# An Efficient Work-Distribution Strategy for Gridding Radio-Telescope Data on GPUs

John W. Romein
romein@astron.nl
Netherlands Institute for Radio Astronomy (ASTRON)
Postbus 2, 7990 AA Dwingeloo, The Netherlands

## ABSTRACT

This paper presents a novel work-distribution strategy for GPUs, that efficiently convolves radio-telescope data onto a grid, one of the most time-consuming processing steps to create a sky image. Unlike existing work-distribution strategies, this strategy keeps the number of device-memory accesses low, without incurring the overhead from sorting or searching within telescope data. Performance measurements show that the strategy is an order of magnitude faster than existing accelerator-based gridders. We compare CUDA and OpenCL performance for multiple platforms. Also, we report very good multi-GPU scaling properties on a system with eight GPUs, and show that our prototype implementation is highly energy efficient. Finally, we describe how a unique property of GPUs, fast texture interpolation, can be used as a potential way to improve image quality.

## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; J.2 [**Physical Sciences and Engineering**]: Astronomy

## General Terms

Algorithms, Experimentation, Performance

## Keywords

Gridding, sky image, convolutions, GPU

## 1. INTRODUCTION

During the past decades, astronomers, computer scientists, and engineers have been developing new generations of radio telescopes that improve on sensitivity, image resolution, and data quality. The data rates of these telescopes are enormous and increasing with every generation, and so are the processing requirements. New types of radio telescopes like LOFAR [13], that uses tens of thousands of simple receivers rather than some tens of large dishes, even rely more on digital signal-processing techniques than ever before. This trend continues with the development of a telescope more powerful than all other telescopes in the world together — the Square Kilometre Array (SKA) [4].

The imager is a critical component in the data processing pipeline of a telescope. Basically, the sampled data from the telescopes is (after considerable preprocessing) added to a grid, after which the grid is Fourier transformed to create a sky image. It is also one of the most expensive operations, in terms of processing requirements. For LOFAR, roughly half the time of all post-observation processing is spent in creating sky images. For the SKA Phase 1, the required amount of image processing power is estimated to be in the petaflop range [3], and in the exaflop range for the full SKA.

The gridding stage is a good candidate for parallel processing on many-core accelerators like GPUs. However, traditional gridder implementations, designed to run on CPUs, heavily rely on memory caches and high main-memory bandwidth. Accelerators have less bandwidth per FLOP, threatening the efficiency with which this application can be run. Recently, some other work-distribution strategies for accelerators have been published (in this paper, we use the term *algorithm* for sequential algorithm, and the term *work-distribution strategy* or *strategy* for the way in which an algorithm is parallelized). Some of these strategies improve on spatial locality and thus on memory performance, at the cost of additional computations, by sorting and searching data [9, 5, 6, 11]. None of these efforts achieves more than 14% of the peak FPU performance; they are typically closer to 4%. This illustrates that achieving good performance for this algorithm is hard. Moreover, at these efficiencies, one cannot hope to build the SKA.

This paper presents a new, highly efficient work-distribution strategy for GPUs, that grids radio-telescope data typically an order of magnitude faster than other GPU gridders. Our strategy minimizes device-memory accesses, but does not rely on sorting or searching data. We implemented the strategy in CUDA and OpenCL, and compare performance on several high-end platforms. We show that the strategy scales well on an eight-GPU system, and that it is highly energy efficient. We also describe how texture interpolation hardware in GPUs can possibly contribute to a better image quality, and show its effect on performance.

This paper is structured as follows. In Section 2, we explain the basics of imaging radio telescope data. Then, in Section 3, we elaborate on related work. Section 4 explains the new strategy. In Section 5, we briefly discuss our prototype implementation, and in Section 6, we evaluate the per-


*ICS'12,* June 25–29, 2012, San Servolo Island, Venice, Italy.

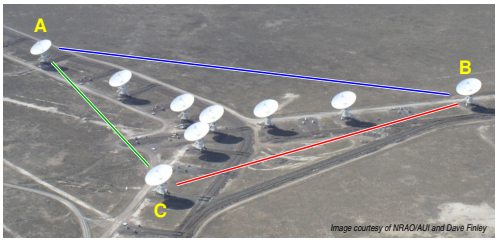


**Figure 1: Three (out of 36) baselines between nine telescopes.**

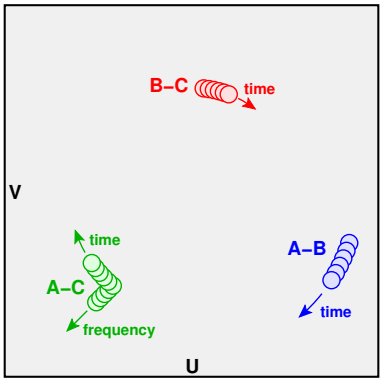


**Figure 2: Visibilities from consecutive times and frequencies are placed onto the UV-grid.**

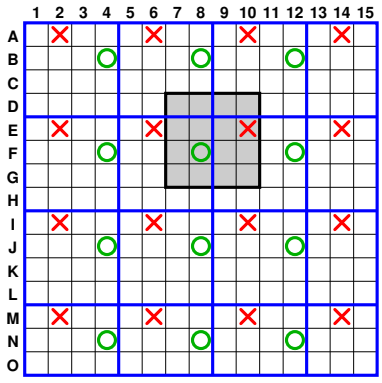


**Figure 3: UV-grid divided into subgrids.**

formance of the new strategy on multiple platforms, compare with other accelerator-based gridders, show multi-GPU scaling characteristics, and demonstrate that this form of computing is highly "green." Section 7 discusses future work and Section 8 concludes.

## 2. IMAGING RADIO TELESCOPE DATA

To increase the sensitivity and resolution of images, telescopes often combine data from multiple antennas. Each antenna samples the electromagnetic spectrum at a high rate. The samples are digitized and converted to complex numbers that represent the phase and amplitude of the signal(s) that come from the observed source(s). The data from multiple antennas are correlated by multiplying the samples of each *pair* of antennas. These products are integrated over some time interval, to keep the output data rate manageable. Figure 1 shows a few telescope pairs, which we call *baselines*.

The integrated product of the samples of an antenna pair is called a *visibility*. Each visibility has an associated $(u,v,w)$ coordinate, which depends on the position of the antennas, on the position of the observed source, on the frequency of the observed signal, and on the time. The $(u,v,w)$ coordinates for an antenna pair changes over time due to rotation of the earth, that alters the antenna's positions with respect to the observed source. After correlation, the visibilities undergo some processing (removal of interference, calibration, etc.) to improve data quality.

Since visibilities are sampled in the Fourier domain, they are placed on a $UV$-grid. A final two-dimensional FFT then converts the UV-image to a sky-image. Placement of the visibilities on the UV-grid is the topic of this paper.

A visibility is placed onto the UV-grid using its $u$ and $v$ coordinates. However, the visibility does not contribute to a single grid point, but to neighboring grid points as well. This contribution is computed by *convolving* the visibility with a convolution matrix, i.e., by multiplying the (complex) visibility with each of the (complex) weights in the matrix, and by adding the result to the grid. How the contents of a convolution matrix are obtained, is far beyond the scope of this paper; for interested readers, we refer to [8]. Here, we consider the convolution matrix as a precomputed matrix of complex weights.

Figure 2 shows how visibilities from the three antenna pairs from Figure 1 are placed onto a UV-grid. Consecutive visibilities (in time) from one baseline and one frequency are placed in an elliptic curve over the grid. Visibilities from higher frequencies and larger baselines follow ellipses with larger diameters. Telescope configurations and observation times are typically chosen so that the UV grid is optimally covered with visibility data, to get the best image quality.

For a particular baseline, the convolution matrix slides slowly in time and frequency over the grid. This movement is not too fast, otherwise the visibility would be smeared over a too large area in the UV-grid, reducing image quality. The movement is also not too slow, otherwise the visibilities could have been integrated over a larger time and/or over more frequencies earlier on in the processing pipeline, reducing the data rate and processing time. The speed of the movement depends largely on the baseline length and the grid size, but it generally takes tens of visibilities (in time) to move one grid point away. After (almost) one day of observing, the ellipse is completed, but it is perfectly possible to generate images from shorter observations.

When creating wide-field images, we cannot treat the spheroidal form of the earth and the observed part of the sky as flat planes. In this case, the $w$ coordinate (in the third dimension) is non-zero, and we use a technique called *wide-field imaging*. Using a single convolution matrix is not sufficient then. The *W-projection* algorithm [2] uses different convolution matrices for different values of $w$. Typically, the W-dimension is partitioned into several tens of W-planes, with different convolution matrices for each W-plane.

Additionally, the W-projection algorithm increases precision in the U and V directions as well. Since the $(u,v,w)$ coordinates of a visibility are floating-point numbers with non-zero fractional parts, the convolved visibility cannot be added exactly at grid points with integer U and V coordinates. To increase accuracy, the W-projection algorithm uses multiple convolution matrices (typically, $8 \times 8$ per W-plane) for different fractional parts of $u$ and $v$. For example, there is a convolution matrix for fractional parts (.0,.0), one for (.0,.125), one for (.375,.625), etc. The convolution matrix that is the closest one to the fractional parts of $u$ and $v$ is then used. The W-projection algorithm increases accuracy by *oversampling* the convolution function when creating the $8 \times 8$ convolution matrices. All convolution weights together

```
FOR bl IN baselines DO
  FOR time IN times DO
    FOR chan IN channels DO
      (u,v,w) = getUVWcoordinates(bl, time, chan)
      overSampU = int(8 * frac(u)) // oversampling: use most appropriate convolution matrix
      overSampV = int(8 * frac(v))
      FOR convV IN 0 TO convSize DO
        FOR convU IN 0 TO convSize DO
          weight = convFuncs[int(w)][overSampV][overSampU][convV][convU]
          FOR pol IN {XX,XY,YX,YY} DO
            grid[int(v) + convV][int(u) + convU][pol] += visibilities[time][bl][chan][pol] * weight
```

**Algorithm 1: Core of the W-projection algorithm.**

form a five-dimensional array, indexed by $w$, the fractional parts of $u$ and $v$, and the two coordinates within the convolution matrix.

Typically, telescopes sample the electromagnetic spectrum in two orthogonal polarizations, X and Y. The correlator cross-correlates these polarizations, so that visibilities come in quadruples: XX, XY, YX, and YY. In fact, we create four images from these visibilities, one for each polarization. Each group of four visibilities has the same ($u$,$v$,$w$) coordinates and is convolved using the same convolution matrix, but the results are placed onto different grids.

The W-projection algorithm is summarized in Algorithm 1. It iterates over baselines, times, and frequency channels, looks up the ($u$,$v$,$w$) coordinates of the current four visibilities, determines the most appropriate convolution function, and multiplies the four visibilities with the convolution matrix, and adds the result to the four grids.

# 3. RELATED WORK

Convolutions are commonly used to implement generic image operations like blurring and edge detection. Here, each pixel of an output image is the sum of weighted neighboring pixels from the input image; the weights are stored in what is called a "mask" or "filter". Generic image-processing convolutions on GPUs (or other many-core hardware) have been studied extensively (for example, [1, 7]), and are commonly used in tutorials on GPU programming (e.g., a sample implementation is distributed in the AMD OpenCL SDK).

However, gridding radio-telescope data is different from generic image convolutions, in the sense that we convolve samples rather than images, that the access patterns are different and less predictable, and that creating a sky image is computationally much more expensive. The output is an image, though (in the Fourier domain). The amount of literature contributions on accelerated radio-telescope convolutions is much smaller. Below, we elaborate on four studies that are related to our work.

The GPU gridder that is being developed for the Murchison Widefield Array (MWA) is one of them. Edgar et. al. [5] describe how visibilities for the MWA are gridded using a gridder written in CUDA. They recognized that actively adding a convolved visibility to a subset of the UV grid is not thread safe on the hardware they use — and if it were, adding convolved visibilities directly to the grid would require a prohibitive amount of (atomic) device-memory accesses. Therefore, their approach is to associate each grid point with a CUDA thread, and search, for all grid points, the visibilities that contribute to a grid point. Since there are, in their case, on average only 60 out of 130,816 visibilities that do contribute something to a grid point, in its basic form, each thread would waste an enormous amount of time searching for the visibilities of interest. Hence, they sort the visibilities according to their ($u$,$v$) coordinates and put them into bins, so that a thread only needs to search nine bins (the "home" bin, plus eight neighboring bins) for the visibilities of interest. However, eight of out of nine searched visibilities will still not contribute to the thread's grid point, but do add to the overhead. The work-distribution strategy presented in this paper, which will be described in the next section, neither needs sorting of visibilities, nor needs searching for visibilities, while keeping the amount of accesses to device memory low.

Varbanescu et. al. [11, 12] implemented the W-projection algorithm on the Cell BE processor. The Cell BE is a many-core processor, but its architecture is rather different from the GPU architectures from AMD and Nvidia. A Cell BE processor essentially consists of a PowerPC CPU, and is assisted by eight vector coprocessors, called SPUs, that run their own code. Asynchronous DMA transfers between CPU and SPU memory are explicitly programmed, and aligned multiples of 128 bytes must be transferred to achieve high bandwidth. Also, the SPUs are (four-word) vector processors that can only load/store efficiently if the words are contiguous and aligned in memory. The nature of Cell BE architecture led to a work-distribution strategy where the parallelism is rather coarse grained: an SPU DMAs a convolution matrix and the relevant part of the grid to its local memory, convolves the visibility and adds it to its partial grid copy, and DMAs the new grid copy back to main memory. They implemented many optimizations: standard optimizations like triple buffering, but also application-dependent optimizations that try to improve locality, so that fewer DMAs between host memory and SPU memory are necessary. Their strategy also (partially) sorts visibilities according to their ($u$,$v$,$w$) coordinates and searches for visibilities that contribute to particular grid points. This is done in a way that the benefits for improved locality outweigh the computational costs for sorting and searching.

The W-projection algorithm was also ported, optimized, and benchmarked on GPUs by van Amesfoort et. al. [9]. Apart from the above-mentioned optimizations to improve locality, they focussed on maximizing the obtained device-memory bandwidth. To avoid race conditions, caused by multiple thread blocks updating device memory concurrently, they gave each thread block a private grid in device memory. Unfortunately, with the memory sizes of present-day GPUs, this implementation limits the grid sizes to very low-resolution images, so this method is not usable for telescopes like LOFAR and the SKA.

Humphreys and Cornwell describe a GPU gridder that is optimized to achieve maximum device memory bandwidth [6]. As with the previously mentioned gridder, this gridder also

adds data directly to device memory. Likewise, their gridder is fully memory-bandwidth bound.

# 4. THE GPU-OPTIMIZED WORK-DISTRIBUTION STRATEGY

We now present a new work-distribution strategy that efficiently convolves and grids the visibility data on the UV-grid, without the necessity to sort or search visibilities, while keeping the number of expensive device-memory accesses very low. The basic idea is to accumulate data in registers rather than in device memory. Unfortunately, this can only be achieved using an unintuitive and complex work-distribution strategy.

The strategy works as follows. We decompose the grid into subgrids that have the same size as the convolution matrix. In the example of Figure 3, we use a $15 \times 15$ grid, and a $4 \times 4$ convolution matrix; in reality, both are much larger. We also create a number of threads that, for the time being, equals the number of grid points in a subgrid. Conceptually, each thread "monitors" a large number of grid points; one fixed grid point per subgrid. In this example, we create 16 threads, where one of the threads monitors all grid points marked X; another thread monitors all grid points marked O.

The convolution matrix (the gray-shaded area in the figure) slides slowly over the grid. An important insight is that each thread always monitors exactly one grid point covered by the convolution matrix, no more, no less. Consider, for example, the grid point F8, which is monitored by the "O" thread. As the convolution matrix slides, say, to the left, the thread monitors the same grid point until the matrix hits line 4. At this time, the convolution matrix slides off F8, and the O thread switches to grid point F4.

Since the convolution matrix slides slowly, a thread does not often switch to another grid point. An important consequence is that it can accumulate multiple updates (additions) to a grid point locally in registers. Only when the thread switches to another grid point, it (atomically) adds its local sum to the grid point value that resides in device memory. This significantly reduces the number of device memory accesses; a $n \times n$ convolution matrix that slides one grid point away updates only $n$ out of $n^2$ grid points in device memory.

Algorithm 2 shows simplified pseudo code for the new algorithm. This kernel is invoked for many threads concurrently, where each thread is invoked with different values for *myBL* (myBaseLine), *myU*, and *myV*, the latter two having values between zero and *convSize* - 1. The kernel initializes four complex accumulators (kept in registers), and iterates over a series of times and frequencies. It computes its convolution function indices and grid coordinates, and checks if the grid coordinates have changed since the previous time. Usually, they are still the same, but sometimes, the thread switches to another grid point an adds its local sums to the grid — atomically, since threads that process different baselines might update the same grid point simultaneously. Then, the thread multiplies the four visibilities with its convolution weight, and adds its them to its local sums. It repeats this, until the visibilities for all times and frequency channels have been processed. Finally, the local sums are once added to the grid.

```
KERNEL convolve(..., myBL, myU, myV) IS
sumXX = sumXY = sumYX = sumYY = (0,0)
prevGridU = prevGridV = 0

FOR time IN times DO
  FOR chan IN channels DO
    (u,v,w) = getUVWcoordinates(myBL, time, chan)
    overSampU = int(8 * frac(u))
    overSampV = int(8 * frac(v))
    myConvU = (int(u) — myU) % convSize // unsigned mod
    myConvV = (int(v) — myV) % convSize
    myGridU = int(u) + myConvU
    myGridV = int(v) + myConvV

    IF prevGridV != myGridV OR prevGridU != myGridV THEN
      atomicAdd(grid[prevGridV][prevGridU][XX], sumXX)
      atomicAdd(grid[prevGridV][prevGridU][XY], sumXY)
      atomicAdd(grid[prevGridV][prevGridU][YX], sumYX)
      atomicAdd(grid[prevGridV][prevGridU][YY], sumYY)
      prevGridU = myGridU, prevGridV = myGridV
      sumXX = sumXY = sumYX = sumYY = (0,0)
    END IF

    weight = convFuncs[int(w)][overSampV][overSampU]...
                                        ...[myConvV][myConvU]
    sumXX += visibilities[time][myBL][chan][XX] * weight
    sumXY += visibilities[time][myBL][chan][XY] * weight
    sumYX += visibilities[time][myBL][chan][YX] * weight
    sumYY += visibilities[time][myBL][chan][YY] * weight
  END FOR
END FOR

atomicAdd(grid[prevGridV][prevGridU][XX], sumXX)
atomicAdd(grid[prevGridV][prevGridU][XY], sumXY)
atomicAdd(grid[prevGridV][prevGridU][YX], sumYX)
atomicAdd(grid[prevGridV][prevGridU][YY], sumYY)
```

**Algorithm 2: Memory-bandwidth reduced W-projection.**

Algorithm 2 illustrates how the amount of memory accesses can be reduced, but it can be improved further. Our GPU implementation prefetches visibilities and UVW coordinates from device memory to fast, shared (local) memory, and precomputes some array indices. Also, we removed the expensive modulo operation from the inner loop, and replaced it by a conditional add.

For small convolution matrices, there is one thread per convolution matrix point. For large convolution functions, we let each thread perform the work for multiple convolution matrix points, because the maximum number of threads per thread block is typically in the 256–1024 range.

The grid is conceptually divided into bins that have the same size as the convolution matrix. The W-projection algorithm allows smaller convolution matrices for short baselines, reducing the amount of computations. Our strategy supports this. The visibilities for different baselines can be gridded independently of each other, and the conceptual division of the grid into subgrids can be different for each baseline.

## 4.1 Interpolation

Since convolved visibilities must be placed at grid points with integer coordinates, the W-projection algorithm picks the most suitable convolution matrix from a large set, depending on the fractional parts of the $u$ and $v$ coordinates, and on the $w$ coordinate. On GPUs, it seems attractive to take another approach, since the texture units have special-purpose hardware to quickly interpolate values in a one, two, or three-dimensional texture. Instead of using a five-dimensional array to store all convolution weights, we create a three-dimensional texture, organized as a stack of (two-

dimensional) convolution functions. Each plane in this stack describes the convolution function for a particular value of $w$. This way, we create a 3D-texture, that describes the convolution function, for all values of $w$. By using floating-point indices, the convolution function can be sampled everywhere, using interpolation. This way, we use the fractional parts of the ($u$,$v$,$w$) coordinates to place the convolved visibilities at non-integer grid points. The size of the 3D-texture can be different from the size of the convolved visibility matrix that is added to the grid, since the texture indices can be scaled. A larger texture is more accurate, but causes many misses in the texture cache, especially if the texture is sparsely sampled.

The pseudo codes in Algorithm 1 and Algorithm 2 are slightly modified to allow interpolation. Instead of using *overSampU* and *overSampV* to read *convFuncs*, we call a function *interpolateConvFunc(u,v,w)* that linearly interpolates the eight nearest points in the cube with convolution values, using the floating-point coordinates $u$, $v$, and $w$.

Using a 3D-texture potentially leads to a higher image quality (e.g., with a higher dynamic range), something that we did not yet investigate. Additionally, it is likely that the texture can be significantly smaller than with the classic W-projection algorithm; we think that fewer W-planes are needed, and that the oversampling factors in U and V directions can be much lower than $8 \times 8$, because interpolation and oversampling are both techniques that improve accuracy by taking the fractional parts of the ($u$,$v$,$w$) coordinates into account.

# 5. IMPLEMENTATION DETAILS

We first wrote a reference implementation for the classic W-projection algorithm and the interpolation algorithm in C++. It follows the ideas from a reference implementation by Tim Cornwell, but our implementation is highly optimized: it is multi-threaded and uses AVX vector intrinsics. These eight-word vector instructions are used to efficiently convolve the four complex numbers from the four polarizations in parallel. This way, we compute four complex multiply-adds with four arithmetic AVX instructions. Three additional AVX shuffle instructions are necessary to permute the real and imaginary operands, and one more AVX move instruction stores the result.

The reference implementation still uses the old idea to add the convolved visibility directly to the grid in main memory, and heavily relies on the memory cache to cache the parts of a grid that are actively being added to. To attain high cache-hit ratios, it is of importance to reduce the working set size to something that fits in the L1 cache. This can be achieved by moving the "convV" loop in Algorithm 1 two levels up. Not applying this optimization results in a performance penalty of up to a factor of 18.5. It is important to note that this optimization cannot be applied to GPUs: the large amount of active threads write to many more different locations in main memory than can be cached.

To test our new strategy for the W-projection algorithm on GPUs, we implemented a prototype gridder in both CUDA and OpenCL. The CUDA implementation runs on Nvidia GPUs only. The OpenCL implementation runs on all CPUs and GPUs that support OpenCL. The code that runs on the hosts uses the OpenCL C++ bindings with exception support, which leads to much more concise code than the C bindings.

The convolution matrices are either stored as texture (in OpenCL terminology: image), or a normal array. We only use textures on platforms that support them, and when this actually improves performance. The classic W-projection algorithm does not interpolate the texture.

We distribute the work over the threads (work items) and thread blocks (work groups) as follows. Since the visibilities for different baselines are typically placed on different parts of the grid, we create one thread block per baseline, which are independently processed by the multiprocessors of the GPU. The threads within the thread block then process the visibilities of a number of frequency channels, timesteps, and all polarizations, for a single baseline. This way, we maximize register reuse due to locality on the grid. The kernels transfer visibility and UVW data from mapped host memory through the PCIe bus. Since this data is reused by multiple threads, each kernel first stores the data in shared (local) memory, synchronizes all threads (within the multi-processor), and then performs the convolution computations, This way, the threads have quick access to visibility and UVW data.

# 6. PERFORMANCE RESULTS

We measured the performance of our new strategy on the following combinations of hardware, programming languages, and platform vendors:

| hardware | platform & language | peak GFLOPS | peak GB/s | power Watt |
|---|---|---|---|---|
| Nvidia GTX 680 | Nvidia CUDA | 3090 | 192 | 195 |
| Nvidia GTX 680 | Nvidia OpenCL | 3090 | 192 | 195 |
| AMD HD 7970 | AMD OpenCL | 3789 | 264 | 230 |
| 2× Intel E5-2680 | AMD OpenCL | 343 | 102 | 260 |
| 2× Intel E5-2680 | Intel OpenCL | 343 | 102 | 260 |
| 2× Intel E5-2680 | Intel C++ | 343 | 102 | 260 |

These are the latest high-end CPUs and GPUs available, manufactured using comparable technologies (28–32 nm).

We used real ($u$,$v$,$w$) coordinates from a six-hour LOFAR observation with 44 antennas (946 baselines; 10 s. integration time, and one subband of 16 frequency channels). A full observation consists of hundreds of subbands, thus the amount of time to grid an entire observation would be several hundreds of times higher than the execution times mentioned below.

## 6.1 Performance measurements of the W-projection algorithm

We first measured the performance of our strategy for the W-projection algorithm. On the X-axes of Figures 4, 5, and 7, we vary the convolution matrix size. Although the implementation supports baseline-dependent convolution matrix sizes, we use fixed sizes for all performance measurements, to better understand the performance results. All performance measurements were done using a quad polarized, $2048 \times 2048$ grid. We use a fixed $8 \times 8$ oversampling rate and 32 W-planes, because the execution times hardly depend on these parameters, while these parameters are supported by all platforms.

Figure 4 shows the performance of our prototype implementations. The left graph shows the gridding execution times for one subband of the six-hour observation. The reasons for the large differences in execution times for the different platforms will be explained in the remainder of this section. The slopes in the curves are due to the (quadrati-

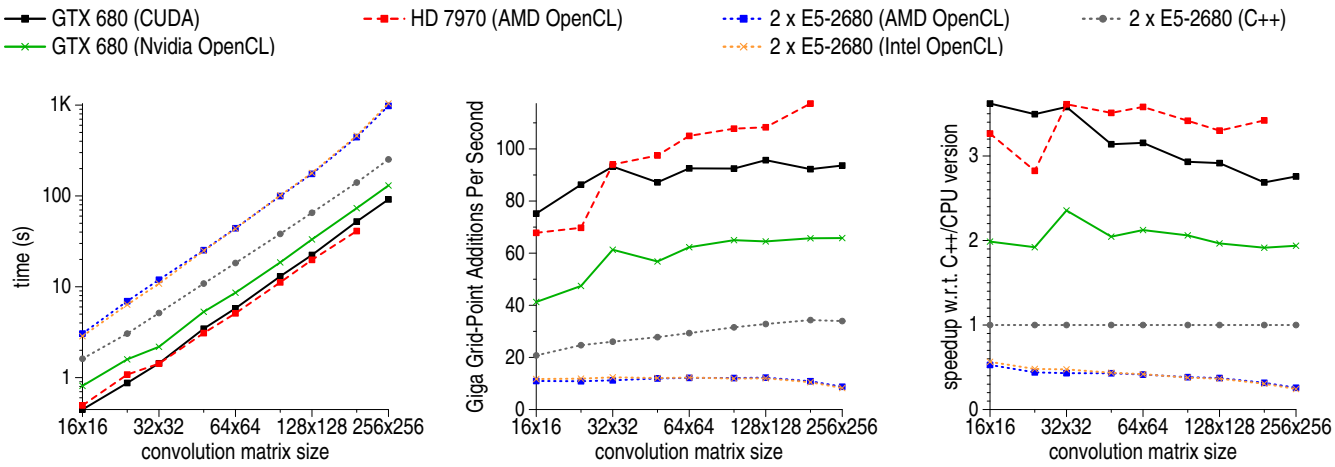


**Figure 4: Performance of our new strategy for the W-projection algorithm.**

cally) increased amount of work that is involved with larger convolution matrix sizes.

The middle graph of Figure 4 demonstrates the efficiency of our new work-distribution strategy. We express the efficiency in terms of *Giga Grid-Point Additions Per Second* (GGPAPS). Note that GGPAPS counts the number of updates in registers, not the number of updates to grid points in device memory. The number of "useful" GFLOPS per second is eight times the number of GGPAPS, since a complex multiply-add costs four real multiplications and four real additions (and maps well to fused multiply-add instructions).

### 6.1.1 The Nvidia GeForce GTX 680

We will first look at the performance of the CUDA version on a GTX 680. This is the only version for which the use of textures reduced the execution times; the benefits from the additional texture cache outweigh the overhead from extra instructions to do a texture lookup. Thus for this analysis, we enabled textures. We used the Nvidia visual profiler to study the behavior of our application.

For medium and large-size convolution matrices, it updates up to 93.6 billion grid points per second. This equals to 749 GFLOPS (24.2% of the theoretical peak performance), excluding all kinds of overhead. These overheads are substantial: for $256 \times 256$ convolutions, only 37% of all executed instructions are FPU instructions (fused multiply-adds) that operate on the visibility data. Most instructions (41%) are integer instructions used as loop variable or array index, the remainder are shared memory loads (7%), texture lookups (4.7%), comparisons that set predicates (4.7%), branch instructions (4.7%) and a few miscellaneous instructions. Only 0.094% of the executed instructions are atomic memory additions.

The new strategy successfully reduces the amount of device memory accesses: only 0.23% of all grid point updates need access to device memory. The measured device memory bandwidth is 53.0 GB/s (out of a maximum of 192 GB/s), of which 19.8 GB/s is due to grid point updates and 33.2 GB/s due to texture cache misses. The operational intensity (i.e., the amount of arithmetic operations per byte transferred to/from device memory) is 14.1 FLOPS/byte, which is on par with the peak GFLOPS/peak bandwidth (16.1 FLOPS/byte). The costs of transferring visibilities and UVW coordinates from host memory through the PCIe bus to the GPU card are negligible, due to good overlap between computations and communication. A 87.2% texture hit rate is sufficient. The achieved occupancy is high (0.952): each multiprocessor runs two blocks of 1024 threads concurrently, using the full register file and nearly all shared memory.

Unfortunately, the profiler does not point us at the real bottleneck: the application seems neither compute bound, nor memory bound, or limited by PCIe bandwidth. The real culprit is the atomicity of the global memory additions, and the atomic aspect of this update is not covered by the profiler. Even though only 0.23% of all grid point updates result in an atomic memory update, the costs of these occasional updates are high: the atomic nature of these additions is responsible for 26% of the total run time. To remove the atomic nature of these updates, we could use a private grid per active block of compute threads if the device memory were somewhat larger (3–7 GB, depending on the convolution function size, assuming a $2048 \times 2048$ grid), but current GTX 680s are limited to 2 GB in size.

For the smallest convolution matrix sizes, the performance is also good, almost as good as for medium and large convolution matrices. This configuration requires quite different tuning parameters, though. With $16 \times 16$ matrices, each block has 256 threads, and we run 6 blocks per multiprocessor concurrently, yielding a measured occupancy of 0.694 (the theoretical maximum occupancy is 0.75). Increasing the number of concurrent blocks to 8 per multi-processor (for a theoretical occupancy of 1.00) did not improve performance anymore.

On the same GTX 680, the OpenCL implementation is clearly slower than the CUDA implementation. One reason for this is the fact that an atomic floating-point addition to device memory is natively supported in CUDA, and has to be implemented using an atomic compare-and-swap primitive in OpenCL, which must be repeated until it succeeds. With CUDA, an atomic floating-point addition is translated into a single, predicated, atomic add instruction in the binary executable of the GTX 680, while the binary executable of the OpenCL implementation requires seven instructions, including an even more expensive compare-and-swap. The performance impact is large: up to 55% of the performance

difference between CUDA and OpenCL is caused by the absence of an atomic floating-point add in OpenCL. Unfortunately, even the new OpenCL 1.2 specification does not mention this as an extension. The remainder of the performance difference is explained by the fact that for these measurements, we did not use images (textures), as doing so increases the total runtime. The CUDA version uses 1D textures, but the current OpenCL 1.1 specification only supports 2D and 3D images. The benefits of using the texture cache does not outweigh the additional overhead of indexing a multi-dimensional image. OpenCL 1.2 will allow 1D images.

### 6.1.2 The AMD Radeon HD 7970

We will now discuss the results from the AMD HD 7970 GPU. In most cases, it is the fastest GPU, with a maximum performance of 117.4 GGPAPS. This is not surprising, since the HD 7970 has a 23% higher FPU peak performance, 37% more memory bandwidth, and an 18% higher maximum power consumption than GTX 680. However, for small convolution functions, the HD 7970 performs worse than the GTX 680. One reason for this is that the AMD OpenCL runtime does not (yet) overlap I/O and computations, even though we submit alternating I/O and compute commands to multiple queues (by multiple host threads). We saw this non-overlapping behavior in other applications as well. In contrast, the Nvidia runtime optimally overlaps I/O and computations on the GTX 680. If communication would have fully overlapped on the HD 7970, it would have achieved 79.3 GGPAPS, slightly more than the GTX 680. Fortunately, we found that the run time on the HD 7970 could be improved by mapping host memory into the GPU address space, letting the GPU cores read host memory. The other way around, mapping device memory into the host address space (an AMD extension), did not work for anything but impractically small buffers. The graph in Figure 4 shows the best obtained performance, hence for the version that maps host memory into the GPU address space.

A second cause for the lower performance of the HD 7970 on small convolution functions is the larger impact of the absence of support for native atomic floating-point additions. On this device, it is hard to estimate the performance if it would have supported atomic additions, but we see that the impact is high if we replace the atomic compare-and-swaps by non-atomic additions (yielding wrong results). The performance then increases from 67.8 to 84.9 GGPAPS for $16 \times 16$ convolutions. If I/O would also overlap with computations, the performance would further increase, up to 119.8 GGPAPS. Still, the new Graphic Core Next architecture used by the HD 7970 is a major leap forward over AMD's previous architecture: the HD 7970 runs our application between 5.0 and 5.6 times as fast as the older HD 6970. The gap between Nvidia's current Kepler architecture and its previous Fermi architecture is much narrower; the GTX 680 is 1.18–1.66 times as fast as the GTX 580, though it also reduces power consumption by ~25%.

### 6.1.3 The dual Intel Xeon E5-2680

On a dual Xeon E5-2680 CPU, our C++/AVX implementation runs highly efficiently. The performance varies between 20.8 and 33.9 GGPAPS. This corresponds to 48–79% of the FPU peak performance. The hand-written AVX intrinsics improved performance by a factor 2.3–3.4 times, compared to compiler-vectorized code from the intel compiler. Thus, for CPUs, the old idea of adding convolved visibilities directly to the grid is not a bad idea, *provided that* the application uses hand-written AVX intrinsics and is optimized to restrict the working set size to something that fits in the L1 cache. Again, the latter cannot be done on a GPU, because there are too many threads in flight for the amount of cache that is available.

On a CPU, the OpenCL version is slower than the C++/AVX implementation. The AMD OpenCL compiler generates 4-word vector operations from the gridding kernel, as it does not see how it could generate 8-word vector operations. Without using 8-word vector operations, there is no way to keep up with the hand-written C++/AVX implementation. It does, however, accumulate grid updates in vector registers. A small penalty is paid for the use of atomic compare-and-swap instructions, but the penalty is at most 6%, lower than on the GPUs.

Intel's OpenCL runtime system performs similar to the OpenCL runtime from AMD, but only after turning off auto-vectorization: the auto-vectorizer creates code that aggressively spills vector registers to the stack, roughly doubling instead of reducing the execution times. Without auto-vectorization, the compiler does not attempt to collapse operations from multiple work items into a single vector instruction, but the compiler still can emit vector instructions, e.g., from float4 operations in a *single* work item. With auto-vectorization turned off, the performance is on par with the AMD OpenCL runtime, which is not surprising, because the generated code is highly similar.

### 6.1.4 CPUs vs. GPUs

Figure 4 (right) shows speedups with respect to the C++/AVX implementation on a dual Xeon E5-2680 CPU. The GPUs are 2.7–3.6 times faster than a pair of almost the fastest general-purpose CPUs currently available. The different architectures require different approaches. To obtain good performance on CPUs, one can rely on vector instructions and efficient caches. To obtain good performance on GPUs, one *must* reduce memory bandwidth consumption.

## 6.2 Comparison with other accelerator-based gridders

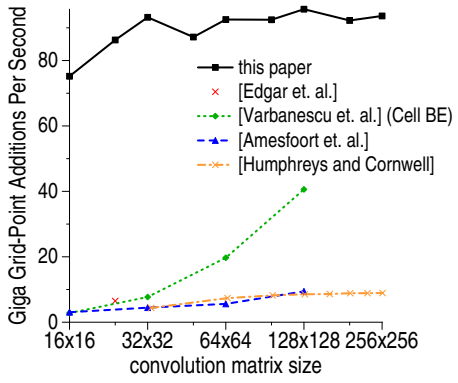


**Figure 5: Comparison with other accelerator-based gridders.**

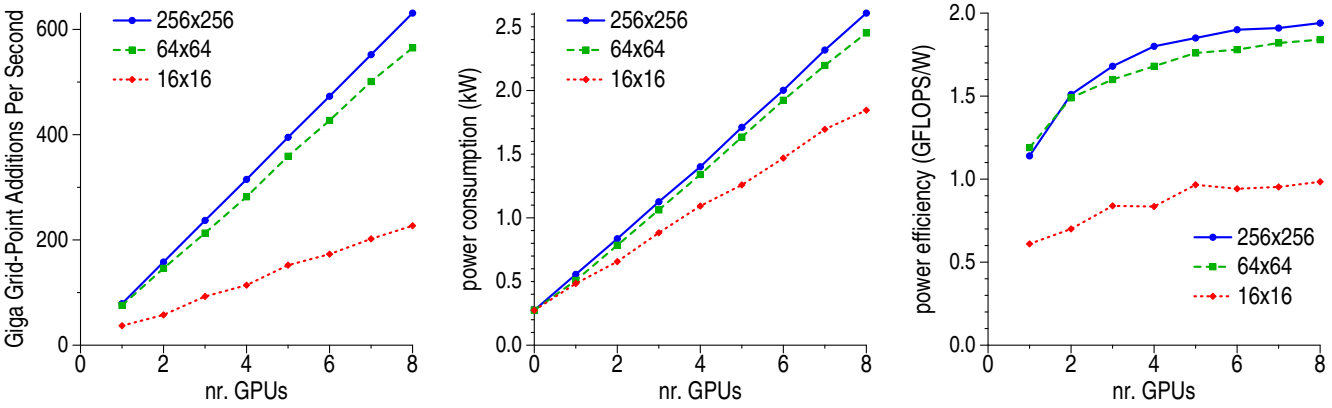


**Figure 6: Multi-GPU scaling performance (left), power consumption (middle), and power efficiency (right) for W-projection gridding of different convolution matrix sizes, up to eight GTX 580 GPUs.**

Below, we compare our results to those from all other accelerator-based gridders that we are aware of. Figure 5 graphically shows the performance of published results, multiplied with the estimated difference in hardware performance between the GTX 680 that we used and the hardware that the others used.[1]

The MWA gridder achieves 2.3 GGPAPS (130,186 baselines, 12 channels, a $24 \times 24$ convolution matrix, four polarizations in 1.57 seconds), using a Tesla C1060 GPU [5]. For the same size convolution matrix, our strategy runs 37.5 times faster (86.3 GGPAPS) on hardware that has 1.9 times the memory bandwidth. It must be noted, however, that the Tesla C1060 does not support atomic floating-point additions to device memory; to run our strategy on a Tesla C1060, a slower atomic compare-and-swap instruction must be used. On the other hand, their imager neither uses multiple W-planes, nor does it do oversampling, and this simplifies convolution matrix index calculations and allows the convolution matrix to fit entirely in the texture cache. Under these conditions (thus using atomic compare-and-swaps, a single convolution matrix, and no oversampling), our gridder still achieves 70.0 GGPAPS on a GTX 680.

The performance of the W-projection algorithm was also studied on a dual Cell BE by Varbanescu et. al. [11, 12]. They achieve 0.50–7.2 GGPAPS on hardware that has a quarter of the memory bandwidth. Unlike our work-distribution scheme, they sort visibilities to improve locality and search for visibilities contributing to a grid point. Obtaining good performance on the Cell BE is hard, because the SPUs cannot add values directly to main memory, but have to cache active parts of the grid in their local stores. Even though this architecture is extremely efficient in computing correlations [10], we think it is less suitable for gridding. Also, cache consistency has to be maintained in software by means of explicit DMAs, which places a notorious burden on the programmer.

[1]As the application is memory-I/O bound, we estimate the difference in hardware speed by dividing the peak memory bandwidth of the GTX 680 by the peak memory bandwidths of the devices used by others, plus a 50% safety margin in the advantage of the others, to make sure that we do not underestimate their performance.

Van Amesfoort et. al. [9] achieve 1.5–4.6 GGPAPS on a GTX 280, depending on the convolution matrix size. Corrected for the difference in hardware speeds, our work-distribution strategy is more than an order of magnitude faster. Also, our strategy allows grids that are at least $10 \times 10$ larger in size.

The gridder by Humphreys and Cornwell [6] is very much optimized to achieve maximum device memory bandwidth. However, since the number of device memory accesses is not reduced, their implementation peaks at 3.6 GGPAPS on hardware that has 1.67 times less bandwidth (a Tesla C2070 with ECC enabled). Again, the performance difference on comparable hardware is at least a factor of 12.5.

Although the other works were valuable early research contributions on accelerator-based gridding, our new strategy is convincingly faster than any other accelerated gridder. The difference is typically an order of magnitude.

## 6.3 Multi-GPU scaling and energy efficiency

We also studied the scaling behavior of this strategy for multiple GPUs in a single system, and determined the power efficiency. As we had only one GTX 680 (this architecture has just been released at the time of writing), we used a Tyan B7015 system with eight GTX 580 GPUs. Compared to the GTX 680, the GTX 580 has roughly half the computational power, the same memory bandwidth, and a 26% higher thermal design power. In practice, on the GTX 580 our application runs 40% slower for small convolution functions and 15–20% slower for large and medium sized convolution functions than on a GTX 680.

For these measurements, we used the CUDA implementation of the W-projection algorithm. We divided the work over the GPUs by partitioning the data set in time. This distribution is trivially parallel, thus these chunks are processed independently. Only at the very end of a run, the private GPU grids are transferred to host memory and added together.

On the B7015 system, the connections between CPU cores, host memories, I/O hubs, and GPUs involve multiple PCIe x16 V2.0 busses, PCIe switches, and Quick-Path Interface (QPI) links, which are non-uniform. To minimize contention,

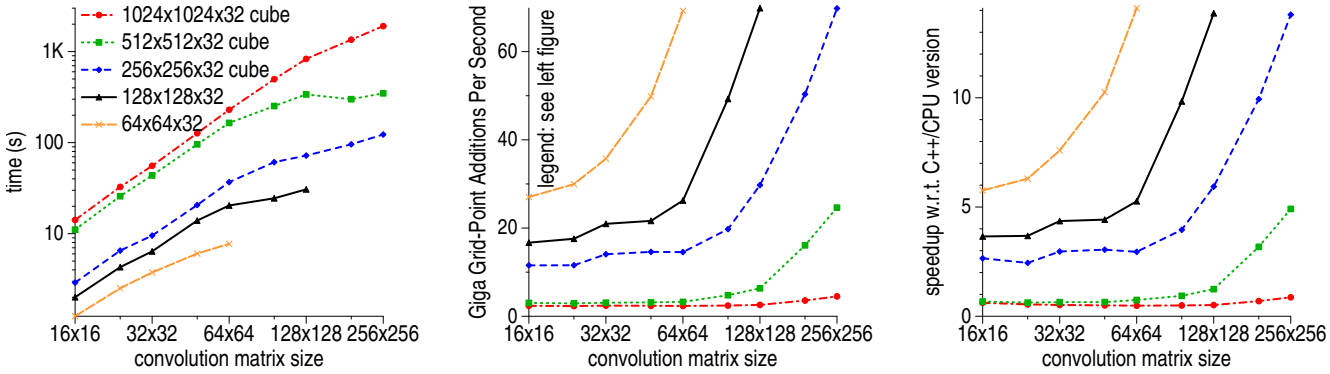


**Figure 7: Performance of the interpolation algorithm on a GTX 680 GPU.**

our prototype application allocates host memory on the CPU that is physically closest to the GPU.

With eight GPUs, the application runs no less than 131,072 threads concurrently in a single system! Figure 6 (left) shows the achieved amounts of GGPAPS for convolution matrices of different sizes, for up to eight GPUs. Except for very small convolution matrices, the speedups are perfect. Scaling for small matrices is still good, but with multiple GPUs and shared busses, there is some contention on the PCIe busses when the GPUs fetch visibilities and UVW coordinates from host memory.

A metered power-distribution unit accurately monitors the instantaneous voltage, current, and power factor of the whole system (including the power supply units). For runs with fewer than eight GPUs, we correct for the idle current of unused GPUs, but the power consumption of the rest of the system is included. Figure 6 (middle) shows that the system draws no less than 2.6 kW under heavy load. Still, the algorithm is highly energy-efficient on this platform, as the achieved amount of useful GFLOPS/Watt is as high as 1.94 (515 pJ/FLOP). For comparison: the C++ implementation on a dual E5-2680 CPU achieves at most 792 MFLOPS/Watt (1.26 nJ/FLOP), while the GTX 580 is from a 1.5 year older generation. We estimate the power efficiency of eight current-generation GTX 680s to be around 2.8 GFLOPS/Watt (360 pJ/FLOP). The middle graph also shows that convolving visibilities with a very small $(16 \times 16)$ matrix, where the amount of GFLOPS is lower than when convolving with large matrices, results in lower energy consumption. The power efficiency (right figure) is high, but not as high as for medium and large matrices.

## 6.4 Performance measurements for the interpolation algorithm

We finally measured the performance of the advanced gridder, that interpolates the convolution function weights that are stored in a 3D-texture. We only implemented the interpolation algorithm in CUDA, but there is no reason why it could not be implemented in OpenCL. OpenCL supports the use of hardware interpolation, but it would suffer from the absence of atomic floating-point additions to device memory. Figure 7 shows performance metrics on a GTX 680, for different texture sizes. For these measurements, we assume that the convolution function is symmetric in the U and V directions, reducing the texture size by a factor of four.

The performance depends very much on the ratio between the sizes of the convolution matrix and the texture. When the convolution matrix is much smaller than the texture, the execution times rapidly increase. In that case, the texture is sparsely indexed, and the texture cache is less effective, because words read from the texture are not reused to compute the values of surrounding convolution matrix weights. However, if the convolution matrix size approaches the texture size, the convolution matrix weights are interpolated from texture entries that are at least partially cached in the texture cache. In that case, the performance is much closer to the performance of the standard W-projection algorithm.

The speedup with respect to the CPU version also depends much on the texture size and on the convolution matrix size, but is in some cases much higher than that of the W-projection algorithm. The performance of the CPU version depends less on the texture size than the GPU versions, but is, in fact, always poor, compared to the W-projection algorithm. With interpolation, the application is 5.4–6.5 times slower than our reference W-projection implementation that does not interpolate. This is not surprising, because the CPU has no dedicated interpolation hardware. The grids that are computed by the GPU version differ slightly from those of the CPU version, due to the limited interpolation accuracy of the GPU. The total powers on the grids, however, are equal in both versions.

The size of the texture has impact on the quality of the eventual image (see Section 7). Since we did not yet determine the impact, we do not know how small the texture can be made without losing too much accuracy.

# 7. FUTURE WORK

A future goal is to develop a GPU imager for the LOFAR radio telescope [13], and later, for the SKA. The ideas in this paper will be useful to achieve good performance. However, low-frequency telescopes like LOFAR need an even more advanced imaging algorithm (AW-projection), that corrects for direction-dependent effects as well. Essentially, this means that the convolution functions become time dependent and have to be recomputed for every five minutes of telescope data. Additionally, the convolution functions will be different for different polarizations.

An open issue is how interpolation of convolution weights in a 3D-texture affects astronomical data quality. As long as the texture contains at least as many entries as the number of oversampled convolution weights of the original W-projection algorithm, interpolation will likely give better results. Unfortunately, as we saw in the previous section, the run times increased by a factor of 6 when a texture is used that is much larger than the convolution matrix size. However, we expect that it is possible to use smaller textures. At least, the interpolation hardware of GPUs allows thinking about interpolation; on CPUs, this is prohibitively slow.

# 8. CONCLUSIONS

We presented a new work-distribution strategy for GPUs, that efficiently convolves radio-telescope data on a grid, a computationally expensive step in the pipeline that creates sky images from radio-telescope data. This strategy significantly reduces the number of device-memory accesses, by accumulating data as long as possible in registers. Unlike previous work-distribution strategies, this strategy neither relies on sorting input data, nor does it search for input data that contributes to some grid point, thus keeping the overhead low.

Performance measurements on various platforms show that the strategy is an order of magnitude faster than other proposed solutions for GPUs, corrected for differences in hardware speed. This order of magnitude improvement is what is necessary to be able to build the Square Kilometre Array. We compared CUDA and OpenCL implementations, and found that the OpenCL implementation suffers from the absence of an atomic floating-point addition to global memory. We also compared different architectures, and show that AMD's new Graphics Core Next architecture is highly competitive with Nvidia's Kepler architecture. Multi-GPU scaling on a system with eight GPUs is good for small-sized convolution matrices and excellent for large ones. The strategy is "green", we achieve almost 2 GFLOP/W (on previous-generation hardware).

Additionally, we showed how the hardware support for interpolations in three-dimensional textures can potentially improve the quality of the generated sky images. However, we also showed that the computational costs can be high, depending on size of the convolution matrix and the size of the texture that stores the convolution functions. Future research must make clear if interpolation or the traditional W-projection algorithm provides a better balance between computational costs and image quality.

## Acknowledgments

We thank Chris Broekema, Ger van Diepen, Rob van Nieuwpoort, and the anonymous reviewers for their comments on a draft of this paper, and Tim Cornwell for a reference implementation of the W-projection algorithm. This work is supported by the SKA-NN grant from the EFRO/Koers Noord programme from Samenwerkingsverband Noord-Nederland, the Astron IBM Dome project, funded by the province Drenthe and the Dutch Ministry of EL&I, and the DAS-4 grant from the Netherlands Organization for Scientific Research (NWO). Intel kindly provided us with a dual Xeon E5 server.

# 9. REFERENCES

[1] U. Bordoloi. Image Convolution Using OpenCL — A Step-by-Step Tutorial, October 2009.

[2] T. Cornwell, K. Golap, and S. Bhatnagar. W-Projection: A New Algorithm for Wide Field Imaging with Radio Synthesis Arrays. In P. L. Shopbell, M. Britton, and R. Ebert, editors, *Astronomical Data Analysis Software and Systems (ADASS XIV)*, number 347 in ASP Conference Series, pages 86–95, Pasadena, CA, October 2004.

[3] P. E. Dewdney and et. al. SKA Phase 1: Preliminary System Description. *SKA Memo*, 130, November 2010.

[4] P. E. Dewdney, P. J. Hall, R. T. Schilizzi, and T. J. L. W. Lazio. The Square Kilometre Array. *Proceedings of the IEEE*, 97(8):1482–1496, August 2009.

[5] R. G. Edgar, M. A. Clark, K. Dale, D. A. Mitchell, S. M. Ord, R. B. Wayth, H. Pfister, and L. J. Greenhill. Enabling a High Throughput Real Time Data Pipeline for a Large Radio Telescope with GPUs. *Computer Physics Communications*, 181(10):1707–1714, 2010.

[6] B. Humphreys and T. Cornwell. Analysis of Convolutional Resampling Algorithm Performance. *SKA Memo*, 132, January 2011.

[7] V. Podlozhnyuk. *Image Convolution with CUDA*, June 2007.

[8] F. Schwab. Optimal Gridding of Visibility Data in Radio Interferometry. In *Proceedings of an International Symposium held in Sydney, Australia*, page 333. Cambridge University Press, August 1983.

[9] A. van Amesfoort, A. L. Varbanescu, H. J. Sips, and R. V. van Nieuwpoort. Evaluating Multi-Core Platforms for Data-Intensive Kernels. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 207–216, Ischia, Italy, May 2009. ACM Press.

[10] R. V. van Nieuwpoort and J. W. Romein. Using Many-Core Hardware to Correlate Radio Astronomy Signals. In *ACM International Conference on Supercomputing (ICS'09)*, pages 440–449, New York, NY, June 2009.

[11] A. L. Varbanescu. *On the Effective Parallel Programming of Multi-Core Processors*. PhD thesis, TU Delft, the Netherlands, December 2010.

[12] A. L. Varbanescu, A. S. van Amesfoort, T. Cornwell, A. Mattingly, B. G. Elmegreen, R. V. van Nieuwpoort, G. van Diepen, and H. J. Sips. Radioastronomy Image Synthesis on the Cell/B.E. In *EuroPar'08*, volume 5168 of *LNCS*, pages 749–762, Las Palmas de Gran Canaria, Spain, August 2008. Springer-Verlag.

[13] M. Vos, A. Gunst, and R. Nijboer. The LOFAR Telescope: System Architecture and Signal Processing. *Proceedings of the IEEE*, 97(8):1431–1437, August 2009.