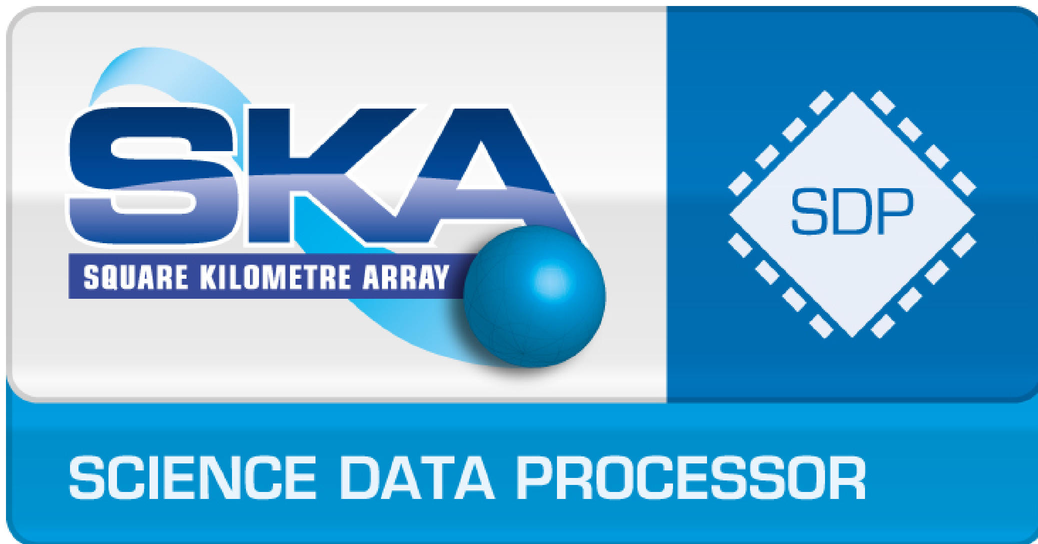
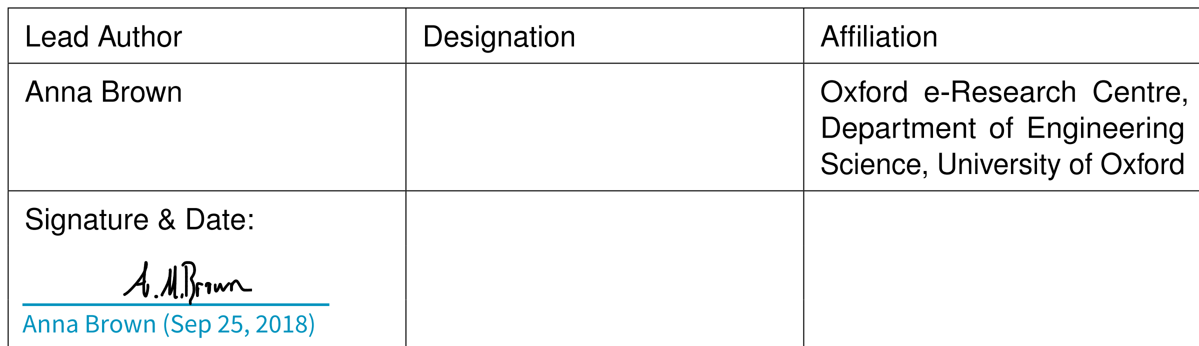


SDP Memo 73: Chebyshev polynomial approximation of kernels in w-projection gridding algorithm on GPU

Document Number .... SDP Memo 73
Document Type .... MEMO
Revision 1
Author .... Anna Brown, Wes Armour
Release Date 2018-09-25
Document Classification .... Unrestricted



Lead Author	Designation	Affiliation
Anna Brown		Oxford e-Research Centre, Department of Engineering Science, University of Oxford
Signature & Date: A.M.Brown Anna Brown (Sep 25, 2018)		

Revision	Date of issue	Prepared by	Comments
1	2018-09-25	Anna Brown	

SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Table of Contents

1 Introduction 3

2 w-projections kernels 3

3 Generating Chebyshev polynomial fits to w-projection kernels 3

4 Evaluation of Chebyshev polynomials 6

5 Integrating Chebyshev polynomial fitting with tile-based gridding approach 7

6 Conclusions and further avenues of investigation 12

References 12

1 Introduction

This report investigated the possibility of speeding up the w-projection gridding algorithm, in which w-projection kernels are convolved with visibility data, on GPU. This was done by approximating the w-projection kernels with Chebyshev polynomials, replacing a memory lookup to a large table with a lookup to a smaller table of coefficients plus evaluation of the polynomial on the fly – ie increasing computational work to reduce memory accesses.

This approach was compared to the GPU optimised tile-based gridding code by Jacques Du Toit of the Numerical Algorithms Group (NAG) [1]. While the basic gridding algorithm is naively bandwidth bound, it was found that optimisations introduced in the tile-based version make the algorithm sufficiently compute bound on GPU to make further addition of computational work increase rather than decrease the run time of the code.

2 w-projections kernels

w-projection kernels are only defined in the image plane but can be generated numerically in the visibility plane by taking the fourier transform at fixed w-values [2].

Maximum sizes for the kernels for typical data sets (used in [1]) are shown in Table 1. Here the oversampling factor is the increase in resolution of the kernels as compared to the grid, in order to account for the visibilities being continuous values that can lie between grid points. wsupport is the radius of the non-zero part of the w-kernel. The number of complex single precision data points stored per w-kernel for a particular w value is $((2 \times wsupport + 1) \times oversample)^2$.

The larger w-kernels are therefore too large to fit fully in L1 cache, which was motivation for trying to avoid using the w-kernel tables.

3 Generating Chebyshev polynomial fits to w-projection kernels

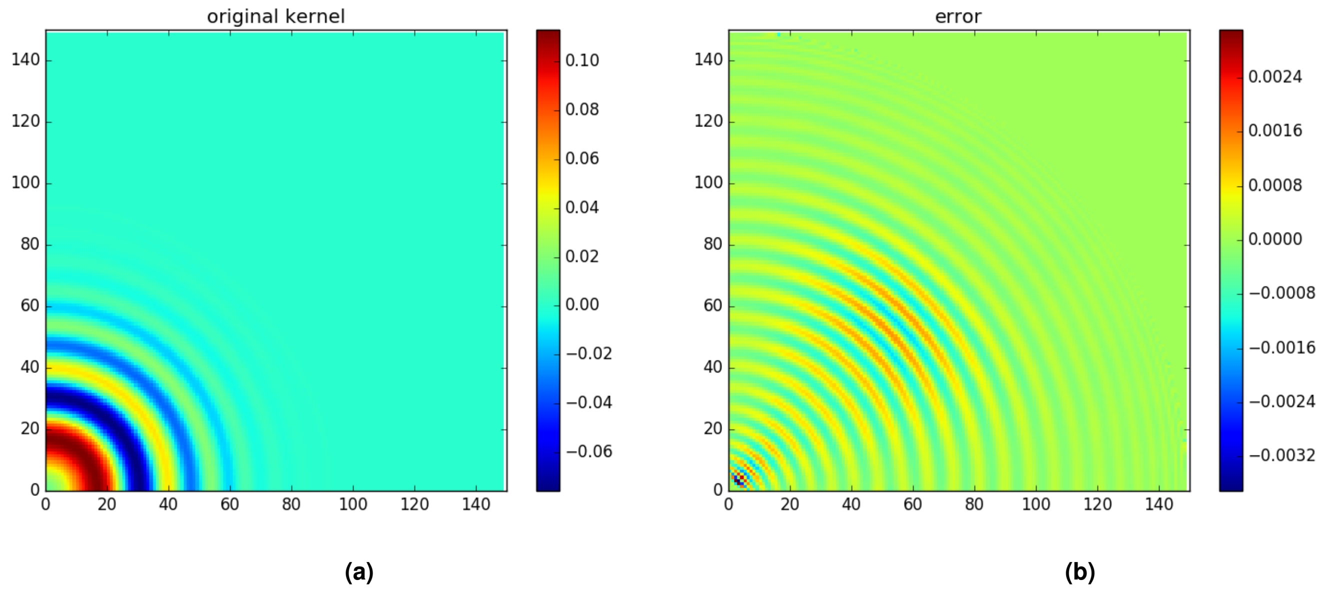
2D fitting

The w-kernels are close to being radially symmetric, being the FFT of the product of a radially symmetric complex function and a spheroidal tapering function. Due to their weak dependence on theta, 2D Chebyshev polynomial fitting was performed with significantly more coefficients in the radial than the theta direction.

The kernel used for testing is shown in Figure 1a. A chebyshev fit of this kernel was made and used to evaluate the values of the kernel at each point, then compared back to the original kernel. Figure 2b shows the RMS error across all points in the kernel, for various numbers of r and theta coefficients in the Chebyshev fit. Fitting was done using the modeling module of astropy v1.3.

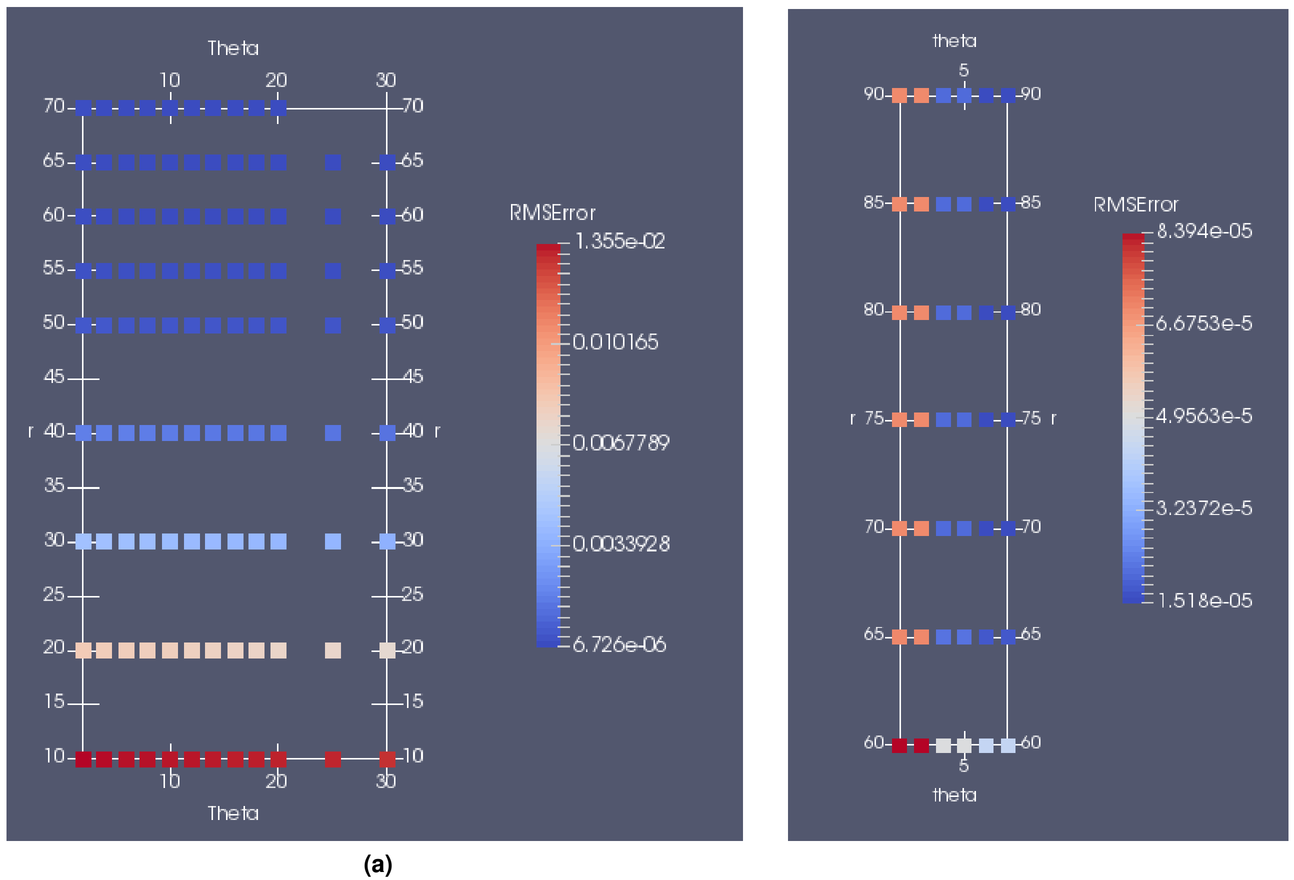
Data Set	oversample	max wsupport	max size (KB)
EL30-EL56	4	95	4560
EL56-EL82	4	72	2628
EL82-EL70	4	44	990

Table 1



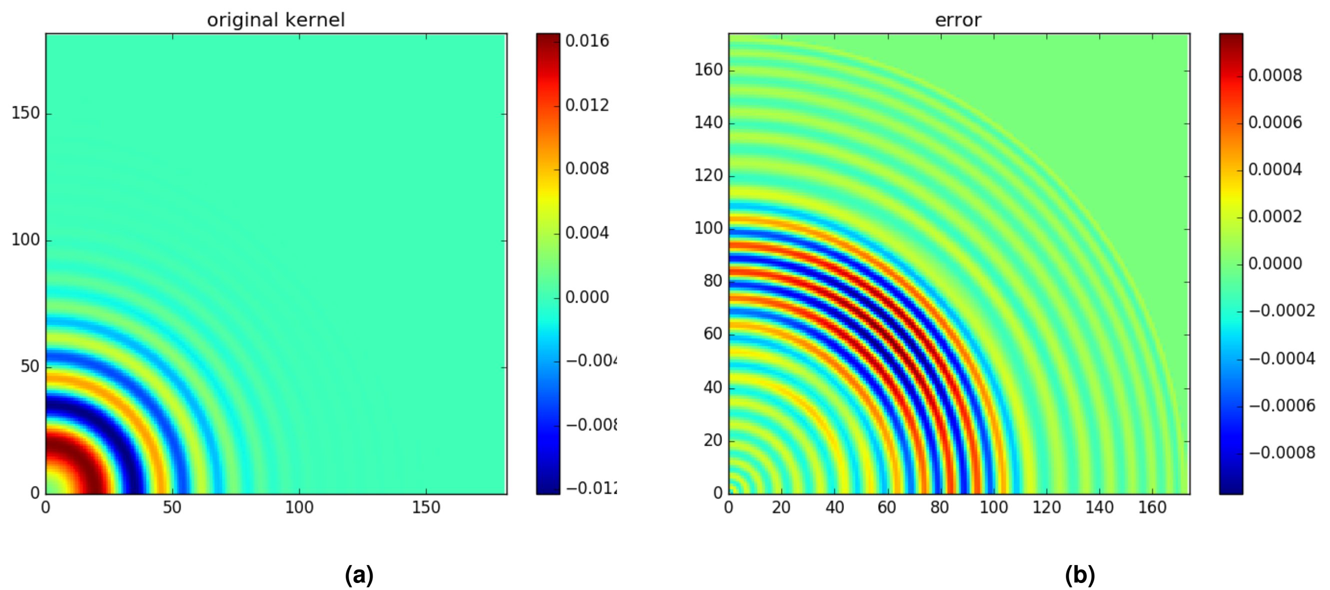
(a) (b)

Figure 1



(a)

Figure 2



(a) **(b)**

Figure 3

From this brief evaluation, it appeared that there were decreasing gains in accuracy after approximately 50 r coefficients and 4 theta coefficients. The absolute error for this configuration is shown in 1b. A larger w-kernel was used here to get an upper bound on the number of coefficients, but there should be fewer coefficients required for smaller kernels.

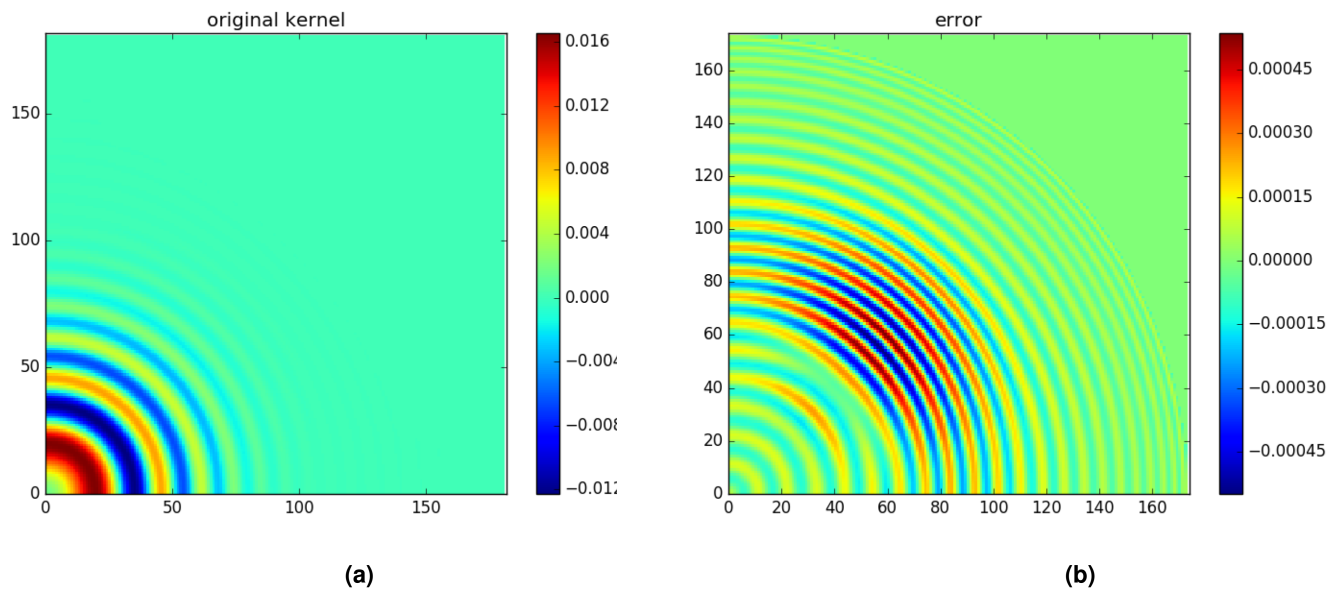
1D fitting

For a full understanding of acceptable error in the w-kernels, images should be generated from these kernels and the error in those images analysed. However, a rough comparison of the error in Figure 1b and the magnitude of the w-kernel values suggests that a 2D fit would require too many coefficients to reach an acceptable level of accuracy.

All further testing was therefore done to get an estimate of performance in the best case that a radially symmetric tapering function could be found. Figure 3a shows a kernel generated using a gaussian, and therefore radially symmetric, tapering function.

The 1D fit was performed with 50 coefficients in a line along y=0 and rotated to recreate the w-kernel for all grid points with radius¡=xMax. The original kernels are well defined for y¡0 (they are symmetric about the y axis) and for y¿yMax. It was found that a better fit could be achieved if the original kernels were extended slightly in each direction. The error compared to the original kernel is shown in Figure 3b for a fit to the original kernel plus 8 grid points on the left and 3 grid points on the right.

The RMS error across all points with radius¡=xMax was 0.00028 and the max error was 0.00098. Note that the value of the fit is undefined for radius¿xMax, so any gridding code using this method would have to be adapted to use a spherical rather than square kernel.



(a)

(b)

Figure 4

1D fitting to r^2

In order to try to improve the fit by stretching out the high gradient area close to the origin, and in order to save a square root when evaluating the radius of each grid point from x and y coordinates, a fit was made to r^2 rather than r. This means that for each data point the x axis coordinate was relabelled to the value x^2 without changing the value at that point, and those data points fed as inputs to the fit. When evaluating the value of the kernel from the Chebyshev polynomial fit at each point in 2D space, the fit was evaluated at coordinate $r^2 = x^2 + y^2$. The error using this method to fit to the same kernel as in Figure 3b is shown in Figure 4b. The RMS error was 0.00014 and the max error was 0.00055.

While fitting to r^2 seems promising for this particular kernel, there were some issues with the r^2 fit to very small kernels being worse than the r fit – it may be that the number of coefficients should be reduced for kernels with small wsupport.

4 Evaluation of Chebyshev polynomials

The following algorithm was used to recursively evaluate a 1D Chebyshev polynomial of order numCoeffs-1 at position r:

```
evalChebyshev(r, coeffs, numCoeffs):
        tMinusMinus=1
        tMinus=r
        chebSum = coeffs[0] + coeffs[1]*tMinus
        for i=2 to numCoeffs-1
                tNext=2.0*r*tMinus - tMinusMinus
                chebSum = chebSum + coeffs[i]*tNext
                tMinusMinus = tMinus
                tMinus = tNext
```

This is difficult to parallelize as each iteration depends on the result of two previous iterations. Instead, in this implementation one thread is used to perform this algorithm in full for a given position r.

This method requires approximately $2 * numCoeffs$ floating multiply-add operations and $numCoeffs$ reads from constant memory.

5 Integrating Chebyshev polynomial fitting with tile-based gridding approach

Tile based approach

Previous optimisation of the gridding algorithm by Jacques Du Toit of NAG resulted in a tile-based gridding code, available at `https://github.com/OxfordSKA/GPU-gridding/tree/NAG` and documented in [1].

In this approach, the entire grid is seperated into tiles and visibilities are sorted into these tiles based on their coordinates. Thread blocks are then assigned one tile each such that each thread is responsible for one grid point in a tile, and blocks then iterate over all visibilities in that tile. If a grid point is covered by the support of a particular visibility, it is convolved with that visibility and the result stored in registers or shared memory (otherwise the thread stands idle). After visibilities have been processed, registers and shared memory are written back to the global grid using atomics. This has the advantage of reducing memory writes back to the global grid and moving most of the memory accesses to fast registers and shared memory.

For the purpose of load balance, the grid is separated into a central and outer region, with the central region having a much higher visibility density, and processed in two seperate gridding kernels.

Chebyshev fitting implementation

Figure 5 shows pseudocode of the original gridding step. See the NAG report for further explanation. Figure 6 shows how the code was adapted to use Chebyshev fitting. Instead of reading the value c of the wkernel from the array conv_kernel, the value of the kernel is calculated from the value of r^2 (rad_squared) at that grid point, where the radius is measured from the exact coordinate of the visibility to the centre of the grid point. The function evalChebyshev takes r^2 as well as a list of coefficients for the correct w value, and implements the algorithm in section 4. As the value of the chebyshev polynomial is not defined for $r^2 > x_{max}^2$, an additional check is required to only evaluate the kernel within this radius.

```
// Pick up which tile we should process
const int pu = blockIdx .y , pv = blockIdx . z ;
// Multiple blocks will process this tile
const int bid = blockIdx .x , nblks = gridDim . x ;
const int my_grid_u_idx = threadIdx . x + tile_u_offset ;
const int my_grid_v_idx_start = threadIdx . y * ( REGSZ + SHMSZ ) + tile_v_offset ;
// Allocate the registers and shared memory
float2 regs [ REGSZ ];
extern __shared__ float2 shmem [];
/* ... snip : set both to zero : ... snip */
for ( int i = tileOffset + bid ; i < tileOffset + num_vis ; i += nblks )
{
    /* ... snip : read in visibility data , set up variables: ... snip */
    const int k = my_grid_u_idx - grid_u ;
    bool is_my_k = (- wsupport <= k && k <= wsupport );
    if ( is_my_k ) {
        for ( int r =0; r < REGSZ ; r ++) {
            const int j = my_grid_v_idx_start + r - grid_v ;
            const bool is_my_j = ( - wsupport <= j && j <= wsupport );
            if ( is_my_j ) {
                float2 c = conv_kernel [ kernel_start + iy * size + ix ];
                c . y *= conv_mul ;
                regs [ r ]. x += ( val . x * c . x - val .y * c . y );
                regs [ r ]. y += ( val . y * c . x + val .x * c . y );
            }
         }
         for ( int s =0; s < SHMSZ ; s ++) {
             /* ... snip : shared memory is treated similarly : ... snip */
         }

     } // END if ( is_my_k )
/* ... snip ... */
```

Figure 5

```
// Pick up which tile we should process
const int pu = blockIdx .y , pv = blockIdx . z ;
// Multiple blocks will process this tile
const int bid = blockIdx .x , nblks = gridDim . x ;
const int my_grid_u_idx = threadIdx . x + tile_u_offset ;
const int my_grid_v_idx_start = threadIdx . y * ( REGSZ + SHMSZ ) + tile_v_offset ;
// Allocate the registers and shared memory
float2 regs [ REGSZ ];
extern __shared__ float2 shmem [];
/* ... snip : set both to zero : ... snip */
for ( int i = tileOffset + bid ; i < tileOffset + num_vis ; i += nblks )
{
    /* ... snip : read in visibility data , set up variables: ... snip */
    const int k = my_grid_u_idx - grid_u ;
    bool is_my_k = (- wsupport <= k && k <= wsupport );
    if ( is_my_k ) {
        for ( int r =0; r < REGSZ ; r ++) {
            const int j = my_grid_v_idx_start + r - grid_v ;
            const bool is_my_j = ( - wsupport <= j && j <= wsupport );
            if ( is_my_j ) {
                float2 distance = // get distance between centre of grid point
                                  // and visibility coordinate
                rad_squared = distance.x*distance.x + distance.y*distance.y
                max_rad = // get maximum x coordinate
                if ( rad_squared <= max_rad*max_rad) {
                    rad_squared = 2*rad_squared/(max_rad*max_rad) - 1;
                    float2 c;
                    c.x = evalChebyshev(rad_squared, real_cheb_coefficients, num_coeffs);
                    c.y = evalChebyshev(rad_squared, imag_cheb_coefficients, num_coeffs);
                }
                c . y *= conv_mul ;
                regs [ r ]. x += ( val . x * c . x - val .y * c . y );
                regs [ r ]. y += ( val . y * c . x + val .x * c . y );
            }
        }
         for ( int s =0; s < SHMSZ ; s ++) {
             /* ... snip : shared memory is treated similarly : ... snip */
        }

     } // END if ( is_my_k )
/* ... snip ... */
```

Figure 6

Data Set	W	Central Box (ms)	Remaining Tiles (ms)	Total (ms)
EL30-EL56	15	595	724	1319
EL56-EL82	6	174	466	640
EL82-EL70	4	129	286	415

(a)

Data Set	W	Central Box (ms)	Remaining Tiles (ms)	Total (ms)
EL30-EL56	15	404	546	950
EL56-EL82	6	186	329	515
EL82-EL70	4	141	249	390

(b)

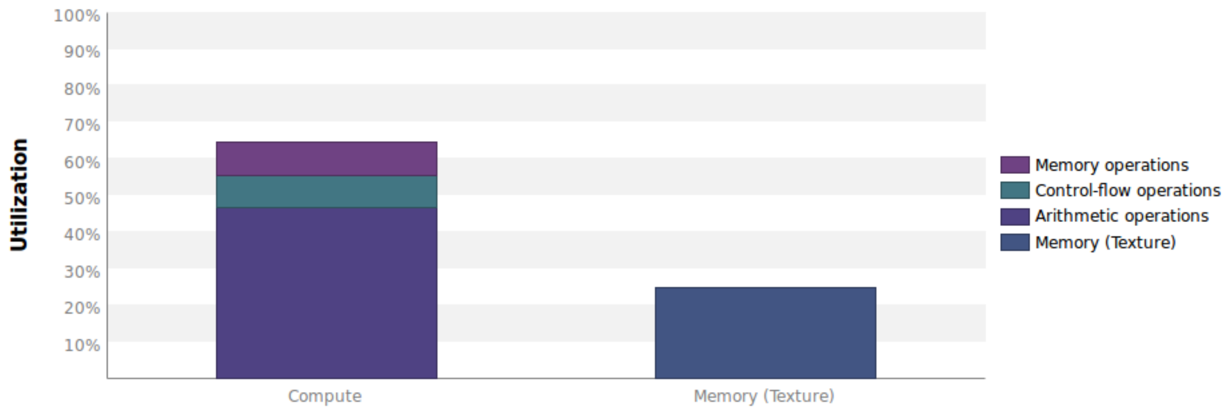
Table 2

Performance

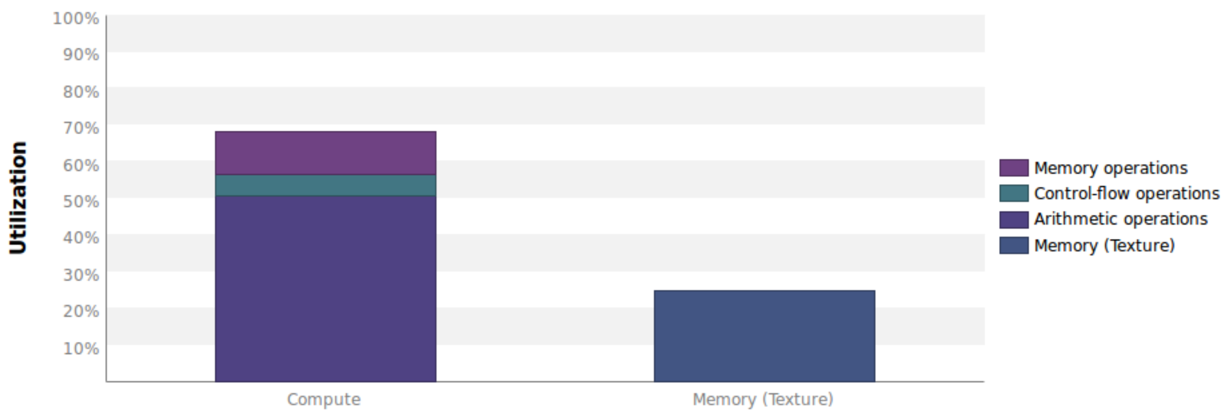
The tile-based code rearranges the order of data points within w-kernels to reduce strided access due to oversampling of the kernels, which appears to significantly improve the caching behaviour of the kernels. This does however leave less room for improvements through removing memory reads to w-kernels, as shown in Tables 2a and 2b. These show the timing for the data sets used in the NAG report with the original code (2a) and the timing for the same code but with references to wkernels replaced with references to a subset of a single wkernel, which can fit in L1 cache (2b). Based off this, there is a maximum 30% improvement possible for the largest data set through improving or eliminating reads to wkernels.

The other difficulty found in adapting the tile based approach was that it is very register heavy, dividing memory local to each block evenly between shared memory and an array of registers. The evaluation of Chebyshev polynomials should put additional load on registers and other compute resources. In addition, the original code has high branch divergence due to only some threads in a warp needing to process a particular visibility based on the distance of the visibility from each thread's grid point. This is seen in figure 7a, which shows the performance limiter for the original tile-based code – compute utilisation is much larger than memory utilisation.

The chebyshev implementation with 50 coefficients was found to run much slower than the original tile-based implementation, taking 3115 ms versus 416 ms to run the outer and inner gridding kernels. Even reduced to 11 coefficients, which from the accuracy analysis above would not seem sufficient, the chebyshev code runs slower, at 952 ms. Figure 7b shows the performance limiter for the 11 coefficient version of the chebyshev code. It can be seen that there is higher compute utilisation with no significant reduction of memory usage. This, combined with the longer runtime, suggests that we are seeing the greater workload required to evaluate chebyshev polynomials. The other main factor contributing to the longer runtime of the chebyshev code appears to be warp divergence, with a non predicated warp efficiency of 38% for the chebyshev code versus 60% for the original code. This may be due to the need to eliminate data points which lie outside the maximum radius of the w-kernel, but could use further investigation.



(a)



(b)

Figure 7

6 Conclusions and further avenues of investigation

Adding Chebyshev fitting of w-kernels to the w-projection gridding code optimised by NAG introduces accuracy errors without any benefit to the run time of the code. This is believed to be due to the highly optimised accesses to w-kernels used in that code, as well as the fact that the code is already compute bound and therefore doesn't have spare compute resources for additional operations to evaluate chebyshev polynomials.

This highlights the importance of understanding the limiting factor (eg compute or bandwidth) of an algorithm when optimising that algorithm. It also shows that measuring percentage useage of the available compute resources on a GPU is not in itself enough to predict the performance of a code. As seen here, it is possible to use more compute resources on the GPU by choosing an algorithm that requires more work, without increasing the run time of the code.

It is expected that any compute bound gridding code would have the same problems found here, though it may be worth implementing some kind of kernel fitting scheme for a memory bound code.

Any code which used chebyshev fitting in 1D as done here would still need to address the problem of finding an acceptable radially symmetric tapering function, with a more thorough investigation of the distortions that would be introduced at the corners the (square) image and which are currently minimised by using a square tapering function.

As an alternative to Chebyshev fitting, it may be worth investigating splines, which would use more total coefficients but may reduce the number of calculations needed to evaluate the wkernel at each particular grid point.

Finally, as branch divergence is a significant factor in performance, it may be worthwhile seeing how divergence is affected by using more realistic data, in which the wsupport of visibilities should change much more slowly over time.

References

[1] Dingle, du Toit & Hopkins, *SDP Memo 36: Convolution Gridding on CPU, GPU and KNL*, The Numerical Algorithms Group, Manchester, 2017

[2] Cornwell, Golap & Bhatnagar (2008), IEEE Journal of Selected Topics in Signal Processing, Vol. 2, Issue 5, p.647-657