intensive operations such as gridding the data can also be subjected to similar treatment.

Acknowledgment

I would like to thank Scott Ransom for advice on GPU usage and for the use of his workstation for testing.

TABLE I
Scaling with Sky Model size

Number of Comps	Thread s	GPU s	Ratio
1	1.6	2.48	0.6
4	3.6	2.52	1.4
8	3.7	2.44	1.5
16	6.4	2.50	2.6
32	11.7	2.45	4.8
64	23.5	2.51	9.4
128	45.9	2.44	19
256	91.6	3.40	27
512	183.1	6.04	30
1024	365.3	11.58	32
2048	729.4	22.54	32
4096	1458	44.55	33
8192	2912	98.56	30

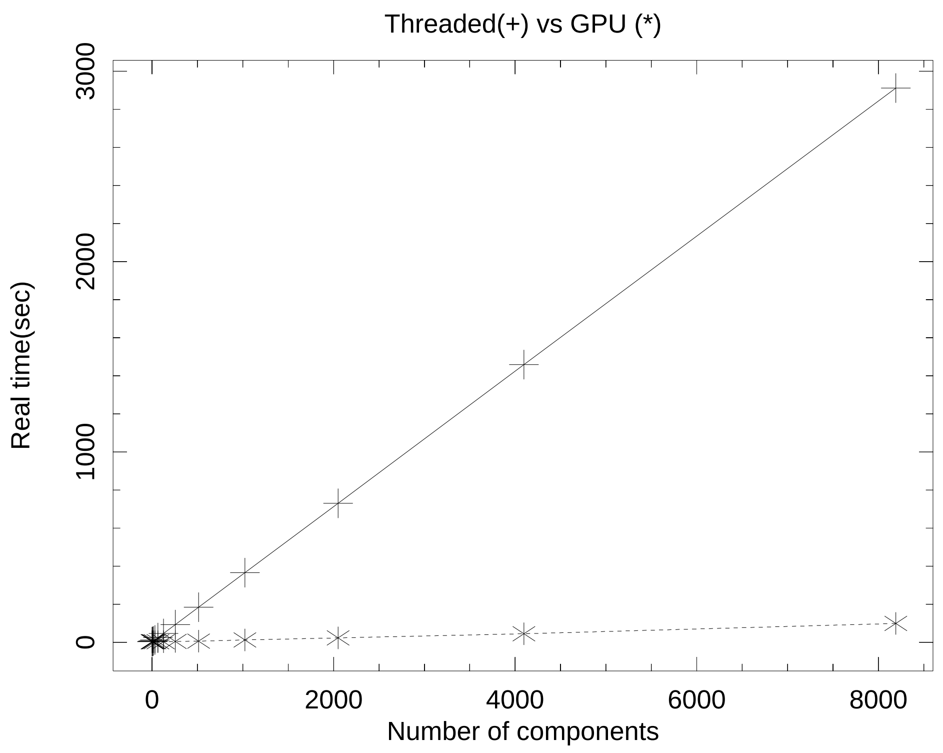

Fig. 1. Plot of real time v. number of CLEAN components in sky model for the threaded and GPU implementations. Thread values are shown as “+” and solid line and GPU values are “*” and a dotted line.

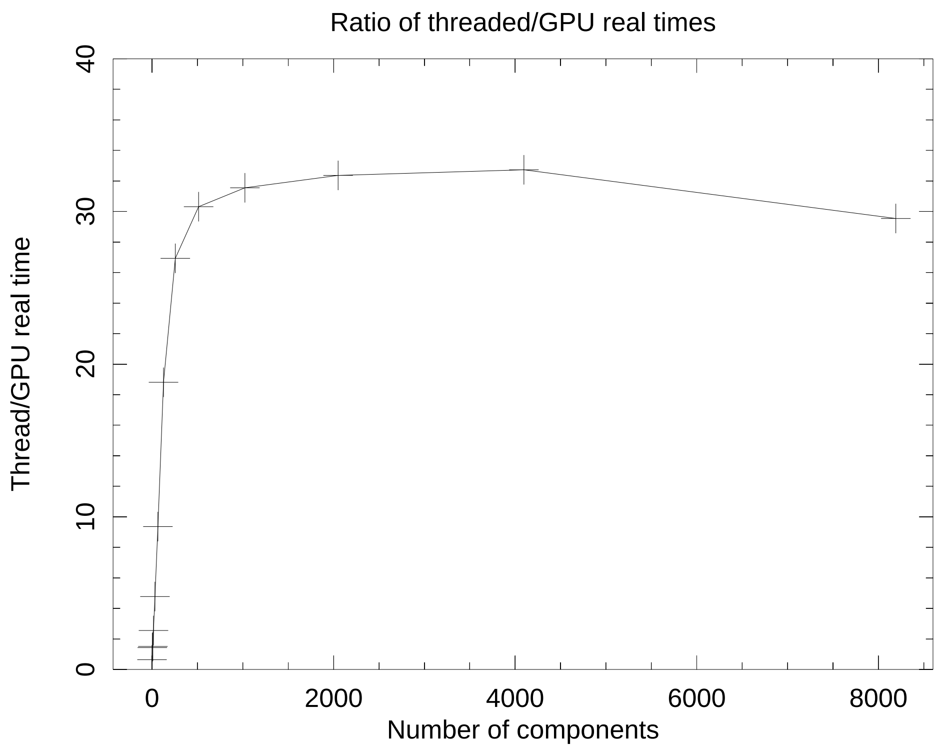

Fig. 2. Plot of the ratio of real times used by the threaded and GPU (GTX 780) processing as a function of the number of components in the CLEAN sky model.

Appendix

GPU code

The following is the source code for the kernel used in the GPU test.

```
__global__ void dftKernel(float *g_out,
  float *g_in,float *Model,float *FreqArr,
  int nModel, int nrparm, int lenvis)
{
    // beginning of a visibility
    int idx   = blockIdx.x * lenvis;
    // channel numbe
    int ichan = threadIdx.x;
    int i, ivis, iMod = 0;
    float arg, amp, s, c, sumR,sumI,u,v,w;
    float u, v, w;
    float freqFact = FreqArr[ichan];
    // copy random parms if 1st channel
    if (ichan==0) {
       for (i=0; i<nrparm; i++)
           g_out[idx+i] = g_in[idx+i];
    }
    // real part of vis
    ivis = idx + nrparm + ichan*3;

    // Assume random parms start with u,v,w,
    // model = flux, x,y,z factors
    u = g_in[idx];
    v = g_in[idx+1];
    w = g_in[idx+2];
    u *= freqFact;
    v *= freqFact;
    w *= freqFact;
    sumR = sumI = 0.0;
    for (int i=0; i<nModel; i++) {
        amp = Model[iMod];
        arg = u*Model[iMod+1] +
              v*Model[iMod+2] +
              w*Model[iMod+3];
        __sincosf(arg, &s, &c);
        sumR += amp * c;
        sumI += amp * s;
        iMod += 4;
    } // end loop over model comps
    g_out[ivis]   = g_in[ivis]   - sumR;
    g_out[ivis+1] = g_in[ivis+1] - sumI;
    g_out[ivis+2] = g_in[ivis+2];
} // end dftKernel
```

Thread code

The following is the source code executed in each thread of the multi-threaded version.

```
/**
 * Do Fourier transform using a DFT for a
 * buffer of data.  Callable as thread
 * \param arg Pointer to FTFuncArg argument
 * \return NULL
 */
static gpointer ThreadSkyModelFTDFT
  (gpointer args)
{
  /* Get arguments from structure */
  FTFuncArg *largs = (FTFuncArg*)args;
  glong        hivis  = largs->hivis;
  /* lowest vis (0-rel) */
  glong        lovis  = largs->lovis;
  /* Number of random parameters */
  glong        nrparm  = largs->nrparm ;
  /* Number of channels */
  glong        nchan  = largs->nchan;
  /* number of CC model components */
  glong        nmodel  = largs->nmodel;
  /* thread number, <0 -> no threading  */
  glong        ithread  = largs->ithread;
  /* data */
  gfloat       *data  = largs->data;
  /* model as flux, 2 pi x, 2 pi y, 2 pi z */
  gfloat       *model = largs->model;
  /* Frequency scaling array */
  gfloat        *freqArr = largs->freqArr;
  ObitThread    *thread = largs->thread;

  glong iVis, iComp, ichan, lrec, it, jt,
    itcnt, indx;
  gfloat *visData, *ccData, sumReal, sumImag,
    u, v, w;
#define FazArrSize 128  /* Size of arrays */
  gfloat AmpArr[FazArrSize], FazArr[FazArrSize],
    CosArr[FazArrSize], SinArr[FazArrSize];
  gfloat freqFact;


  lrec = nrparm + nchan*3;
  visData = data;
  /* Loop over vis */
  for (iVis=lovis; iVis<hivis; iVis++) {
    /* Loop over channel */
    for (ichan=0; ichan<nchan; ichan++) {
      freqFact = freqArr[ichan];
      u = freqFact*visData[0];
      v = freqFact*visData[1];
      w = freqFact*visData[2];
      indx = nrparm + ichan*3;
      /* Valid visibility? */
      if (visData[indx+2] <= 0.0) continue;
      sumReal = 0.0;
      sumImag = 0.0;
      ccData = model;
      /* outer Loop over models */
      for (it=0;it<nmodel;it+=FazArrSize) {
        itcnt = 0;
        /* inner Loop over models */
        for (iComp=it; iComp<nmodel; iComp++) {
          AmpArr[itcnt] = ccData[0];
          FazArr[itcnt] = ccData[1]*u +
            ccData[2]*v + ccData[3]*w;
          itcnt++;
          ccData += 4;
          if (itcnt>=FazArrSize) break;
        } /* end inner loop */
        /* Convert phases to sin/cos */
        ObitSinCosVec(itcnt,FazArr,SinArr,CosArr);
        /* Accumulate real and imaginary parts */
        for (jt=0; jt<itcnt; jt++) {
          sumReal += AmpArr[jt]*CosArr[jt];
          sumImag += AmpArr[jt]*SinArr[jt];
        }
      } /* end outer loop over model */
    } /* end loop over channel */
    /* Subtract */
    visData[indx]   -= sumReal;
    visData[indx+1] -= sumImag;
    visData += lrec;
  } /* end loop over vis */

  /* Done */
  if (ithread>=0)
    ObitThreadPoolDone (thread,
      (gpointer)&ithread);

  return NULL;
} /* end ThreadSkyModelFTDFT */
```

References

[1] W. D. Cotton, “A Fast Sine/Cosine Routine: Revenge of the Vector Processors,” *Obit Development Memo Series*, vol. 37, pp. 1–9, 2013.

[2] ——, “Note on the Efficacy of Multi-threading in Obit,” *Obit Development Memo Series*, vol. 1, pp. 1–8, 2008.