SDP Memo 086: SKA-SDP Gridding on Graphical Processing Units

Document number	SDP Memo 086
Document Type	MEMO
Revision	1
Author	NVIDIA Corporation
Release Date	2018-10-25
Document Classification	Unrestricted

Lead Author:
NVIDIA Corporation

Released by:

Name	Designation	Affiliation
Bojan Nikolic	SDP Project Engineer	University of Cambridge
Signature & Date: [illegible]		

SKA-SDP Gridding on Graphical Processing Units

NVIDIA Corporation, 2701 San Tomas Expressway,
Santa Clara, CA 95050 USA

1. SKA Gridding

1.1. Basic Algorithm

In radio astronomy, the correlation of the signals from pairs of antennae give us samples of the sky image in the frequency domain. In order to convert this to an image in real space, we must Fourier transform the frequency space image. However, the samples, which are called visibilities are not on a regular grid. Gridding is the process of interpolating those off-grid visibilities to build an image on a regular grid. The function used to interpolate the signal is called the gridding convolution function, or GCF. The GCF may include corrections for the out-of-plane position of the antennae or incorporate some of the image cleaning operations.

To reiterate, the output image, in frequency space, is given by

$$I(w, v) = \sum V_i G(w - w_i, v - v_i)$$

where w_i and v_i are the positions of the visibilities, V_i are the values of the visibilities and G is the gridding convolution function. Since the GCF is of finite size, it is assumed to be zero outside that range. Not all visibilities, therefore, contribute to each image point.

Practically, the visibilities are truncated to a precision smaller than the image grid to limit the size of the GCF, which is typically pre-computed and stored in an array. To improve the performance of memory operations on all architectures, the GCF values for a specific offset within a grid point are collected and stored in contiguous memory.

1.2. Simple serial implementation

This is easily accomplished with the following c-code

```
for (int n=0;n<nvis;n++) {
   for(int wi=gcf_dim/2;wi>-gcf_dim/2;wi--) {
      for(int vi=gcf_dim/2;vi>-gcf_dim/2;vi--) {
         I[w[n]+wi][v[n]+vi] += V[n] * GCF[ws[n]][vs[n]][wi][vi];
      }
   }
}
```

In the above code the visibilities are assumed to be expressed in the `w`, `v`, `ws`, `vs` and `V` arrays with `nvis` as the number of total visibilities. The arrays `w` and `v` are the visibility positions truncated to the image grid, while `ws` and `vs` are the remainders truncated to the subgrid. So, if the original position was (4.1458, 12.8732) and the subgrid is 0.125, then `w,v` would be (4,12) and `ws,vs` would be (1,6). The GCF arrays for each subgrid position are of size `gcf_dim` $\times$ `gcf_dim` and are stored in an array. The variable GCF is a pointer to position `gcf_dim/2, gcf_dim/2` in the array for subgrid 0,0. The array `I` stores the final image. It is zero initialised.

2. Gridding on the GPU

There are several ways to parallelize the gridding algorithm on the GPU.

2.1. Scatter implementation

A simple GPU implementation of gridding assigns each visibility to a threadblock. Each thread in the threadblock computes the contribution for one image point, then writes it to the output image. Since more than one thread may attempt to write to a specific image point, an atomic addition must be used.

This simple implementation for a double precision GCF of size 128×128 can grid 400,000 visibilities onto a 4096×4096 image in 2.34 s, a throughput of 22.4 billion floating point operations per second, or GFLOPS. Here, and throughout this report, performance will be reported assuming 8 floating point operations per visibility per GCF point since the GCF and visibility are complex-valued.

A few easy changes can improve performance

- Sorting the visibilities by x-position then y-position to limit atomic collisions.

- Using launch bounds to boost occupancy. This enables the GPU to cover latency dispatching instructions to additional warps. Two threadblocks of size 32×32 gives the best performance for this implementation.
- Translating visibility positions to integers in a separate kernel. The first few bits give the position within the image grid, (`ws` and `vs` in 1.2). The remaining bits are the truncated position in the image (`w` and `v`).
- Using shared memory to load the visibilities coherently (i.e. using consecutive threads to read consecutive memory positions).

Following these changes, performance improves to 40.0 GFLOPS. In the code accompanying this report and in the git repository, this implementation is used by default. In the remainder of this report, it will be referred to as the scatter approach.

2.2. Gather implementation

Alternatively, we can grid by assigning each thread to exactly one image point. This eliminates the need for atomic addition since only one thread write to any memory location. However, it requires each thread to read many visibilities which are too distant from its image point to contribute. To mitigate this effect, we first sort the visibilities by their position on a lattice of size `gcf_dim/2`. Then, we create a set of bookmarks to enable indexing directly into each lattice position. In this way, we can dramatically reduce the number of repeated reads of visibilities.

This gather implementation achieves 89 GFLOPS for a double precision complex GCF of size 128 × 128 and a subgrid of 0.125 grid points. Note that the optimal launch bounds for each implementation may be different. In this case 50% occupancy gives the best performance. It causes 4 registers to "spill" meaning they are written to global memory, then read as needed.

The performance of this implementation can be improved slightly by running with smaller threadblocks and allowing each thread to compute, store and write more than one image point. This change provides more instruction-level parallelism. With more independent compute instructions per visibility, memory latency can be more effectively covered by dispatching additional instructions.

As before, we can used shared memory to ensure coalesced reads for visibilities and separate the integer part of the positions from the fractional part in a separate kernel. We can also boost performance by reading the GCF through the texture memory path. These changes boost the performance to 102 GFLOPS for the same 128×128 GCF.

Finally, we see slight improvement when we make multiple calls to the GCF kernel, applying a different slice of the GCF each time. The best performance for the 128×128 GCF called the kernel 4 times, each time applying a 128×32 slice of the GCF. This boosts performance to 107 GFLOPS.

Using the hardware counters on the GPU, we can isolate the performance limiters for this implementation. For the test case described above, the utilization of the memory access systems is higher than the utilization of compute resources. The memory operations are dominated by the read of the GCF. For the test case above, the GCF occupies 16 MB in memory which is far larger than both the L1 and texture cache of the GPU. The hit rate for the texture cache is about 18%. The effect of a smaller GCF and of computing the GCF in the kernel will be explored below.

2.3. Moving window implementation

A paper by John Romein of ASTRON[1] was the first to suggest an approach that seeks to minimize the frequency of atomic adds in the simple scatter implementation. To do this, each visibility is assigned to a threadblock as before. Each thread is assigned not to a single image point, but to a lattice of image points with spacing equal to the dimensions of the threadblock. See Figure 1. As before, each thread makes exactly one contribution to the image per visibility and more than one threadblock may contribute to a given image point. However, since threads stay fixed to a smaller number of image points, they are able to accumulate in registers most of the time and only call an atomic write when changing image positions.

This implementation achieves 55 GFLOPS. The performance limiter for this case is the atomic addition. Replacing these with a simple addition boosts the performance to 69 GFLOPS. Since double precision atomic instructions are not native for the Kepler architecture, we have implemented these using an atomic compare-and-swap. In Table 1, we show the performance for the moving window approach, 128×128 GCF, 4096×4096 images and 400,000 visibilities, for double and single precision using three different atomic addition methods. First, is the native atomicAdd command. This is not available for double precision on Kepler, but may be in the Pascal generation of GPUs arriving in 2016. Second, we use atomicCAS to perform the atomic. For the final column, we have replaced the atomic addition with a simple +=. This method gives inaccurate results, but the comparison will help isolate the effect of the atomic.

Table 1: Performance for different atomic addition methods

Precision	native	CAS	+=
single	122.6	92.8	114.1
double	–	54.2	68.7

For single precision, the native atomic is actually faster than the simple addition. Since the atomic addition has a separate pipeline, it reduces the pressure on the floating point arithmetic units, improving performance.

From table 1, we estimate that a hypothetical hardware double precision atomic would improve performance to between 71.6 and 73.8 GFLOPS.

3. Performance variation with problem parameters

Performance of the gridding methods described can vary with problem parameters such as the size of the GCF, the number of visibilities gridded at one time and the image size.

3.1. Dependence on GCF size

Some implementations are particularly sensitive to the size of the GCF. The gather implementation in particular suffers for small GCF dimensions. The graph below (Fig. 2) shows the performance of each implementation as a function of GCF size. For the gather implementation, the size of the GCF slices was varied from 32 to 128 and only the best performance is reported. For the moving window implementation, the threadblock height was varied between 4 and 128 to find optimal performance. The input data was a random set of 400,000 visibilities writing a 4096×4096 image. All values were double precision complex.

The gather implementation gives best performance for all GCF sizes, but the difference is quite small (42 GFLOPS vs. 33 GFLOPS) for small GCF.

3.2. Dependence on visibilities

The total number of visibilities can also have an effect on performance. For the moving window approach, in particular, more visibilities can allow the threadblock to "move" less between visibilities and reduce the number of atomic additions. In the gather implementation, for very small numbers of visibilities, the final write of a 4096×4096 image begins to dominate the instruction count. Figure 3 shows the performance of each implementation as a function of the total number of visibilities. For this data, we used a GCF dimension of 128 and an image size of 4096. All values were double precision complex.

The picture changes for smaller GCF dimensions. Figure 4 shows the performance of all three implementations for a 32×32 GCF. For this case, the moving window implementation shows better performance in all but a few cases.

3.3. Dependence on image size

Changing the size of the image can change the frequency of atomic collisions for the scatter and moving window approaches and change the number of repeated visibility reads for the gather approach. Figure 5 compares the performance of the three methods for a 128×128 double complex GCF and 400,000 visibilities.

All methods generally prefer a smaller image size

3.4. Computed GCF

For some convolution functions, it may be advantageous from a performance perspective to compute the value of the GCF inside the kernel, rather than precomputing and loading from global memory. This an especially tempting approach for the "gather" implementation, which is limited by the GCF read. However, in practice, even the simplest computed GCF hurts performance. In the experiments below, we will use a simple GCF given by

$$g = \exp(A - B * \sqrt{(w - w')^2 + (v - v')^2} \tag{1}$$

where A and B are constants, w and v are the position of the visibility, and w' and v' are the positions of the image point.

For the gather implementation and 400,000 visibilities, this change reduces performance from 107 GFLOPS to 77 GFLOPS. While the gather approach is limited by the GCF read instructions, the compute instruction units are also well utilized. The sincos added as part of the GCF computation triples the instruction count for the kernel. The use of fastmath versions of sin and cos turns this around. With the same test case and the same GCF and fast math, performance is boosted to 122 GFLOPS. The fast math versions of instrinsic functions like sqrt and sincos use single precision operations. This leaves the double precision compute utilities available for multiplication of GCF values and visibilities.

3.5. Single precision

Single precision speeds up performance for all implementations. Switching to single precision for the gather kernels with 4096×4096 images, 128×128 GCF and 400,000 visibilities improves performance from 107 GFLOPS to 145.7 GFLOPS. If the GCF is computed, performance improves from 122 GFLOPS to 151.1 GFLOPS. For the moving window approach, performance improves from 55.5 GFLOPS to 130.0 GFLOPS for a GCF read from memory and from 57.7 to 141.1 GFLOPS for a computed GCF.

3.6. Multiple polarizations

For some systems, signals for several polarizations are available for each visibility. Where memory allows, we can simultaneously build a separate image for each polarization. This has the potential to boost the ratio of floating point operations to fetched bytes since the GCF value need only be fetched once and then multiplied by each of the visibility values. Similarly, we reduce the ratio of integer and indexing operations to image point computations since the indexing can be done once for every four image values. In table 2 we report the performance for 400,000 visibilities with a 4096×4096 image and a 128×128 GCF for each of the three methods. A few details are important here. For these tests, the input data were stored as consecutive sets of N complex numbers, where N is the number of polarizations. The images were stored as one distinct, consecutive image for each polarization. This last point is very important for coalesced writing of the image information.

Also, multiple polarizations changes the kernel quite radically and some tuning of the parameters is required. To build the table, we adjusted the number of registers for each method; the block height (`BLOCK_Y` in Defines.h) for the moving window approach; and the number of points per thread (`PTS`) and width of the GCF stripes (`GCF_STRIPES`) for the gather approach. Also, for the moving window approach, it was necessary to specify explicit unrolling for the loops over polarizations using `#pragma unroll`. Only the best performance is recorded.

Table 2: Performance (in GFLOPS) for the simple scatter, gather and moving window methods for varying number of polarizations

Method	1	2	4
scatter	22.6	23.6	24.2
gather	109.5	171.3	260.6
moving window	58.8	67.6	73.6

4. Code Repository

The code for all the experiments reported here can be found on github at the following location:

`https://github.com/SKA-ScienceDataProcessor/GPUGrid.`

The README file in that repository gives instructions for switching between implementations as well as for enabling computed GCFs and fast math.

5. Conclusions

In this report, we have explored performance of various implementations of gridding on the GPU and report the following findings.

- A "gather" implementation, wherein each thread writes a single image point, seem to give best performance.
- The exception is for small GCF. For GCF dimensions of 32, a "moving window" approach gives best performance for some large and some small batches of visibilities.
- Performance suffers for batches of fewer than about 100,000 visibilities for the "gather" approach and about 4,000,000 visibilities for the moving window approach.
- Smaller images generally give better performance. That is, there is a super-linear effect with increasing image size. This effect is modest for the gather method, but greater for the moving window method.
- Building several images for multiple polarizations at one time can significantly boost performance, but increases GPU memory requirements.

- At this stage, and using Kepler GPUs, performance is limited by the bandwidth between DRAM and L2.
- Computing the GCF in the kernel improves performance, but only using fast math.

Further work could include investigating realistic, computed convolution functions and quantifying the error introduced by the use of fast math. We particularly thank John Romein for access to his code.

6. References

[1] John W. Romein. An Efficient Work-Distribution Strategy for Gridding Radio-Telescope Data on GPUs. In *ACM International Conference on Supercomputing* (ICS12), Venice, Italy, June 2012

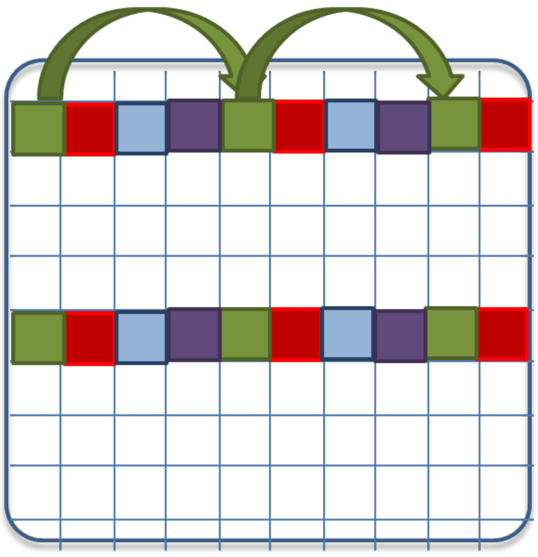

Figure 1: The lattice of image points for each thread

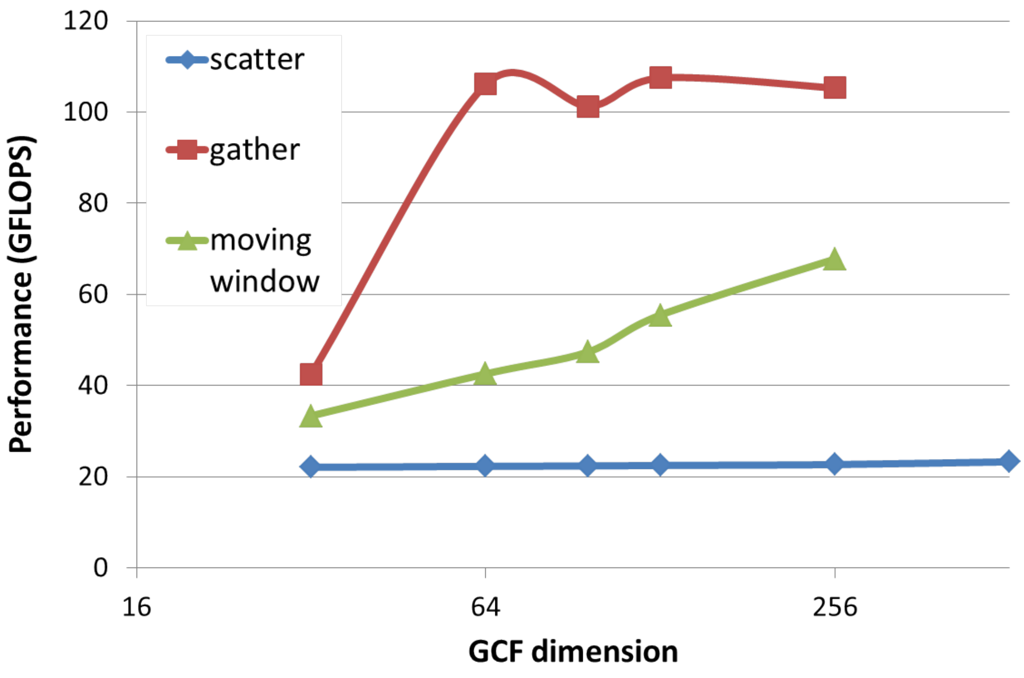

Figure 2: Performance for different implementations as a function of GCF dimension

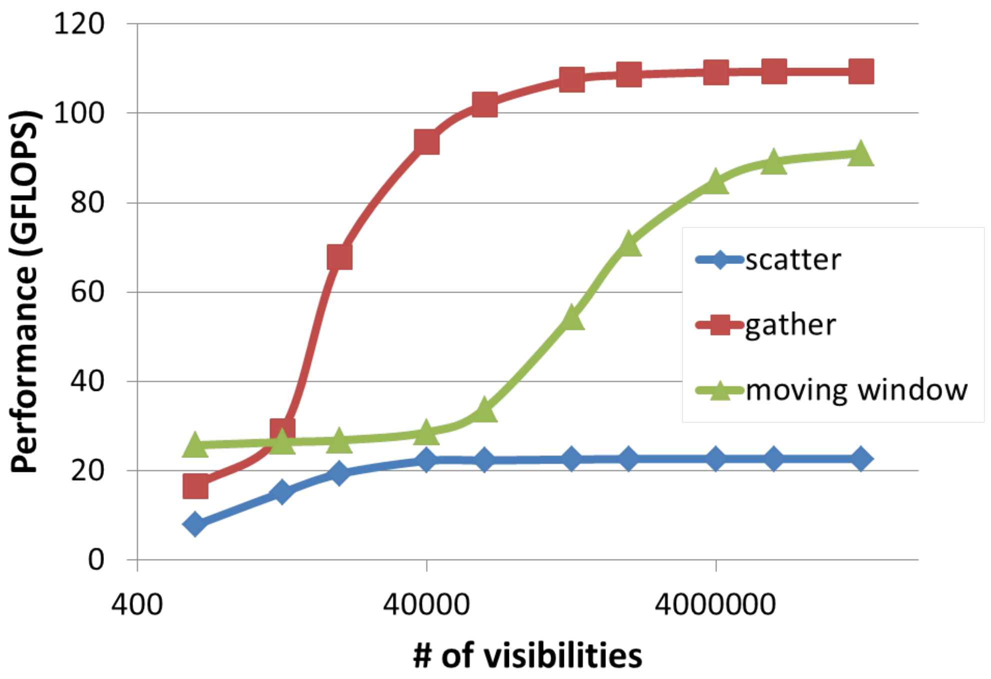

Figure 3: Performance for gridding implementations as a function of the number of visibilities for 128 $\times$ 128 GCF

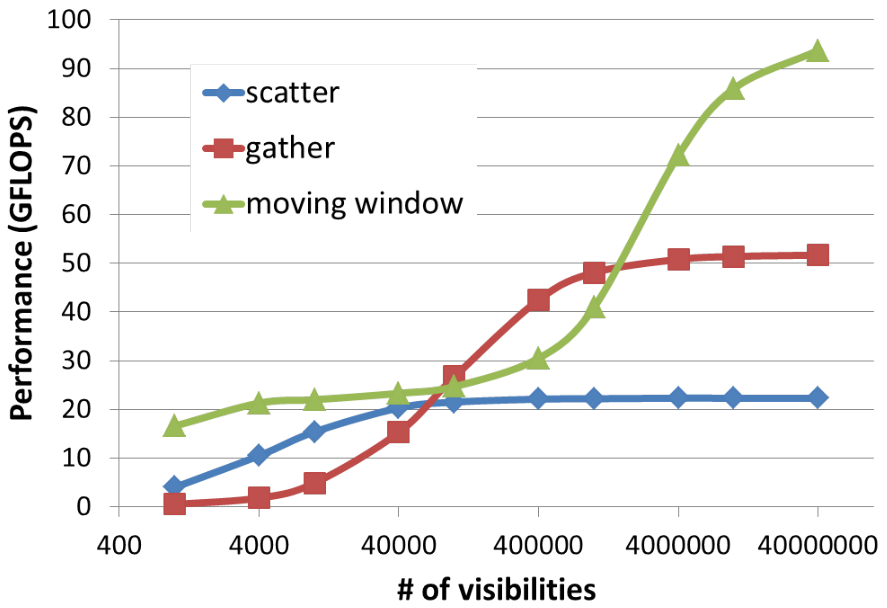

Figure 4: Performance for gridding implementations as a function of the number of visibilities for 32×32 GCF

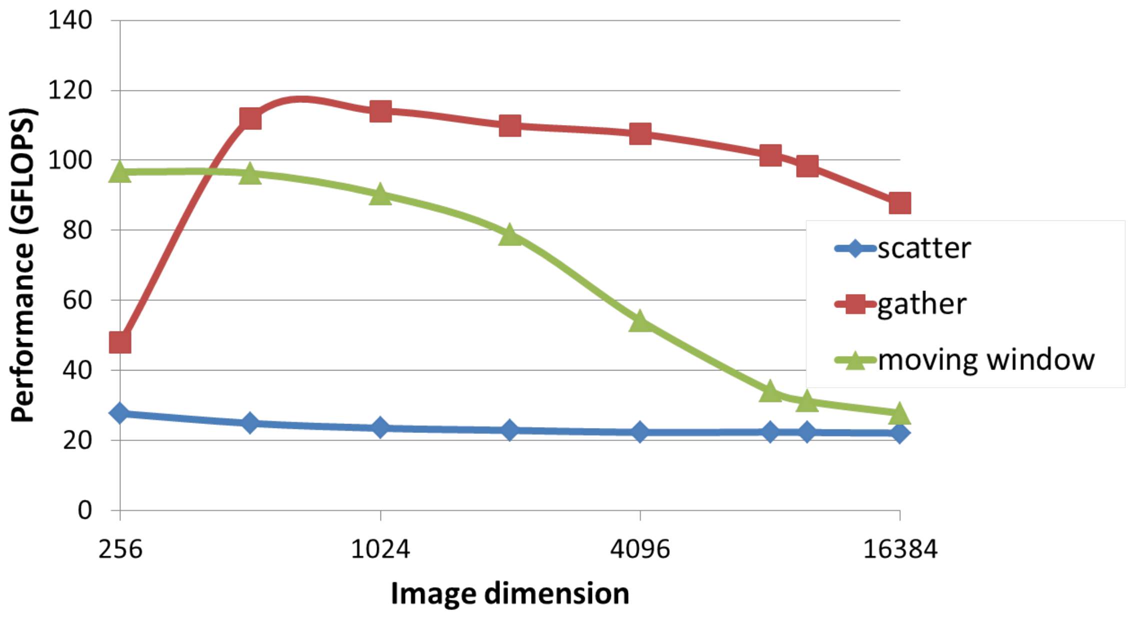

Figure 5: Performance for gridding implementations as a function of the number of image size