Comparison of GPU, Single– and Multi–threading for Interferometric Gridding

W. D. Cotton, January 17, 2014

***Abstract*—One of the more compute intensive aspects of radio interferometry is the conversion of the sampled visibilities to a "dirty" image. Interferometric measurements inherently randomly sample locations on the u–v plane making the direct usage of an fast Fourier Transform (FFT) to transform to the image domain impractical. The current practice is to convolve the measured samples with a continuous function which is then re-sampled and accumulated onto a regular grid. This grid can then be used with an FFT algorithm to efficiently derive an image from the data. This algorithm presents a number of challenges to an efficient implementation on Graphics Processor Units (GPUs). GPUs are optimized for image manipulation wherein memory is imaged sequentially in streaming operations. The convolutional gridding algorithm updates random patches in the grid resulting in inefficient GPU memory access. Furthermore, various combinations of convolved visibilities will overlap on the grid in a difficult to predict pattern so the accumulation into the grid must be done via an "atomic" operation to assure that the operation is done reliably. This memo explores several implementations of the convolutional gridding algorithm for single– and multi–threaded CPU and GPU applications as well as the inclusion of AVX coding in a multi–threaded test. The best GPU implementation tested was not quite as fast as the best simple multi–threaded CPU implementation but several times faster than a single threaded CPU implementation. Multi–threading with AVX intrinsics is the clear winner in this comparison being over two and a half times faster than the best GPU implementation. The frequency of serious memory errors in the absence of error correction is explored and found to be quite rare.**

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

I. Introduction

IMAGING of data from modern radio interferometers can be a very compute intensive operation. Data are sampled along continuous tracks of locations in the aperture (AKA pupil) plane, essentially the Fourier transform of the image plane. Use of the efficient discrete Fast Fourier Transform (FFT) algorithm to convert such data to the image plane is not possible. The current technique is to convolve the individual data samples with a continuous function of limited extent; the resultant function can then be re-sampled onto a regular grid and accumulated onto the vertices of this grid. The resulting regular grid can then be subjected to the FFT algorithm. The GPU implementations in this memo use nVidia cards programmed in CUDA.

II. Imaging Considerations

While gridding radio interferometer data can be compute intensive, it is only part of a more complex set of operations. For a general description of radio interferometry, see [1].

The convolving function used in gridding is relatively arbitrary but in practice, one with desirable properties is chosen. Two of the more desirable, and common, properties are limited support to minimize the computational load and good alias rejection of the artifacts of the application of the discrete FFT. [2] discusses anti–aliasing filters; a good compromise is a 7×7 spheroidal wave function as separable factors in u and v. A table of values can be pre–computed for a set of fractional u/v cells and used in the gridding.

The high sensitivity and dynamic range of current and planned interferometers generally require wide fields of view. Especially at lower frequencies, the field of view can be large enough that a flat approximation of the sky is inadequate [3]. The solution to this problem discussed in this memo is "faceting", covering the field of view with a piece-wise flat mosaic of sub-images [4].

Faceted imaging is very well suited to parallel processing as described in [4]. The same data is used in each facet with a different rotation of the (u,v,w) vector and a separate position shift. Furthermore, each facet uses a separate grid making a multi–threaded implementation straightforward.

III. Memory Access

A. memory latency

Digital memory is arraigned in a hierarchy with increasing size with increasing latency. The details depend on the particular hardware but generally include registers, cache(s), main memory and disk storage. Computing is more efficient when data accessed resides in low latency memory. The longer latency storage is generally read in continuous blocks into lower latency storage. GPUs have a further complication that memory accesses can be "coalesced" among threads of a thread block which greatly speeds access in a restricted set of circumstances. These all mean that memory access should be as sequential as possible and a thread block should access a contiguous block of memory with the appropriate alignment properties in a given cycle.

GPUs are optimized for image–like operations so put a high premium on sequential access of blocks of memory with fairly restrictive alignment properties. CPUs are better at hitting a random pattern in memory and frequently have relatively large caches to increase performance. GPUs tend to have more registers but smaller cache than CPUs.

B. memory updates

A further memory issue is the problem of updating memory from multiple independent threads. A common operation

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

consists of 1) reading a word from memory, 2) modifying it, 3) replacing it in memory. In general, there is no guarantee that when thread A performs this operation that thread B will not have modified the memory in question between steps 2 and 3; this is know as a race condition. In the context of gridding, there are two solutions, first, have multiple grids, one per active thread and second, "atomic" operations. In an atomic operation, the several steps are coupled and the memory in question is "locked" against modification by other threads during the operation.

Either of these solutions comes at some cost. Multiple grids require more memory and any necessary combination operations. Multiple grids are practical for a limited number of threads, especially when multiple facets are being gridded in parallel but becomes impractical for the thousands of threads available in GPUs.

Atomic adds avoid the complication of multiple grids but impose a substantial overhead. Locking CPU memory can dramatically increase the time for an operation; GPUs may have hardware support for atomic operations but still impose an overhead and threads updating a given memory word run sequentially. For nVidia GPUs programmed in CUDA there is the additional complication that neither CUDA arrays nor pinned host memory support atomic operations.

The interferometric gridding algorithm described above is a relatively poor match to best practices for CPUs and a very poor match for GPUs. The relatively arbitrary locations of the footprint of a given visibility in the grid allows for limited blocks of contiguous memory which seldom conform to the memory alignments for efficient GPU access. The arbitrary location of visibility samples in u–v space puts pressure on memory caching, usually favoring CPUs.

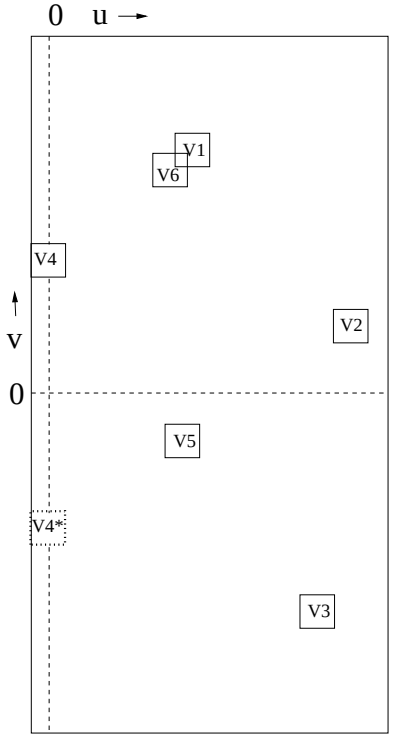

Fig. 1. Schematic view of the visibility grid showing the $u > 0$ half plane plus the additional columns needed to include the footprint of the convolution kernel in the $u < 0$ half plane. Square boxes are the footprints of the convolution kernels of the various visibility measurements, labeled V1...V6. V4 straddles the u=0 line and its conjugate is shown dotted in V4*. The footprints for V1 and V6 overlap allowing the possibility of a race condition when they are accumulated onto the grid.

IV. Gridding Implementations

Several test-bed programs were constructed to test gridding algorithms for single–threaded CPU, multi–threaded CPU, multi–threaded + AVX and GPU implementations. All performed the same operations, gridding the same set of simulated visibilities onto the same grids. Data were gridded onto multiple facets using a 7×7 convolution separated into u and v factors and pre–tabulated. GPU implementations were programmed in CUDA, CPU implementations in c. The operations used for each visibility are:

- Rotate u,v,w for facet
- Calculate position phase shift per channel
- Rotate phase of visibility per channel
- Apply data weighting
- Compute center grid cell for visibility
- Compute index for fractional cell convolution functions in u, v.
- For each cell in convolution kernel:
  - Multiple visibility by convolution function in u, v
  - Accumulate into grid

The gridding is for a single Stokes parameter and is therefore Hermitian; meaning only a single half plane needs to be accumulated. Data in the $u < 0$ half plane are folded onto the $u > 0$ half plane. Since the convolution function is of finite width, this will still result in cells in the $u < 0$ half plane being accessed. Complex logic inside of inner loops are always a hindrance to performance so extra columns are added to the grid for the few columns in negative u. When the gridding is finished, these columns need to be "flipped" and the complex conjugate added to the corresponding conjugate cells. A representation of the gridding process is illustrated in Figure 1 which shows overlapping visibility footprints and one overlapping into the $u < 0$ half plane. In practice, there will be a very large number of potentially overlapping convolution kernels. Gridding parameters are given a structure shown in the appendix in section **GridInfo Structure**.

A. Single–threaded CPU

The gridding routine is given in the Appendix in section **Single–Thread code** as routine gridKernel and the routine to "flip" the extra u columns is routine gridFlipKernel. Each call to gridKernel grids all the channels for a given visibility into the grid for a given facet.

B. Multi–threaded CPU

A multi–threaded implementation of the test gridding in which each call to the gridding routine processes all the channels for a given visibility into the grid for a given facet. Each facet is gridded in a separate thread.

In order to reduce the overhead of thread start-up, the thread pools of the gthreads library is used and the threading itself uses gthreads. The fast sine/cosine approximations in [5] were used.

C. Multi–threaded CPU+AVX

Intel Advanced Vector Extensions (AVX) technology is available on recent CPUs allowing significant performance enhancements. This uses the wide memory bus to operate on 8 floats simultaneously. The detailed implemention tested is structured in a similar fashion to the two pass GPU implementation described below. An implementation based on AVX intrinsics was included in a multi–threaded version of the test program.

D. GPU

Two variations of GPU implementations were tested. The first, a simple one pass implementation where each thread gridded all channels of a given visibility on a given facet. The relative performance of this implementation was poor so a second, two pass implementation was tested. The CUDA code for the single pass implementation is given in the appendix in section **one pass GPU code**.

The two pass implementation allowed for more efficient memory access but requires an outer loop over channel. The first pass calculates the visibility to be gridded for a given channel and facet and saves this as well as the cell and convolution kernel location in GPU memory for each visibility. A second pass then has one thread per cell in the convolution kernel per real and imaginary part. This allows contiguous blocks of memory to be updated each cycle. The second pass uses shared memory to speed access to the common data saved from the first pass. A different ordering of the tabulated convolution function also speeds access by locating the entries for a given fractional cell to be contiguous. The CUDA code for the two pass implementation is given in the appendix in section **two pass GPU code**. One difference between CPU and GPU implementations was that in CPU implementations, the computation of the phase to rotate each visibility to a given facet used double (64-bit) precision and the GPU implementation used single (32-bit) precision.

V. Timing Tests

Each of the implementations was tested using 100 iterations of gridding 10240 visibilities of 1024 channels into 7 facets for 2048×2048 cell images. This is roughly the equivalent of gridding several hours of EVLA B configuration data without baseline dependent time averaging. The data used uv coverage derived from a simulated EVLA data-set. The gridding parameters of the 7 facets were set to the same values in order to test the results for memory errors in the GPU implementations without error correction. GPU implementations used multiple streams to get maximal overlap of data transfer and computation.

Timing tests were performed on two machines, k2 and lhotse. k2 has an nVidia GTX25 GPU and lhotse has both

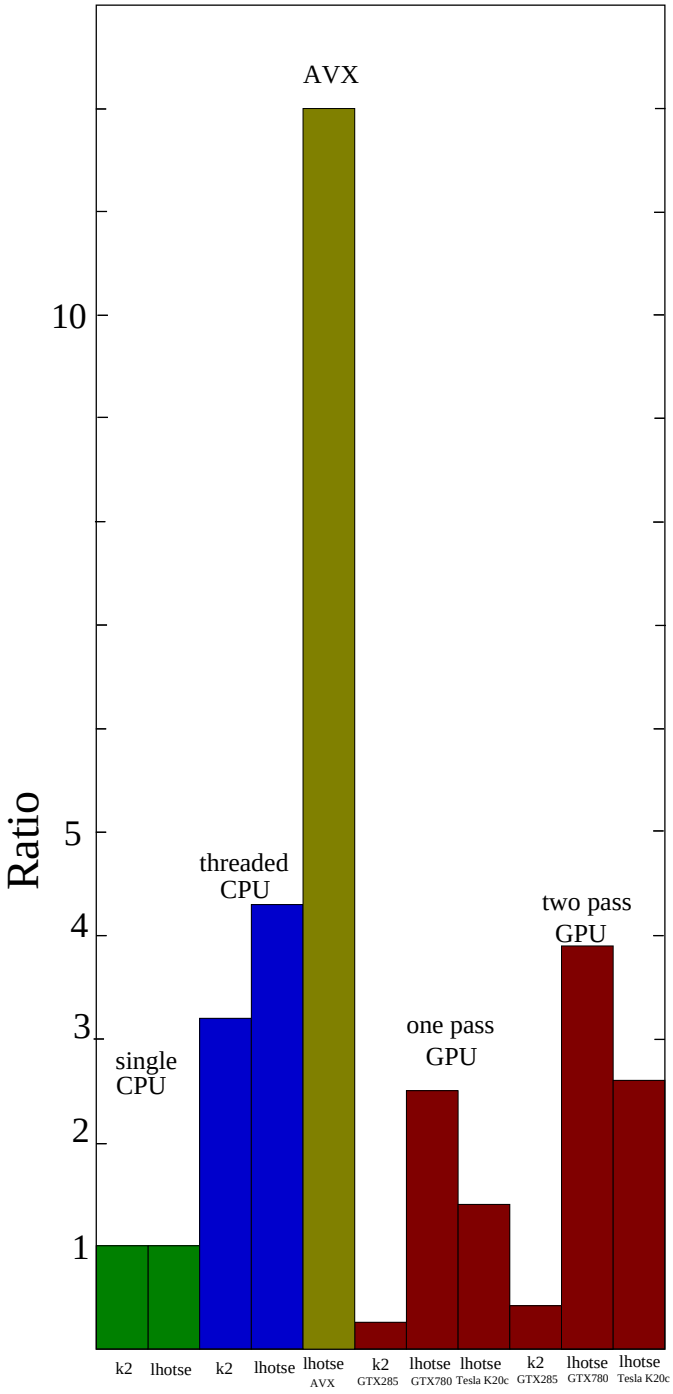

Fig. 2. Bar chart of performances relative to single–threaded CPU time on the same machine for the various tests. Green is single threaded CPU, blue is threaded CPU, yellow is threaded AVX and red GPU implementations.

an nVidia GTX780 and Tesla K20c GPUs. lhotse has 6 cores hyper-threaded, an Intel(R) Xeon(R) CPU E5-1650 0 @ 3.20GHz with cache size= 12288 KB and 64 GBytes of main memory. k2 has 4 cores hyper-threaded, an Intel(R) Core(TM) i7 CPU 940 @ 2.93GHz, cache size = 8192 KB and 12 GByte of memory. The GTX285 has 30 processors with 240 cores, the GTX780 has 12 processors with 2304 cores and the Tesla K20c has 13 processors with 2496 cores. The Tesla K20c was used with error correction enabled, the other GPUs have no error correction. The GTX780 and Tesla K20c GPUs were compute capability 3.5 while the GTX285 was 1.3. k2 does not support the AVX extensions so the threaded–AVX version was only tested on lhotse.

Timing results are given in table I. The "ratio" column gives the ratio of the single–threaded CPU time to the wall clock time of the given test for the given computer. The results are also presented in Figure 2.

VI. Memory Errors

nVidia makes two lines of GPUs, one intended for gaming applications (represented here by the GTX285 and GTX780) and the other for more general computing application (here the Tesla K20c). The two lines differ by approximately a factor of 4 in price (gaming versions being cheaper) but the gaming versions lack memory error correction. Memory errors have the potential to corrupt the computed results.

Interferometric imaging involved a large amount of averaging of noisy data making it relatively insensitive to GPU memory errors. Furthermore, the order of the execution of threads is not fixed and the results will differ in detail between different runs. Bit errors in the mantissa will generally not produce significant errors in the result but errors in the sign or exponent bits might.

The timing test involved computing multiple facet grids which are done independently but for purposes of detecting memory errors, the gridding parameters were all set to the same. At the end of the processing, the grids retrieved from the GPUs were compared and any difference greater than a part in 10^4 was flagged as an error. A loop over the timing test for the two pass gridding on the GTX285 on k2 was run for extended periods and the log examined to determine the frequency of significant memory errors. In 342 hours of testing using 634 executions of the timing program, no errors were detected by this test.

TABLE I
Timing Results

Method	CPU/GPU Device	Real time s	ratio
Single–Threaded CPU	k2	1024	1.0
Single–Threaded CPU	lhotse	806	1.0
Multi–Threaded(7) CPU	k2	316	3.2
Multi–Threaded(7) CPU	lhotse	188	4.3
Multi–Threaded+AVX	lhotse	67	12.0
1 pass GPU	k2/GTX285	3086	0.26
1 pass GPU	lhotse/GTX780	319	2.5
1 pass GPU	lhotse/tesla K20	576	1.4
2 pass GPU	k2/GTX285	1930	0.42
2 pass GPU	lhotse/GTX780	205	3.9
2 pass GPU	lhotse/tesla K20	312	2.6

VII. Discussion

A previous comparison, multi–threaded CPU and GPU DFT based interferometer model calculations [6] showed a dramatic performance advantage of the GPU whereas the best performance of a GPU in the present test was not quite as good as the simple multi–threaded result and threaded use of AVX intrinsics resulted in a speed 2.8 times better than the GPU. The difference between the gridding and model calculation tests is the pattern of memory usage. The earlier DFT test had far more computing per memory access and the pattern of memory usage was a good match to optimum GPU usage. The current gridding test, in contrast, has fewer computations per memory access and the pattern of memory access is a very poor match to optimum GPU usage.

The two pass version of the GPU gridding routines to allow for a better pattern of memory access resulted in a 50-60% reduction in run times for the GTX285 and GTX780 GPUs and an 85% reduction for the Tesla K20c using memory error correction. A variant of the GPU two pass test was run wherein the outer loop was over visibility and the inner loop over channel. This has the property that multiple channels of a given visibility will access the same region of the grid whereas independent visibilities will tend to be scattered on the grid. An inner loop over channel will increase the cache hit ratio but also increase the frequency of collisions in the atomic updates. The increased contention for atomic access appears to have overwhelmed the better cache hit ratio as this variant took 6 times longer on the GTX780 and over 25 times longer on the GTX285. Padding the length of the rows to minimize bank conflicts in the second pass of the two pass implementation had negligible effect on the GTX780 and slightly increased the run time on the GTX285.

The use of AVX intrinsics in the multi–threaded implementation is clearly superior to the GPU implementations tested for this application. The threaded+AVX test case ran 2.8 times faster than the two pass GPU test. It seems likely that much of this advantage is the more efficient memory access from reading and writing blocks of 8 floats. The single thread per facet grid removes the need for atomic adds. More CPU cores can be effectively used with a corresponding speed up by using a grid per core and then combining the grids for each facet when the gridding is complete. This technique resulted in a 45% speedup on smeagle (16 cores with AVX) when using 14 cores over using 7.

The use of memory error correction increased the run time by 50–80% while the occurrence of significant memory errors appears quite low. More testing of the occurrence rate of memory errors is needed to determine if the slower, and more expensive GPUs with error correction are warranted. In nearly a week and a half of continuous testing, no errors were detected by a simple (relatively insensitive) test on the GTX285.

The GTX285 run time in Table I was twice the single–threaded CPU time and 7 times slower than the multi–threaded CPU run time on the same machine. This older model of GPU appears not to be competitive with CPUs for this application.

ACKNOWLEDGMENT

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

APPENDIX

GridInfo Structure

The following is structure used to contain the gridding parameters and arrays..

```
/* Header file for testGrid */
/* Gridding info for GPU */
typedef struct  {
/** Number of facets */
long nfacet;
/** Number of channels */
long nchan;
/** Number of random parameters in visibilities */
long nrparm;
/** length in floats of visibilities */
long lenvis;
/** Size of facets (cells) */
long nx, ny;
/** Size of each grid (nx x ny/2+1+convWidth/2 x 2 floats) */
long sizeGrid;
/** Convolution kernal width */
long convWidth;
/** Number of convolution entries per cell */
long convNperCell;
/** Scaling for u,v to cells */
float uscale, vscale;
/** guardband in u, v in lambda */
float guardu, guardv;
//** shift parameters per facet [3] */
float *shift;
/** UVW, Rotation matrix per facet [3][3]*/
float *rotUV;
/** Frequency scaling array (nchan) */
float *freqArr;
/** gridding kernal */
float *convfn;
/** Facet grids, each sizeGrid floats */
float *grids;
} GridInfo;
```

Single–Thread code

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

```
// Processes all channels in a visibility for a given facet
// Grid has halfWidth extra columns in u added to avoid complication near u=0
//  The extra rows in the conjugate region need to be added to half plane before FFT.
void gridKernel(GPUGridInfo *gridInfo, gfloat *vis_in, int kvis, int ifacet, gfloat *debug)
{
    glong ivis         = kvis * gridInfo->lenvis; // beginning of visibility
    glong halfWidth    = gridInfo->convWidth/2;         // half width of convolution kernal
    glong fullWidth    = gridInfo->convWidth;           // full width of convolution kernal
    glong convNperCell = gridInfo->convNperCell;        // resolution of kernal
    glong ichan, iu, iv, icu, icv, jvis, lrow, halfv, it, off, iphase;
    gfloat *rot        = &gridInfo->rotUV[ifacet*9];
    gfloat *shift      = &gridInfo->shift[ifacet*3];
    gfloat *grid       = gridInfo->grids+(ifacet*gridInfo->sizeGrid);
    gfloat *convfn     = gridInfo->convfn;
    gfloat *gp;
    gfloat u,v,w, uu, vv, ww;
    gfloat vr, vi, vw, vvr, vvi, tr, ti;
    gfloat phase, c, s, phaseSign, freqFact;
```

```
    gfloat *convu, *cconvu, *convv;
    gdouble dshift[3], dphase;
    static const gdouble twopi = 2*G_PI;
    static const gdouble itwopi = 1.0/(2*G_PI);

    lrow  = 2*(1 + gridInfo->nx/2 + halfWidth);  // length of grid row in gfloats
    halfv = gridInfo->ny/2;
    dshift[0] = (gdouble)shift[0];
    dshift[1] = (gdouble)shift[1];
    dshift[2] = (gdouble)shift[2];

    //  Assume random parameters start with u,v,w
    u = vis_in[ivis];
    v = vis_in[ivis+1];
    w = vis_in[ivis+2];
    // rotate u,v,w for facet
    uu = u*rot[0] + v*rot[1] + w*rot[2];
    vv = u*rot[3] + v*rot[4] + w*rot[5];
    ww = u*rot[6] + v*rot[7] + w*rot[8];
    // Only gridding half plane, need to flip to other side?
    if (uu<=0.0) {
      phaseSign = -1.0;
      /*uu = -uu;*/
      /*vv = -vv;*/
      /*ww = -ww;*/
    } else { // no flip
      phaseSign = 1.0;
    }
    // loop over channels
    for (ichan=0; ichan<gridInfo->nchan; ichan++) {
      // real part of vis
      jvis = ivis + gridInfo->nrparm + ichan*3;
      freqFact = phaseSign * gridInfo->freqArr[ichan];
      u = uu * freqFact;  // Scale u,v,w to channel
      v = vv * freqFact;
      w = ww * freqFact;
      vvr = vis_in[jvis];
      vvi = vis_in[jvis+1];
      vw  = vis_in[jvis+2];
      /* Data valid? positive weight and within guardband */
      if ((vw>0.) && (u<gridInfo->guardu) && (fabs(v)<gridInfo->guardv)) {
// position shift, double  and subtract n pi turns
dphase =  (u*dshift[0] + v*dshift[1] + w*dshift[2]);
iphase = (glong)(dphase*itwopi);
phase = dphase - iphase * twopi;
s = sin(phase);
c = cos(phase);
vr = c*vvr - s*vvi;
vi = s*vvr + c*vvi;
// weighted data
vr *= vw;
vi *= vw;
// convert to cells
u *= gridInfo->uscale;
v *= gridInfo->vscale;
iu = (glong)(u+0.5) - halfWidth;
if (v>=0.0) iv = (glong)(v+0.5) + gridInfo->ny/2 - halfWidth;
else        iv = (glong)(v-0.5)+ gridInfo->ny/2 - halfWidth;
```

```
// start in convolution function
it = (glong)(convNperCell * (u - (int)u));
convu = convfn + it;
if (v>0) it = (glong)(convNperCell * (v - (glong)v));
else     it = (glong)(convNperCell * ((glong)v - v));
convv = convfn + it;
for (icv=0; icv<fullWidth; icv++) {
  off = iu*2 + (iv+icv)*lrow;
  gp = grid + off;
  cconvu = convu;
  for (icu=0; icu<fullWidth; icu++) {
    tr = vr*(*cconvu)*(*convv);
    ti = vi*(*cconvu)*(*convv);
    *gp += tr; gp++;
    *gp += ti; gp++;
    cconvu += convNperCell;
  } // end u convolution loop
  convv += convNperCell;
}  // end v convolution loop
      } // end valid
    }  // end channel loop
} // end gridKernel

// Flip/add conjugate rows in grid
// block.x = vrow, threads in block = facet
void gridFlipKernel(GPUGridInfo *gridInfo, glong ifacet, glong vrow)
{
   glong halfWidth    = gridInfo->convWidth/2;    // half width of convolution kernal
   gfloat *grid  = gridInfo->grids+(ifacet*gridInfo->sizeGrid);
   gfloat *gi, *go;
   glong iu, vc, lrow;

   lrow = 1 + gridInfo->nx/2 + halfWidth;  // length of grid row
   vc = gridInfo->ny - vrow;               // conjugate row number
   // loop over u columns
   gi = grid + vc*lrow + halfWidth;
   go = grid + vrow*lrow + halfWidth + 1;
   for (iu=0; iu<halfWidth; iu++) {
     go[0] += gi[0];
     go[1] -= gi[1];
     go += 2; gi -= 2;
   }
   }
} // end gridFlipKernel
```

one pass GPU code

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

```
// Integrated gridding routine, loops over channels
// Blocks are 16 x 16 visibilities
// Grid cells are facets x blocks of visibilities
// NB: Grid has halfWidth extra columns in u added to avoid complications
// near u=0. The extra rows in the conjugate region need to be added to
// half plane before FFT (gridFlipKernel).
//  gridInfo    = structure with gridding controls and arrays (testGrid.cuh)
//  vis_in      = Visibility array, 1 Stokes, assumed to start with u,v,w
__global__ void gridKernel(GridInfo *gridInfo, float *vis_in)
{
    int kvis          = blockIdx.y*blockDim.x*blockDim.x +
```

```
                    threadIdx.x*blockDim.x + threadIdx.y; // vis number
int ivis          = kvis * gridInfo->lenvis;       // beginning of visibility
int ifacet        = blockIdx.x;                    // facet number
int halfWidth     = gridInfo->convWidth/2;         // half width of convolution kernal
int fullWidth     = gridInfo->convWidth;           // full width of convolution kernal
int convNperCell  = gridInfo->convNperCell;        // resolution of kernal
float *rot        = &gridInfo->rotUV[ifacet*9];    // (u,v,w) rotation matrix per facet
float *shift      = &gridInfo->shift[ifacet*3];    // position shift parameters per facet
float *grid       = gridInfo->grids+(ifacet*gridInfo->sizeGrid);  // u-v grid per facet
float *convfn     = gridInfo->convfn;              // tabulated convolution function
float guardu      = gridInfo->guardu;              // u guardband (wavelengths)
float guardv      = gridInfo->guardv;              // v guardband (wavelengths)
float *gp;
int iu, iv, icu, icv, jvis, lrow, ichan, icufn, icvfn, offset;
float u,v,w, uu, vv, ww, vr, vi, vw, vvr, vvi, tr, ti;
float phase, c, s, phaseSign=1.0, freqFact;
float *convu, *cconvu, *convv, cu, cv;

lrow = 2*(1 + gridInfo->nx/2 + halfWidth);  // length of grid row in floats

//  Assume random parameters start with u,v,w
u = vis_in[ivis];
v = vis_in[ivis+1];
w = vis_in[ivis+2];
// rotate u,v,w for facet
uu = u*rot[0] + v*rot[1] + w*rot[2];
vv = u*rot[3] + v*rot[4] + w*rot[5];
ww = u*rot[6] + v*rot[7] + w*rot[8];
// Only gridding half plane, need to flip to other side?
if (uu<=0.0) {
  phaseSign = -1.0;
} else { // no flip
  phaseSign = 1.0;
}
// loop over channels
for (ichan=0; ichan<gridInfo->nchan; ichan++) {
  // index of real part of vis
  jvis = ivis + gridInfo->nrparm + ichan*3;
  vvr = vis_in[jvis];
  vvi = vis_in[jvis+1];
  vw  = vis_in[jvis+2];
  // Data valid? positive weight and within guardband
  if ((vw>0.) && (uu<guardu) && (fabs(vv)<guardv)) {
    freqFact = phaseSign * gridInfo->freqArr[ichan];
    u = uu * freqFact;  // Scale u,v,w to channel
    v = vv * freqFact;
    w = ww * freqFact;
    // position shift
    phase =  phaseSign * (u*shift[0] + v*shift[1] + w*shift[2]);
    __sincosf (phase, &s, &c);
    vr = c*vvr - s*vvi;
    vi = s*vvr + c*vvi;
    // weight data
    vr *= vw;
    vi *= vw;
    // convert to cells
    u *= gridInfo->uscale;
    v *= gridInfo->vscale;
```

```
        iu = lroundf(u);            // to grid cell number
        iv = lroundf(v);
        // start in convolution function
        icufn = lroundf((convNperCell*(iu + 0.5 - u)));
        convu = convfn + icufn;
        icvfn = lroundf((convNperCell*(iv + 0.5 - v)));
        convv = convfn + icvfn;
        cv = *convv;
        iu -= halfWidth;          // to absolute grid coordinates
        iv += gridInfo->ny/2 - halfWidth;
         for (icv=0; icv<fullWidth; icv++) {
           offset = (iu)*2 + (iv+icv)*lrow;
           gp = grid + offset;
           cconvu = convu;
           cu = (*cconvu)*cv;
           for (icu=0; icu<fullWidth; icu++) {
        tr = vr*cu;
        ti = vi*cu;
     // atomic update of grid
     atomicAddFloat(gp++, tr);
     atomicAddFloat(gp++, ti);
     cconvu += convNperCell;
     cu = (*cconvu)*cv;
           } // end u convolution loop
   convv += convNperCell;
   cv = *convv;
           }  // end v convolution loop
 } // end data valid
    }  // end channel loop
} // end gridKernel
```

two pass GPU code

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

```
extern  __shared__  float Cvx[];  // block convolution fn, u then v

// Routine allowing float atomicAdd on less capable systems
// address  = pointer for memory to update
// val      = value to add
// returns previous value in address
__device__ float atomicAddFloat(float* address, float val)
{
#if __CUDA_ARCH__ >=200
    return atomicAdd(address, val);
#else
    unsigned int* address_as_ul = (unsigned int*)address;
    union floatEquiv {
      unsigned int       iarg;
      float              farg;
    };
    union floatEquiv arg, assumed, old;
    old.iarg = *address_as_ul;
    do {
        assumed.iarg = old.iarg;
arg.farg = val+assumed.farg;
        old.iarg = atomicCAS(address_as_ul, assumed.iarg, arg.iarg);
    } while (assumed.iarg != old.iarg);
    return old.farg;
# endif
```

```
} // end atomicAddFloat

// First pass gridding routine, saves visibilities, center grid cells
// and first convolution indices in d_gridContr, d_gridCentr and_gridCfn
// Blocks are 16 x 16 visibilities
// Grid cells are facets x blocks of visibilities
// NB: Grid has halfWidth extra columns in u added to avoid complications
// near u=0. The extra rows in the conjugate region need to be added to
// half plane before FFT (gridFlipKernel).
//  gridInfo    = structure with gridding controls and arrays (testGrid.cuh)
//  vis_in      = Visibility array, 1 Stokes, assumed to start with u,v,w
//  d_gridContr = device storage for visibilities (r,i) per vis, per facet
//  d_gridCentr = device storage for center grid cells (u,v) per vis, per facet
//  d_gridCfn   = device storage for  first convolution indices (u,v) per vis/facet
//  ichan       = (0-rel) channel number
__global__ void gridPrepKernel(GridInfo *gridInfo, float *vis_in,
       float *d_gridContr, int *d_gridCentr, int *d_gridCfn, int ichan)
{
    int kvis          = blockIdx.y*blockDim.x*blockDim.y +
                        threadIdx.x*blockDim.x + threadIdx.y; // vis number
    int ivis          = kvis * gridInfo->lenvis;        // beginning of visibility
    int ifacet        = blockIdx.x;                     // facet number
    float *rot        = &gridInfo->rotUV[ifacet*9];     // (u,v,w) rotation matrix per facet
    float *shift      = &gridInfo->shift[ifacet*3];     // position shift parameters per facet
    float *gridContr  = &d_gridContr[ifacet*2*gridInfo->nvis];  // pointer to facet storage
    int   *gridCentr  = &d_gridCentr[ifacet*2*gridInfo->nvis];  // pointer to facet storage
    int   *gridCfn    = &d_gridCfn[ifacet*2*gridInfo->nvis];    // pointer to facet storage
    int halfWidth     = gridInfo->convWidth/2;         // half width of convolution kernal
    int convNperCell  = gridInfo->convNperCell;        // resolution of kernal
    int jvis, iu, iv, icufn, icvfn;
    float u,v,w, uu, vv, ww, vr, vi, vw, vvr, vvi, tr, ti;
    float phase, c, s, phaseSign=1.0, freqFact;

    // ignore vis more than the max
   if (kvis>=gridInfo->nvis) return;

    //  Assume random parameters start with u,v,w
    u = vis_in[ivis];
    v = vis_in[ivis+1];
    w = vis_in[ivis+2];
    // rotate u,v,w for facet
    uu = u*rot[0] + v*rot[1] + w*rot[2];
    vv = u*rot[3] + v*rot[4] + w*rot[5];
    ww = u*rot[6] + v*rot[7] + w*rot[8];
    // Only gridding half plane, need to flip to other side?
    if (uu<=0.0) {
      phaseSign = -1.0;
    } else { // no flip
      phaseSign = 1.0;
    }
    // channels values: index of real part of vis
    jvis = ivis + gridInfo->nrparm + ichan*3;
    vvr = vis_in[jvis];
    vvi = vis_in[jvis+1];
    vw  = vis_in[jvis+2];

    // Data valid? positive weight and within guardband
    if ((vw>0.) && (uu<gridInfo->guardu) && (fabs(vv)<gridInfo->guardv)) {
```

```
        freqFact = phaseSign * gridInfo->freqArr[ichan];
        u = uu * freqFact;  // Scale u,v,w to channel
        v = vv * freqFact;
        w = ww * freqFact;
        // position shift
        phase = phaseSign * (u*shift[0] + v*shift[1] + w*shift[2]);
        __sincosf (phase, &s, &c);
        vr = c*vvr - s*vvi;
        vi = s*vvr + c*vvi;
        // weighted data
        tr = vr * vw;
        ti = vi * vw;
        // convert to cells
        u *= gridInfo->uscale;
        v *= gridInfo->vscale;
        iu = lroundf(u);        // to center grid cell number
        iv = lroundf(v);
        icufn = convNperCell*lroundf(convNperCell*(iu + 0.5 - u));
        icvfn = convNperCell*lroundf(convNperCell*(iv + 0.5 - v));
        iu -= halfWidth;        // to absolute grid coordinates
        iv += gridInfo->ny/2 - halfWidth;
      } else {  // data invalid
        tr    = 0.0;
        ti    = 0.0;
        iu    = 0;
        iv    = 0;
        icufn = 0;
        icvfn = 0;
       }   // end data invalid

    // save to device memory
    gridContr[2*kvis]   = tr;
    gridContr[2*kvis+1] = ti;
    gridCentr[2*kvis]   = iu;
    gridCentr[2*kvis+1] = iv;
    gridCfn[2*kvis]     = icufn;
    gridCfn[2*kvis+1]   = icvfn;
} // end gridPrepKernel

// Second pass gridding routine, uses visibilities, center grid cells
// and first convolution indices in d_gridContr, d_gridCentr and_gridCfn
// Blocks are (uconv, x r,i) x vconv entries in the convolution
// threads are (width*2, width)
// Grid cells are facets x visibilities
// threads are (width*2, width)
//  gridInfo    = structure with gridding controls and arrays (testGrid.cuh)
//  d_gridContr = device storage for visibilities (r,i) per vis, per facet
//  d_gridCentr = device storage for center grid cells (u,v) per vis, per facet
//  d_gridCfn   = device storage for  first convolution indices (u,v) per vis/facet
//  ichan       = (0-rel) channel number
__global__ void gridAccumKernel(GridInfo *gridInfo,
       float *d_gridContr, int *d_gridCentr, int *d_gridCfn, int ichan)
{
    int kvis   = blockIdx.y;    // vis number
    int ifacet = blockIdx.x;    // facet number
    int halfWidth    = gridInfo->convWidth/2;         // half width of convolution kernal
    float *grid       = gridInfo->grids+(ifacet)*gridInfo->sizeGrid;  // facet grid
    int   cw          = gridInfo->convWidth;          // convolution width
```

```
    int i, jri, ju, jv, iuc, ivc, lrow, offset, icufn, icvfn;
    float tt, *gp;
    __shared__ float  gridContr[2];   // shared memory to speed access
    __shared__ int    gridCentr[2];
    __shared__ int    gridCfn[2];

    // ignore vis more than the max
    if (kvis>=gridInfo->nvis) return;

    //  global to shared memory, first two threads in block for u,v parts
    if ((threadIdx.x<2) && (threadIdx.y==0)) {
       i = threadIdx.x;
       gridContr[i] = d_gridContr[ifacet*2*gridInfo->nvis+2*kvis+i];
       gridCentr[i] = d_gridCentr[ifacet*2*gridInfo->nvis+2*kvis+i];
       gridCfn[i]   = d_gridCfn[ifacet*2*gridInfo->nvis+2*kvis+i];
    }

    //  which part of contribution to grid
    jri   = threadIdx.x%2;     // real (0), or imaginary (1)
    ju    = (threadIdx.x/2);   // u index of convolution
    jv    = threadIdx.y;       // v index of convolution
    lrow  = 2*(1 + gridInfo->nx/2 + halfWidth);  // length of grid row in floats
    __syncthreads();

    // data
    tt    = gridContr[jri];   // real/imag value
    iuc   = gridCentr[0];     // center cell in u
    ivc   = gridCentr[1];     // center cell in v
    icufn = gridCfn[0];       // first element in u convolution function
    icvfn = gridCfn[1];       // first element in v convolution function

    // Fetch convolution values to shared memory
    if ((threadIdx.y==0) && (jri==0)) Cvx[ju]    = gridInfo->convfn[icufn + ju];
    if (threadIdx.x==0)               Cvx[cw+jv] = gridInfo->convfn[icvfn + jv];
    __syncthreads();

    // multiply by convolution fn.
    tt *=  Cvx[ju]*Cvx[cw+jv];

    //  location in grid
    offset = (iuc+ju)*2 + (ivc+jv)*lrow + jri;
    gp = grid + offset;

    // atomic update grid
    atomicAddFloat(gp, tt);
} // end  gridAccumKernel

// Finish up gridding by adding nevative u cells to their conjugate points
// Blocks are facet
// Grid cells are rows in v
// block.x = vrow, threads in block = facet
//  gridInfo    = structure with gridding controls and arrays (testGrid.cuh)
__global__ void gridFlipKernel(GridInfo *gridInfo)
{
   long ifacet  = threadIdx.x;                  // facet number
   long vrow    = blockIdx.x;                   // v row number
   long halfWidth    = gridInfo->convWidth/2;    // half width of convolution kernal
   long fullWidth    = gridInfo->convWidth;      // full width of convolution kernal
```

```
   float *grid  = gridInfo->grids+(ifacet*gridInfo->sizeGrid);  // facet grid
   float *gi, *go;
   long iu, vc, lrow;

   lrow = 2*(1 + gridInfo->nx/2 + halfWidth);  // length of grid row
   vc = gridInfo->ny - vrow;              // conjugate row number

   // loop over u columns
   gi = grid + vc*lrow;
   go = grid + vrow*lrow + 2*fullWidth;
   for (iu=0; iu<halfWidth; iu++) {
     go[0] += gi[0];
     go[1] -= gi[1];   // conjugate
     go -= 2; gi += 2;
   }
} // end gridFlipKernel
```

REFERENCES

[1] A. R. Thompson, J. M. Moran, and G. W. Swenson, Jr., *Interferometry and Synthesis in Radio Astronomy, 2nd Edition*, Thompson, A. R., Moran, J. M., & Swenson, G. W., Jr., Ed. Wiley-Interscience, 2001.

[2] F. R. Schwab, "Optimal Gridding of Visibility Data in Radio Interferometry," in *Indirect Imaging. Measurement and Processing for Indirect Imaging. Australia, Cambridge University Press, Cambridge, England, New York, NY, L C # QB51.3.E43 I53 1984. ISBN # 0-521-26282-8. P.333, 1983*, 1983, pp. 333–+.

[3] R. A. Perley, "Imaging with Non-Coplanar Arrays," in *Synthesis Imaging in Radio Astronomy II*, ser. Astronomical Society of the Pacific Conference Series, G. B. Taylor, C. L. Carilli, and R. A. Perley, Eds., vol. 180, 1999, pp. 383–+.

[4] W. D. Cotton, "Parallel Facet Imaging in Obit," *Obit Development Memo Series*, vol. 6, pp. 1–3, 2009.

[5] ——, "A Fast Sine/Cosine Routine: Revenge of the Vector Processors," *Obit Development Memo Series*, vol. 37, pp. 1–9, 2013.

[6] ——, "Comparison of GPU and Multithreading for Interferometric DFT Model Calculation," *Obit Development Memo Series*, vol. 35, pp. 1–5, 2013.