Image-Domain Gridding on Graphics Processors

Bram Veenboer, Matthias Petschow and John W. Romein
ASTRON (Netherlands Institute for Radio Astronomy)
PO Box 2, 7990 AA Dwingeloo, The Netherlands
Email: {veenboer,petschow,romein}@astron.nl

***Abstract*—Realizing the next generation of radio telescopes such as the Square Kilometre Array (SKA) requires both more efficient hardware and algorithms than today's technology provides. The recently introduced image-domain gridding (IDG) algorithm is a novel approach towards solving the most compute-intensive parts of creating sky images: gridding and degridding. It avoids the performance bottlenecks of traditional AW-projection gridding by applying instrumental and environmental corrections in the image domain instead of in the Fourier domain. In this paper, we present the first implementations of this new algorithm for CPUs and Graphics Processing Units (GPUs). A thorough performance analysis, in which we apply a modified roofline analysis, shows that our parallelization approaches and optimization leads to nearly optimal performance on all architectures. The analysis also indicates that, by leveraging dedicated hardware to evaluate trigonometric functions, GPUs are both much faster and more energy-efficient than regular CPUs. This makes IDG on GPUs a candidate for meeting the computational and energy-efficiency constraints of future telescopes.**

I. Introduction

At the present time, the worlds largest radio telescope, the *Square Kilometre Array* (SKA) [1], is being developed. It will consist~~s~~ of thousands of dishes and hundreds of thousands of antennas; its goal is to "enable astronomers to monitor the sky in unprecedented detail and survey the entire sky much faster than any system currently in existence" [1]. However, even for existing telescopes such as the LOw-Frequency ARray (LOFAR) [2], with in the order of 50,000 antennas grouped into about 50 stations, data processing remains challenging – in particular, when so called direction-dependent effects (DDEs) have to be taken into account [3].

Creating sky images is especially computational intensive: both efficient algorithms and processing are needed to meet the ambitious time and power constraints of instruments such as the SKA [4]. According to a requirements analysis of a pipeline that creates sky images, the subparts of *gridding* and *degridding* are the most dominant parts of the computation [5] Recently, a novel algorithm for these operations was introduced: *Image-Domain Gridding* algorithm (IDG) [6]. Its aim is to address the shortcomings of more traditional methods currently in use – in particular, the challenging DDEs.

The main contributions of this paper are the following: (1) We present the *first* implementations of the IDG algorithm on CPUs and GPUs; (2) We present the *first* efficient *degridding* implementation on GPUs ever; (3) We apply a thorough analysis of the achieved performance, in which we apply a roofline analysis [7] at another abstraction level, where not only the hardware limitations, but also the limitations of the supporting mathematical software is taken into account; (4) We provide an assessment of the increasingly important *energy consumption* on representative CPU and GPU platforms.

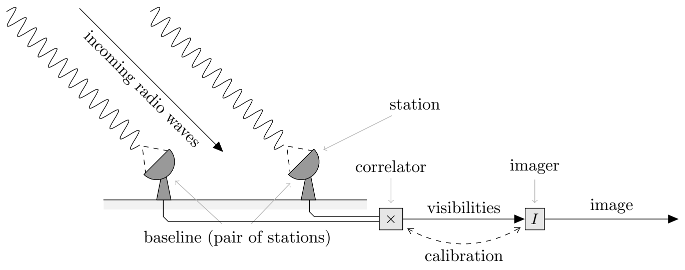

Fig. 1: Incoming radio waves are received by a pair of stations. Since the stations are spaced apart, the signal is out of phase. The visibilities, which are correlations of station signals, contain information on amplitude and phase of the source.

The rest of the paper is organized as follows: In Section II, we provide the necessary background describing the problem that needs to be solved. After discussing related work in Section III, we describe the IDG algorithm in Section IV. In Section V, we show how the algorithm is most efficiently mapped onto state-of-the-art CPUs and GPUs (Intel Xeon, AMD Fury X, and Nvidia GTX 1080). In Section VI, we present experimental results for all architectures. A thorough analysis uncovers architectural and other features that are most relevant to performance and energy efficiency.

II. Background

A radio telescope detects electromagnetic waves that originate from radio sources in the universe. The signals are used, among other things, to construct a map of the sky containing positions, strengths, and polarization of the sources. As opposed to optical telescopes, which usually consists of a single receiver, as mentioned above, modern radio telescopes such as the LOFAR and the SKA are comprised of many small antennas.[1]

As shown in Fig. 1, the creation of a sky image requires roughly three steps: (1) the digitized signals from pairs of distinct stations are correlated to produce measurement data (so called *visibilities*), (2) calibration is used to estimate and correct instrument parameters and environmental effects, and (3) an *imaging* step converts partially corrected visibilities into a sky image.

More precisely, for every frequency channel ν, every station of dipole antennas produces two signals (one for each polarization), and multiplying and integrating (correlating) the

[1] A notable exception is the single dish Five-hundred-meter Aperture Spherical radio Telescope (FAST) [8].

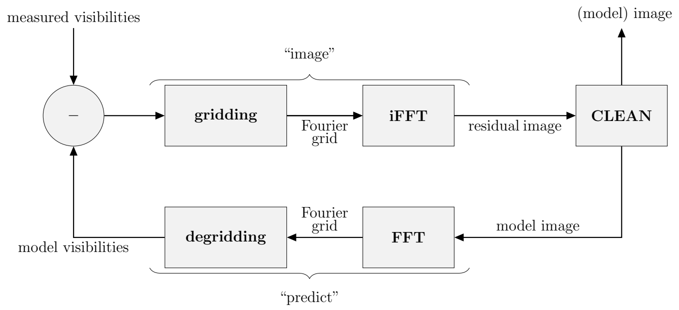

Fig. 2: The imaging step for a single subband.

signals of station pair (p, q) for a short time (in the order of seconds) produces a single visibility, $V_{pq}^{(\nu)} \in \mathbb{C}^{2x2}$. Every visibility is associated with a uvw-coordinate, $(u_{pq}^{(\nu)}, v_{pq}^{(\nu)}, w_{pq}^{(\nu)}) \in \mathbb{R}^3$, which describes the distance of the two stations in units of wavelength – thus, depends on frequency ν. Ignoring the correction for direction-*in*dependent effects, the relation between visibilities and sky brightness, $B(l, m) \in \mathbb{R}^{2x2}$, is given by the following measurement equation [9]:

$$V_{pq}^{(\nu)} = \iint_{\ell\, m} A_p B A_q^H e^{-2\pi i\left(u_{pq}^{(\nu)}\ell + v_{pq}^{(\nu)}m + w_{pq}^{(\nu)}n\right)}\, d\ell\, dm, \quad (1)$$

where $\ell, m \in \mathbb{R}$ are direction cosines of sky coordinates, $n = 1 - \sqrt{1 - \ell^2 - m^2}$, and $A_p^{(\nu)}(\ell, m), A_q^{(\nu)}(\ell, m) \in \mathbb{C}^{2\times 2}$ describe the aforementioned DDEs. In other words, for a given pair of stations and time, a visibility is the weighted sum of the sky brightness in all directions.

Without the W-terms, $e^{-2\pi i w_{pq}^{(\nu)} n}$, and the A-terms, and after uniformly discretizing the sky coordinates, the relation between visibilities and sky image becomes that of a discrete Fourier transform for non-equidistant data (NDFT). In such a scenario, the sky image can be obtained by a non-uniform Fast Fourier Transformation (NFFT), which comprises three steps: (1) the non-uniform visibilities are "gridded" onto a uniform grid by applying an operation that corresponds to a convolution, (2) an inverse FFT is applied to the grid, and (3) a correction of the first step is performed [10]. For Eq. (1) however, the gridding step has also to take the W-terms and A-terms into account. Therefore, we define *gridding* as the means to get, up to a simple correction, the Fourier transform of the discretized sky image from the measured visibilities. Similarly, we define *degridding* as the means to get visibilities from the Fourier transform of the discretized sky image.

In the imaging step (Fig. 2), the measured visibilities are processed independently for different spectral frequency ranges (so called *subbands*). It starts with an empty sky model. After "imaging" (gridding and inverse FFT) the visibilities, one or more bright sources, which mask the more interesting weak sources, are extracted using a variant of the CLEAN algorithm and added to the *sky model* (see [11] and references therein). For these sources, the visibilities are "predicted" (FFT and degridding) and subtracted from the input to reveal fainter sources. This process is repeated until the sky model converges.

III. Related work

The traditional approach to gridding is known as *W-projection* [12] and it neglects the correction of DDEs (the A-terms): the W-terms are treated as a convolution in Fourier space. However, when antennas are spaced far apart from each other to resolve the high spatial frequencies and observing large fields, the support of the W-terms can become large and they have to be recomputed frequently, making this technique inefficient and memory intensive [13]. The combination of W-projection gridding and *W-stacking* improve on this technique by limiting the support size of W-terms at the cost of using more memory [14], [15].

The computational challenge increases even further when the correction for DDEs is taken into account [13]. The correction can be done in a similar manner than the W-term correction – called *A-projection*. Applying both corrections results in the so called *AW-projection* gridding [16].

Most state-of-the-art imagers make use of one or more of the various gridding algorithms and their implementations: e.g., CASA [17] and LOFAR's AWImager [18] uses W-projection and AW-projection, while WSClean [15] uses W-projection in conjunction with W-stacking. A-term correction is included in WSClean, but performance is much lower when this feature is enabled. However, the integration of IDG into WSClean is planned to improve performance for cases where DDEs are of concern.

The first efficient implementation of W-projection gridding on GPUs has been reported in [19], and has since then further been improved [20], [21]. To the best of our knowledge, no gridding and degridding implementation correcting for both W-terms and A-terms has been presented that runs highly efficiently on CPUs and/or GPUs. Even more, no degridding routine has been presented for GPUs at all. As we show in the following, the IDG algorithm, while running efficiently correcting only for the W-term, allows corrections for DDEs at negligible additional cost. It therefore alleviates most of the limitations of traditional AW-projection gridding.

IV. The IDG Algorithm

For every subband (and, if W-stacking is used, every W-plane), we have a situation as depicted in Fig. 3. Due to the earth rotation, the measurements associated with one baseline (i.e., one pair of stations) draw tracks in the form of an ellipsoid in the (u, v)-plane, which is the discrete Fourier Transform (DFT) of the sky image (simply called *the grid* hereafter). As the subband frequency range is further discretized into C *channels*, each baseline is associated with C tracks and T time steps.

Traditional W-projection and AW-projection apply a convolution kernel to each of the visibilities, as illustrated in Fig. 3. The fraction of non-zero pixels of the grid is called the uv*-coverage*. In order to increase the uv-coverage for high-resolution images, *longer* baselines (that is, with larger u and v values) must be used. Visibilities do not exactly map to grid coordinates, which is corrected by oversampling the convolution function. The convolution kernels in W-projection or AW-projection gridding form a potentially large multi-dimensional data structure that scales quadratically in size with

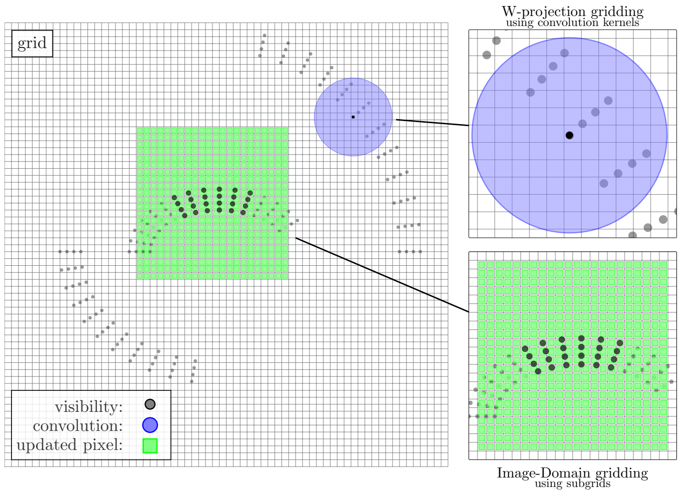

Fig. 3: In traditional W-projection and AW-projection gridding, visibilities are gridded using convolutions (top-right) as opposed to correcting the W-term and A-term effects in the image domain (bottom right). For the latter, neighboring visibilities are gridded on small 'subgrids'.

```
1  $S = 0$;                                   // $\mathbb{C}^{\tilde{N}\times\tilde{N}\times 2\times 2}$ subgrid
2  for $y$=1,...,$\tilde{N}$ do
3      for $x$=1,...,$\tilde{N}$ do
4          for $t$=1,...,$\tilde{T}$ do          // time
5              for $c$=1,...,$\tilde{C}$ do      // channel
6                  $\alpha = f(x,y) \cdot g(u(t,c), v(t,c), w(t,c))$;
7                  $\Phi = \cos(\alpha) + i\sin(\alpha)$;
8                  for $p$=1,2 do        // polarization
9                      for $q$=1,2 do
10                         $S(y,x,p,q) \mathrel{+}= \Phi \cdot \tilde{V}(t,c,p,q)$;
11                     end
12                 end
13             end
14         end
15     end
16 end
17 apply_aterm($S$);
18 apply_spheroidal($S$);
19 store $S$;
```

Algorithm 1: Pseudocode that is executed for every subgrid in the gridder kernel. For every evaluation of $\sin(\alpha)$ and $\cos(\alpha)$, 17 real-valued multiply-add operations are performed (one in the evaluation of $f()$, 16 in the addition to S).

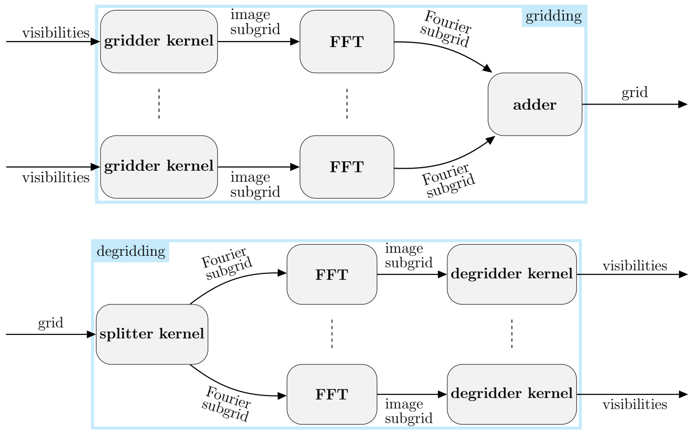

Fig. 4: The Image-Domain routines are drop-in replacements for the gridding and degridding step in Fig. 2.

both the number of pixels in one dimension of the image and an oversampling factor. IDG makes use of the classical convolution theorem [22] to perform both W-correction and A-correction in the *image domain*. Consequently, large convolution functions are not needed.

At the center of the algorithm are so called *subgrids*, which represent low-resolution versions of the sky brightness for a small subset of visibilities (Fig. 3). These subgrids are placed onto the grid in such a way that they cover a subset of $\tilde{T} \times \tilde{C}$ visibilities and their associated W-term and A-term convolution kernels (see also Section V). Every pixel of the subgrid is then computed as a direct sum of shifted visibilities, as shown in Algorithm 1. Afterwards, both W-term and A-term corrections are applied. Since we have performed the corrections in the image domain, the subgrid has to be Fourier-transformed before the result is added to the grid (i.e., four $\tilde{N} \times \tilde{N}$ FFTs per subgrid, one for every of the four combinations of p and q).

The entire process is illustrated in Fig. 4. Image-Domain Gridding consists of three steps: (1) The visibilities are gridded onto subgrids by the *gridder kernel*, which applies Algorithm 1 for every subgrid; (2) the subgrids are Fourier-transformed. This step will be referred as the *subgrid FFTs*; (3) the transformed subgrids are added to the grid by the *adder*.

The *degridding* step is similar to the gridding step, but proceeds in reverse order: First, subgrids are extracted from grid by the *splitter*, then every subgrid is Fourier transformed by an inverse FFT, and finally the associated visibilities are predicted by the *degridder kernel*, which is similar to the gridder kernel and shown in Algorithm 2.

The minimum size of the subgrids follow from the size of the W-term and A-term convolutions kernels and the tapering function that used to reduce aliasing (such as a spheroidal, which is used in our case). For the LOFAR telescope, subgrids as small as 24×24 pixels are found to provide sufficient accuracy to exceed the accuracy of traditional gridding [6]. Furthermore, larger subgrids (e.g. up to 64×64) can be used in connection with W-stacking to dramatically limit the number of required W-planes [6]. Due to the smaller memory footprint, IDG not only makes the computation efficient, but also enables the creation of larger, higher-resolution images [6].

V. Architectures & Optimization

In this section we demonstrate how the IDG algorithm is efficiently mapped onto modern CPUs and GPUs. The CPU class will be represented by Intel's Xeon processors family, while the GPU class is represented by an AMD Fury X (Fiji architecture) and Nvidia GTX 1080 (Pascal architecture).

A. The execution plan

Before any execution, we need to specify the subgrid locations and associate visibilities to them. This is done in form of an *execution plan*.

```
 1 $\tilde{V} = 0$;                    // $\mathbb{C}^{\tilde{T}\times\tilde{C}\times 2\times 2}$ visibilities
 2 apply_spheroidal($S$);
 3 apply_aterm($S$);
 4 for $t$=1,...,$\tilde{T}$ do                 // time
 5     for $c$=1,...,$\tilde{C}$ do             // channel
 6         for $y$=1,...,$\tilde{N}$ do
 7             for $x$=1,...,$\tilde{N}$ do
 8                 $\alpha = -f(x,y)\cdot g(u(t,c),v(t,c),w(t,c))$;
 9                 $\Phi = \cos(\alpha) + i\sin(\alpha)$;
10                 for $p$=1,2 do     // polarization
11                     for $q$=1,2 do
12                         $\tilde{V}(t,c,p,q)$ += $\Phi\cdot S(y,x,p,q)$
13                     end
14                 end
15             end
16         end
17     end
18 end
19 store $\tilde{V}$;
```
Algorithm 2: Pseudocode that is executed for every subgrid or work item in the degridder kernel.

If $\mathcal{V} = \{V_{pq}(t,c) : \forall t\ \forall c\}$ denotes the visibilities from all baselines, the positions of the subgrids induce a partitioning $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \ldots \cup \mathcal{V}_n$. This process of positioning the subgrids to cover all visibilities is implemented in form of a greedy algorithm. As depicted in Fig. 5, not only the visibilities need to be covered by the subgrids, but also the support of their associated AW-projection convolution kernels [6]. Thus, for each baseline, starting with time step 1 and having $\tilde{C}$ channels that can be covered by an $\tilde{N}\times\tilde{N}$ subgrid, we include as many time steps as possible (each with $\tilde{C}$ channels) until they cannot be covered anymore. In that case, we create a new subgrid (with a new positions on the grid) to cover the remaining channels and repeat the process at time step $\tilde{T}+1$.

We might additionally require that $\tilde{T} \leq \tilde{T}_{\max}$ (where $\tilde{T}_{max}$ is architecture-specific and dependent on the other observation parameters) to limit the maximal number of time steps that are associated with a single subgrid. Such an approach keeps the amount of computation to be performed for each subgrid comparable, and the memory required for that computation limited.

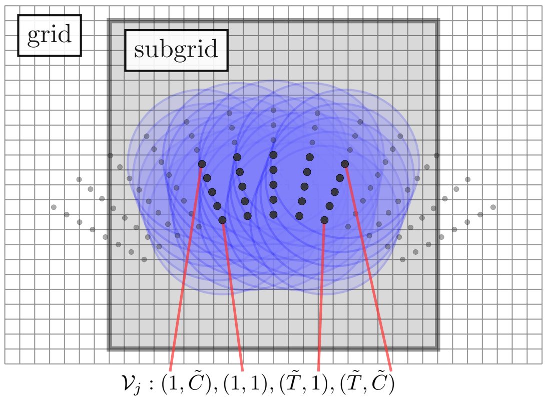
grid
subgrid
$\mathcal{V}_j : (1,\tilde{C}), (1,1), (\tilde{T},1), (\tilde{T},\tilde{C})$

Fig. 5: A subset of visibilities ($\mathcal{V}_j$, black dots), including their associated AW-projection convolution kernels (blue circles), is covered by a subgrid.

We call each subgrid S_j (including its metadata such as its position in the grid) together with its associated visibilities $\mathcal{V}_j$ (including uvw-coordinates) a *work item*. The set of all n *work items* is called the *work*, and is generated by the execution plan. After grouping $m \leq n$ work items into a *work group*, the gridder and degridder kernels process a work group by using Algorithm 1 and Algorithm 2 for every work item, respectively. This work division hierarchy is illustrated in Fig. 6, and we will refer to it later to show how the algorithm is mapped differently onto the two distinct architectures classes, CPU and GPU.

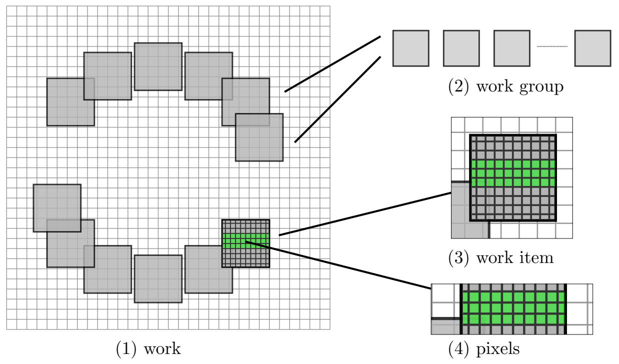

Fig. 6: Work division within IDG: (1) The *work* is split into (2) work groups, represented by a subset of all subgrids. (3) A work group consists of *work items*, represented by an individual subgrid. (4) For gridding, the smallest unit of work is the computation of a single pixel. For degridding, the smallest unit of work is the computation of a single visibility.

B. *CPU*

The CPU class is represented by Intel's Xeon server processor family. These CPUs are available in a wide variety of models and vary in core-counts (up to 24 cores) and clock speed (up to 3.4 Ghz). The last generations (Haswell-EP and Broadwell-EP) have a similar underlying architecture: they implement the AVX2 instruction set and have, for each core, 64 KB of L1 Cache and 256 KB L2 Cache. Furthermore, all cores share a L3 cache of 2.5 MB per core. The clock speed is dynamically adjusted depending on the workload.

Every core contains two floating point execution units (FPUs), which both support vector fused multiply-add (FMA) instructions, with 8-element single precision SIMD vector width. To fully utilize all CPU cores and execution units, IDG needs to exploit both thread-level and data-level parallelism. Finding the right granularity for both levels is key to good performance and scalability. We now detail our strategy for various subparts: *a) gridder kernel, b) degridder kernel, c) subgrid FFTs,* and *d) adder and splitter*.

a) Gridder kernel: The gridder kernel (as all other kernels implemented in C++) is executed for every *work group*. It distributes the *work items* over all logical cores using OpenMP. That is, each thread computes a subset of the subgrids according to Algorithm 1.

The most important performance optimizations are the following: (1) We load and transpose visibility data of $T_B \leq \tilde{T}$

Listing 1: The *reduction* clause instructs the compiler to vectorize over channels. Each vector loop executes 16 FMAs.

```
#pragma omp simd reduction(+:...)
for (int c = 0; c < NR_CHANNELS; ++c) {
    pix_pp_re += vis_pp_re[c] * Phi_re[c];
    pix_pp_im += vis_pp_re[c] * Phi_im[c];
    pix_pp_re -= vis_pp_im[c] * Phi_im[c];
    pix_pp_im += vis_pp_im[c] * Phi_re[c];

    // [... same for 'pix_pq' and 'pix_qp']

    pix_qq_re += vis_qq_re[c] * Phi_re[c];
    pix_qq_im += vis_qq_re[c] * Phi_im[c];
    pix_qq_re -= vis_qq_im[c] * Phi_im[c];
    pix_qq_im += vis_qq_im[c] * Phi_re[c];
}
```

time steps and $C_B \leq \tilde{C}$ time steps at a time into memory-aligned arrays to allow for non-strided data access. (T_B and C_B are a platform-specific optimization parameters; that is, the computation is performed in batches.) At this moment, we also separate real and imaginary part of the operands. (2) The sine/cosine-computations (Line 7 of Algorithm 1) are precomputed for the entire batch of visibilities with either Intel's Short Vector Math Library (SVML) or Vector Math Library (VML), whichever is faster. (3) We vectorize the channel loop (Line 5) by writing the computation in the form of a SIMD reduction illustrated in Listing 1.

The vectorization works best when the number of channels is a multiple of the SIMD vector width, as otherwise the *remainder*(C_B, SIMD_WIDTH) channels will be processed using masked vector instructions. This implies that wider vectors will not necessarily result in higher performance. Finally, we aid compiler assisted vectorization in the remainder of the kernel by using *runtime compilation*, i.e. we only compile the kernel when the parameters are known at runtime.

b) Degridder kernel: We distribute the work in the same manner as in the gridder kernel: each thread processes a subset of a work group by applying Algorithm 2 on every work item. In other words, each thread computes the visibilities for a subset of the subgrids. The kernel optimizations are similar to the optimizations for the gridder kernel. A notable difference is that we apply vectorization over pixels (Line 7 in Algorithm 1). This works best when the number of pixels in one row of the subgrid is a multiple of the SIMD vector width.

c) Subgrid FFTs: All subgrids are Fourier-transformed before adding them to the grid and after splitting the subgrids from the grid, respectively. This is an embarrassingly parallel process and most efficiently done by using a highly-optimized math library such as the Intel's Math Kernel Library (MKL).

d) Adder and splitter: As subgrids might partially overlap in the grid, for the adder, parallelization over subgrids would imply prohibitive synchronization costs. Instead, we parallelize over the rows of the grid to avoid concurrent access to the same pixels. For the splitter, overlapping subgrids are not a problem as the data from the grid is read-only. Therefore, we parallelize over subgrids.

C. GPU

AMD GPUs are programmed in OpenCL, the open GPGPU programming standard by Khronos [23], and NVIDIA GPUs are programmed in either OpenCL or CUDA, the proprietary standard by Nvidia [24]. Both of these standards include a runtime system and a set of C/C++ extensions to operate the GPU. Apart from syntactic differences, these standards offer the same basic functionality. In the remainder we adhere to the CUDA terminology.

Compared to CPUs, high-end GPUs such as Nvidia's GTX 1080 have a much higher core count (2560 cores) and run at a lower clock speed (1733 Mhz). The cores are organized in smaller groups of 128 cores, which are called *Streaming Multiprocessors* (SMs). Every SM contains a register file, load-store units, and dedicated caches. GPUs lack a L3 cache as found on most CPUs, instead each SM has a software-managed cache. GPUs are connected to a *host* machine using PCI-E and have their own device memory. The device memory is limited in size (8 GB), but faster than on a regular Xeon-based system (e.g. 320 GB/s versus 68 GB/s). Since it resides in a separate memory space, this means that data needs to be copied explicitly over the PCI-E bus to and from device memory.

On any GPU, threads are organized in a three-level hierarchy: grid-level, block-level, and warp-level. When a kernel is executed, the GPU spawns a *grid* of *thread-blocks* (both of user-specified dimensions) and dispatches the thread-blocks onto the SMs.

It is possible to impose light-weight barriers within a thread-block to synchronize threads; however, synchronization between thread-blocks requires costly atomic operations on device memory. When branches are encountered during kernel execution the implication are similar to masked vector-instructions on CPUs and result in sub-optimal performance.

This GPU programming model requires a different parallelization strategy compared to the CPU implementation. We detail our parallelization and optimization strategies in the following paragraphs.

a) Asynchronous I/O and kernel execution: We use triple-buffering to prevent the GPU from being idle during data transfers. To this end we start three threads on the host, allocate buffers on the device (threefold) and create three different CUDA *streams*: one for host-to-device memory transfers, one for kernel execution, and one for device-to-host memory transfers. Each host thread processes a subset of the work groups of the work by issuing operations onto the streams. CUDA *events* are used to synchronize between streams and ensure sequential consistency between the operations within the kernel invocation; i.e., the kernel only starts when the input data is transferred.

Fig. 7 illustrates how this technique overlaps I/O and kernel execution. At the beginning of this example execution, all threads enqueue three operations on the respective streams: (1) copy of input data from host to device (HtoD), (2) kernel execution and (3) copy of results from device to host (DtoH). Furthermore, thread T_1 only starts the copy of input data for the next kernel invocation (indicated with dashes) when its input buffer can be overwritten.

model	type	architecture	clock (GHz)	core config[a] = #FPUs	peak (TFlops)	mem size (GB)	mem bw (GB/s)	TDP (W)
Intel Xeon E5-2697v3	CPU	Haswell-EP	2.60[b]	$2\times14\times2\times08 = 448$	2.78[b]	$\leq$1536	136	290
AMD R9 Fury X	GPU	Fiji	1.050	$1\times64\times1\times64 = 4096$	8.60	4	512	275
NVIDIA GTX 1080	GPU	Pascal	1.80[b]	$1\times40\times2\times32 = 2560$	9.22[b]	8	320	180

[a] #ICs $\times$ #compute units $\times$ FPU instructions/cycle $\times$ vector size
[b] turbo mode enabled

TABLE I: The three architectures used in this comparison.

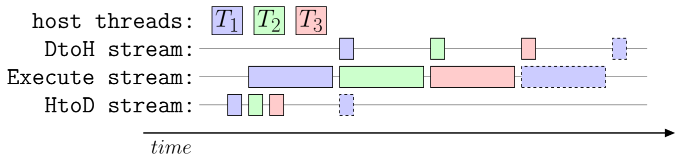

Fig. 7: Three host threads and three CUDA streams are used to implement triple buffering. This way, memory copies from and to the GPU are overlapped with kernel execution as much as possible.

b) Gridder kernel: The gridder kernel is launched once for every work group. The number of thread blocks is set according to the number of work items in the work group. The number of threads in a thread block is an architecture specific optimization parameter: We found that 192 and 256 threads works best for the Nvidia GTX 1080 and the AMD Fury X, respectively.

In order to process a single work item, a thread block executes Algorithm 1. We apply the following optimizations: (1) Line 2 and 3 have been collapsed and threads are mapped onto pixels based on their position in the thread block. (2) Since the size of input data scales linearly with $\tilde{N}$ and $\tilde{C}$, we process the input data in batches. Thus, we transpose and prefetch the batch of data into fixed size shared memory buffers. (3) While iterating through the batch data, every thread accumulates all contributions to its current pixel in registers. Only when the threads have finished the entire batch, they write the pixel value to device memory using coalesced memory accesses. (4) The sine/cosine computation in Line 7 is performed using regular `sin` and `cos` function calls. During compilation we pass `-use_fast_math` to the CUDA compiler to enable the use of optimized sine/cosine functions. These functions have a maximum error of 2 units in the last place (ulps) [25], which is sufficient for IDG [6].

c) Degridder kernel: As for the gridder kernel, the degridder kernel is launched once for every work group. For the processing of a single work item, a more elaborate parallelization strategy is used, in which threads have two different roles. We apply the following optimizations to Algorithm 2: (1) We collapse Line 4 and 5 and map threads onto (a subset of) visibilities based on their position in the thread block – the first mapping. (2) We collapse Line 6 and 7 and also map the threads onto pixels – the second mapping. According to the second mapping, every thread loads a pixel value from device memory, applies the spheroidal and the A-term correction (Line 2 and 3), and stores the result into shared memory. Furthermore, every thread evaluates $f(x, y)$ in Line 8 and stores the result in shared memory. At this point threads return to their first role (according to the first mapping) and perform the remainder of the algorithm: they evaluate $g()$ in Line 8 and perform Line 9 to 12. Next, the procedure is repeated for the next batch of pixels. (3) During the entire process, threads keep their first role and update their associated visibility in registers. (4) Like in the gridder kernel, the sine/cosine-evaluations (Line 9) are executed using optimized math functions.

The optimal number of threads for this kernel are 128 and 256 threads for the Nvidia GTX 1080 and AMD Fury X, respectively. When the number of threads is lower than the $\tilde{T} \times \tilde{C}$ visibilities associated with the subgrid, the procedure described above is repeated: threads are mapped onto a next batch of visibilities and an iteration over all pixels is performed until the entire work item is processed.

d) Subgrid FFTs: We use the cuFFT and clFFT libraries to compute Fourier transformations for the CUDA and OpenCL implementations, respectively.

e) Adder and splitter: We have two options to add all subgrids onto the grid: (1) Perform the operation on the GPU, or (2) copy the subgrids into host memory and perform the operation on the CPU. The second option is only required when dealing with large images that no longer fit into GPU device memory. For the first option, a relatively simple kernel, where every thread-block atomically adds all pixels to the grid was found most efficient. In CUDA we use the built-in `atomicAdd` function twice, to add both the real and imaginary component of the pixel to the grid. In OpenCL, support for atomic operations on floating point numbers is not available as of version 2.1, so we resorted to a loop that updates grid values using `atomic_cmpxchg`.

Unlike the adder kernel, no synchronization is required for the splitter kernel. For every subgrid we copy the relevant part of the grid into the subgrid buffer.

VI. Results

In this section, after describing the experimental setup, we analyze IDG's performance for each architecture in detail. Additionally, we demonstrate that IDG is far more suited for GPUs: it is almost an order of magnitude faster and more energy efficient than on CPUs.

A. Experimental setup

All experiments were executed on the DAS-5 cluster [26] using the hardware listed in Table I. We refer to the three systems as Haswell (a dual-socket system with Haswell-EP

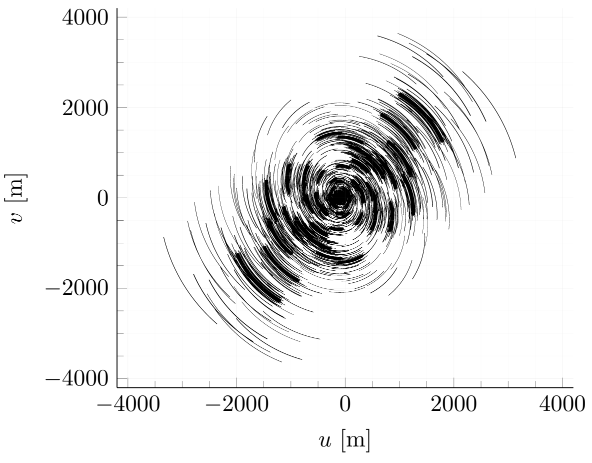

Fig. 8: (u, v)-plane for our test data set.

processors), Fiji (a system hosting an AMD R9 Fury X GPU), and Pascal (a system hosting a NVIDIA GTX 1080 GPU).

For Haswell, in our experiments, we used Intel's compiler version 17.0.0 together with MKL version 2017.0.0; for Fiji, we used the AMD APP SDK 3.0 OpenCL runtime and GPU driver version 1800.11; for Pascal we used CUDA 8.0.27 and GPU driver 367.35. All computations are performed in single precision floating point operations (flops). We therefore omit the term "single precision" from here on.

As performance is data dependent (the uvw-coordinates determine the subgrid configuration and, hence, the computational intensity within the gridder and degridder kernels, the synchronization in the adder, and the total amount of subgrids to be Fourier-transformed), we created a representative benchmark using proposed antenna coordinates for the *SKA-1 low* telescope – the phase 1 subset of the SKA covering the low frequency spectrum [27]. The data set has the following parameters: 150 stations (11,175 baselines), T = 8,192 time steps (at 1 second integration time) and C = 16 channels; the A-terms (for simplicity, all set to identity) are updated every 256 time steps; the subgrids and grid are 24 × 24 pixels and 2048 × 2048 pixels in size, respectively. The (u, v)-plane for this data set is illustrated in Fig. 8.[2]

B. Performance comparison

In Fig. 9 and 10 we present respectively the execution time and throughput (measured in MVisibilities/s) for the single imaging cycle shown in Fig. 2. Both GPUs complete the task almost an order of magnitude faster than Haswell. For all architectures, runtime is dominated by the gridder and degridder kernels (more than 93%). Since the impact on execution time of all other kernels is limited, we focus on the gridder and degridder kernels for the remaining analysis.

[2]Although the data set is representative for a wide variety of use cases, a benchmark covering more of the parameter space will be performed, once IDG is integrated into state-of-the-art imagers We intend to make both the input data as well as the software publicly available.

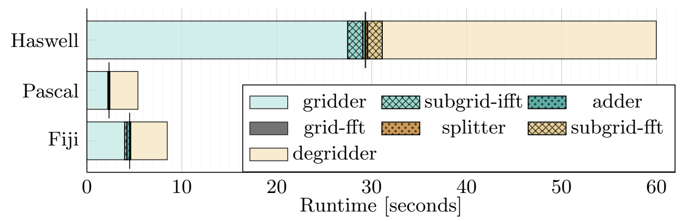

Fig. 9: Distribution of runtime for one full imaging cycle.

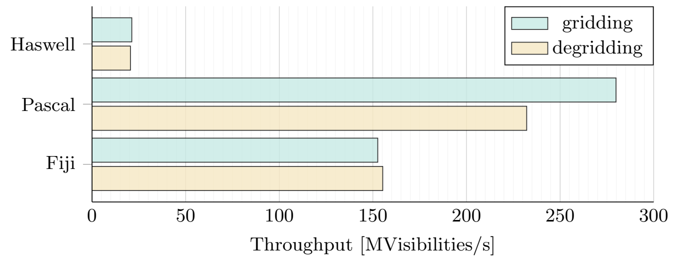

Fig. 10: Throughput for gridding and degridding.

In Fig. 11 we show performance for the two kernels in the form of a roofline plot, where an operation (an *op*) is defined as one of the following: $+, -, *, \sin(), \cos()$. As about 17 out of 18 operations in Algorithms 1 and 2 are FMAs, the plot can be interpreted as TFlops/s, if the sine/cosine-evaluations are ignored. However we include the sine/cosine evaluations in the definition of an operation for two reasons: (1) we do not have any influence on how fast and resource consuming these mathematical functions are evaluated; thus, we have to treat them as black boxes; and (2) on Pascal these operations are supported in hardware [25]. Despite this definition, the peak performance measured in TOps/s is still only achieved when using FMAs exclusively.

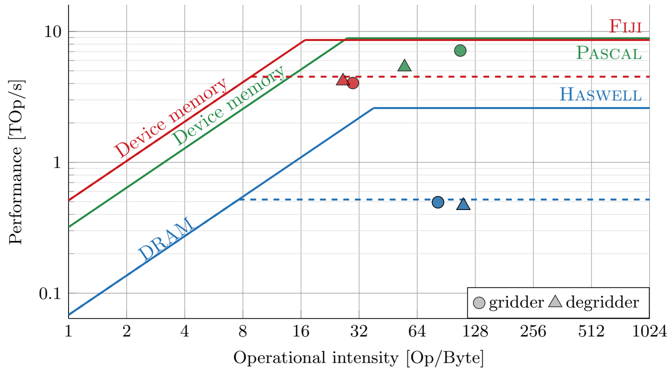

Fig. 11: Roofline analyis: one operation is $+, -, *, \sin(), \cos()$. Peak performance is only attained if non-masked FMA instructions (two operations) are used exclusively.

On all architectures, both kernels are compute bound measured by their operational intensity – the number of operations per byte moved from/to main memory. While the operation count is known exactly, the data movement is measured. Despite being compute bound, only on Pascal we achieve close to the theoretical peak performance. In contrast, on

HASWELL and FIJI, both kernels perform significantly lower than the peak.

C. Performance analysis

On HASWELL and FIJI, although not being compute bound, the performance seems far from optimal and requires further investigation.

1) The HASWELL *and* FIJI *case:* Defining sine/cosine-evaluations as operations comes with a number of complications: (1) their performance *highly* depends on the mathematical library that is used; (2) their performance depends on the settings such as *accuracy* and number of values computed at once; (3) their performance can also be dependent on the values of the input. On HASWELL, to simplify the analysis, in this section, we only use SVML whose performance is independent on the number of values computed. Furthermore, we select medium accuracy (maximum of 4 ulps error), and values in the range of $[-10^4, 10^4]$, to achieve the highest performance. On FIJI, the native AMD math library is used by specifying the `-cl-fast-relaxed-math` flag during compilation.

As the gridder and degridder kernel perform 17 real-valued FMA instructions for every sine/cosine-evaluation (see Algorithm 1 and Algorithm 2), this situation is similar to an instruction imbalance in traditional roofline analysis, where peak performance is only achieved if an equal number of multiplication and additions are performed in form of FMAs. In order to determine an upper bound on performance for our workload, we benchmarked the performance for various ratios of

$$\rho = \frac{\text{number of FMAs}}{\text{number of sincos}}.$$

In this definition we use the fact that, for our kernels, sine and cosine are always evaluated on the same argument, which is more efficient than having distinct arguments. The result of the benchmark is shown in Fig. 12.

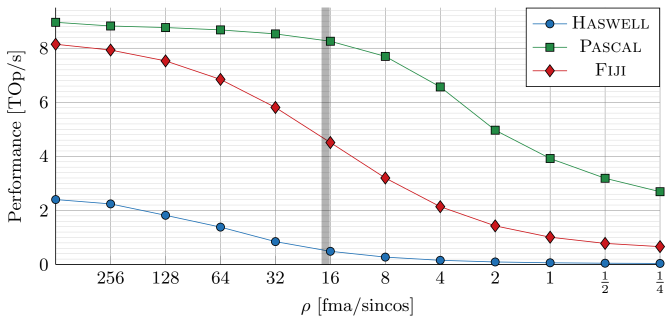

Fig. 12: Operation throughput for various mixes of FMA instructions and sine/cosine-evaluations.

PASCAL's SMs contain a number of special function units (SFUs) that implement the computation of both transcendental functions and interpolation in hardware [28]. Since sine/cosine is handled in a separate processing queue, the performance of PASCAL stays high when ρ decreases. In contrast, on FIJI, the sine/cosine-evaluations are performed by the same ALUs that also compute the FMAs, at a quarter of the rate [29]. Consequently, a more significant performance degradation is observed for small values of ρ. A similar behavior is observed for HASWELL.

Given these insights, we establish new upper bounds for the peak Ops/s for the gridder and degridder kernels in Fig. 11 (dashed lines). These bounds correspond to the values in Fig. 12 for $\rho = 17$. Now, the gridder and degridder kernels are close to optimal, given the limitations of hardware *and* the supporting mathematical library; unfortunately, we cannot use the full computational capacity of HASWELL and FIJI without algorithmic changes.

2) The PASCAL *case:* Although sine/cosine-evaluations are supported in hardware, the measured kernel performance is below the theoretical peak performance (Fig. 11): 74% and 55% of the peak for the gridder and degridder kernel, respectively. To further investigate the cause, we create a second roofline graph in Fig. 13 – this time with respect to shared memory instead of device memory.

Both kernels are close to the shared memory bandwidth bound. Being only limited by shared memory bandwidth, explains the good performance that can be seen in Fig. 11 for PASCAL. Interestingly, the kernels on FIJI are also relatively close to hitting the shared memory bandwidth limit.

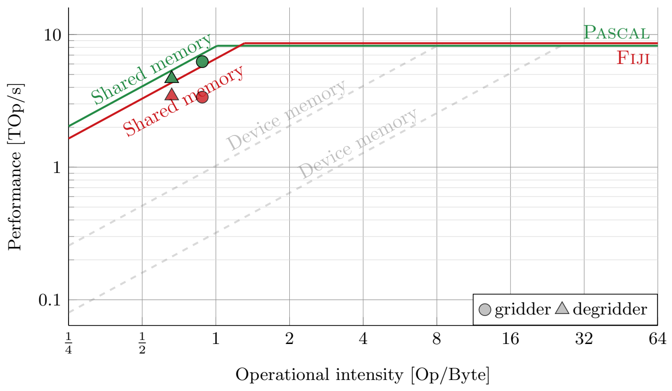

Fig. 13: Roofline graph with operational intensity computed with respect to the amount of bytes transferred from shared memory instead.

D. Energy-efficiency comparison

We use LIKWID [30] to measure the energy usage of both the package (CPU cores and caches) and the DRAM for HASWELL. For FIJI and PASCAL, we measure energy consumption of the full PCI-E device, using PowerSensor [31], which provides power measurements at high time resolution and enables us to analyze energy usage for individual compute kernels. Since both GPUs require a host to operate, we additionally measure energy consumption for the package and the DRAM.

Fig. 14 shows the total energy consumed during the execution of a single imaging cycle. As most time is spent in the gridder and degridder kernels (Fig. 9), most energy is naturally spent in these kernels. Thus, also in terms of total energy consumption, the GPUs outperform the CPU by an order of magnitude. This is even true when the power consumption of the host is taken into account.

In Fig. 15 we present the energy-efficiency of the individual kernels. PASCAL is clearly the most efficient GPU: for the gridder and degridder kernel, it achieves 32 and 23 GFlops/W, respectively. Second, but still with about 13 GFlops/W, comes FIJI. HASWELL lags far behind the GPUs, achieving only about 1.5 GFlops/W.

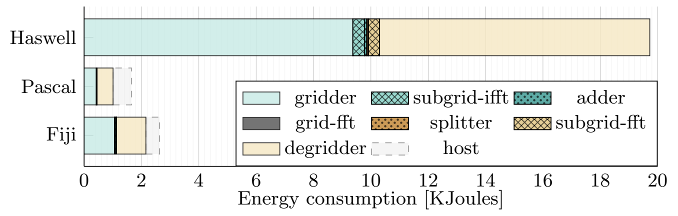

Fig. 14: Distribution of energy consumption for a single imaging cycle.

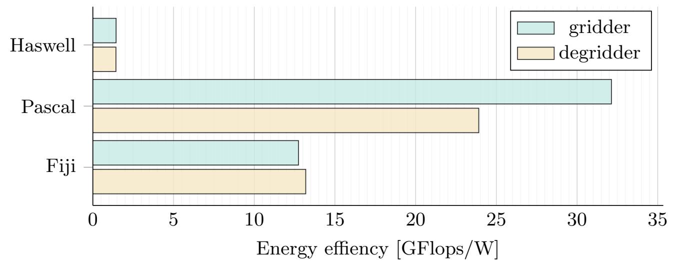

Fig. 15: Energy efficiency for the gridder and degridder kernels.

E. Comparison with W-projection

For the same data set, we compare IDG with the W-projection algorithm introduced in [19]. The latter is one of the fastest implementations of W-projection gridding and we refer to it as WPG in the following. For our tests, WPG uses an oversampling factor of 8 for the W-kernels, and their computation is not included in the measurement of the execution time.

The maximum size of the W-kernels, $N_W \times N_W$ pixels, is determined by the observation settings (the instrument, the field of view, the target location, etc.) [16], [18]. For the LOFAR telescope, although the maximum W-kernel can be as large as 500×500 pixels for pure W-projection gridding, N_W is usually much smaller ($N_W = 30$ on average) [18]. In practice, WPG and IDG are used in conjunction with W-stacking to limit N_W to small values in all situations (e.g. $N_W \leq 16$). Allowing large W-kernels however, reduces the need to have additional W-planes (copies of the grid, where a subset of the visibilities are gridded onto), which can be prohibitively memory consuming for high-resolution images.

With the above in mind, we present in Fig. 16 the performance on PASCAL for various values of N_W, ranging from the relatively large to the relatively small. While for large W-kernel sizes we see comparable performance, for smaller kernels, IDG outperforms WPG significantly. However, very

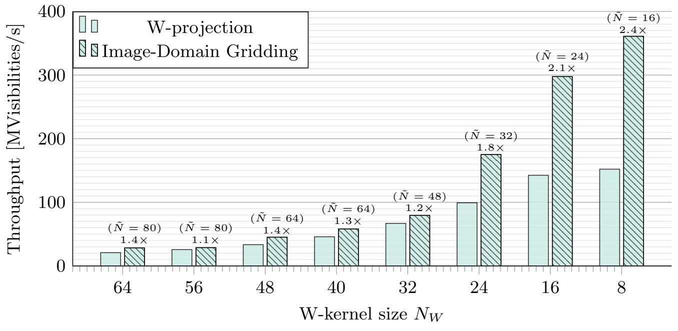

Fig. 16: Throughput of WPG [19] and IDG for various W-kernel sizes. In practice, $N_w \leq 24$ is more common than larger values for N_W. $\tilde{N}$ denotes the subgrid size used within IDG.

recently improvements of WPG have been introduced that can increase its performance from roughly 28% of the peak floating point performance (which me measured in our tests) to 55% in the best case [21]. Even in this scenario, IDG's performance is comparable to W-projection gridding *without the need of the potentially costly computation and storage of the W-kernels.*

More importantly, IDG is particularly suitable if A-term corrections need to be applied; in this case, the additional cost to IDG is negligible, while including this feature to the W-projection gridding (so called AW-projection) "requires significantly more instructions and bandwidth for loading the [convolution kernels], because they are dependent on time, frequency, polarization and possibly baseline, and are not separable" [21]. As a consequence, AW-projection gridding becomes much slower and memory-consuming than W-projection gridding. Although, to draw definite conclusions, a more complete comparison of traditional gridding and IDG is needed, the results suggest that IDG on GPUs is a very significant achievement to address the problems of AW-projection gridding: IDG achieves the performance of the simpler W-projection gridding without the storage overhead, while providing the functionality of the significantly more challenging AW-projection gridding.

VII. CONCLUSIONS

We presented the first implementations of the novel Imaging-Domain Gridding algorithm on CPUs and GPUs; we demonstrated that our parallelization and optimization strategies lead to nearly optimal performance on three distinct architectures: Intel Xeon (Haswell), AMD Fiji, and Nvidia Pascal. In particular for the latter two, the Graphics Processors, the IDG algorithm elegantly maps onto the underlying hardware. While the AMD architecture, similar to the CPU, is limited in performance by evaluations of sine/cosine functions in software, the Nvidia architecture has hardware support for their computation. As a consequence, our code achieves up to 74% of the floating point peak performance and over 30 GFlops/W on Nvidia GPUs. Due to being an order of magnitude faster than on CPUs, the energy efficiency is equally an order of magnitude higher. Therefore, IDG on GPUs is a candidate to meet the demanding computational and energy-efficiency constraints imposed by future telescopes such as the Square Kilometre Array (SKA). In this regard, we hope that our work contributes to making discoveries in radio astronomy possible.

Acknowledgments

This work is supported by the Dutch Ministry of EZ and the province of Drenthe through the ASTRON-IBM Dome grant, the EU FP7 under grant no ICT-610476 (DEEP-ER), and by the NWO through NWO-M (DAS-5 [26]) and Open Competition (Triple-A) grants. The European Commission is not liable for any use that might be made of the information contained in this paper.

References

[1] The SKA Organisation, "Square Kilometre Array," [Online]. Available: www.skatelescope.org.

[2] M. P. van Haarlem *et al.*, "LOFAR: The LOw-Frequency ARray," *Astronomy & Astrophysics*, vol. 556, 2013.

[3] R. J. van Weeren *et al.*, "LOFAR facet calibration," *The Astrophysical Journal Supplement Series*, vol. 223, no. 1, p. 2, Mar. 2016.

[4] E. Vermij, L. Fiorin, R. Jongerius, C. Hagleitner, and K. Bertels, "Challenges in exascale radio astronomy: Can the SKA ride the technology wave?" *International Journal of High Performance Computing Applications*, vol. 29, no. 1, pp. 37–50, Feb. 2015.

[5] R. Jongerius, S. Wijnholds, R. Nijboer, and H. Corporaal, "An end-to-end computing model for the square kilometre array," *IEEE Computer*, vol. 47, no. 9, pp. 48–54, Sep. 2014.

[6] B. v. d. Tol and B. Veenboer, "Image Domain Gridding," unpublished.

[7] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[8] R. Nan *et al.*, "The five-hundred-meter aperture spherical radio telescope (fast) project," *International Journal of Modern Physics D*, vol. 20, no. 06, pp. 989–1024, Jun. 2011.

[9] O. M. Smirnov, "Revisiting the radio interferometer measurement equation," *Astronomy & Astrophysics*, vol. 531, A159, Jul. 2011.

[10] L. Greengard and J.-Y. Lee, "Accelerating the Nonuniform Fast Fourier Transform," *SIAM Review*, vol. 46, no. 3, pp. 443–454, Jan. 2004.

[11] U. Rau, S. Bhatnagar, M. A. Voronkov, and T. J. Cornwell, "Advances in Calibration and Imaging Techniques in Radio Interferometry," *IEEE Proceedings*, vol. 97, pp. 1472–1481, Aug. 2009.

[12] T. J. Cornwell, K. Golap, and S. Bhatnagar, "The Noncoplanar Baselines Effect in Radio Interferometry: The W-Projection Algorithm," *IEEE Journal of Selected Topics in Signal Processing*, vol. 2, no. 5, pp. 647–657, Oct. 2008.

[13] A. Scaife, *SDP Memo: The SDP Imaging Pipeline*, 2016.

[14] T. J. Cornwell, M. A. Voronkov, and B. Humphreys, "Wide field imaging for the square kilometre array," *Proc. SPIE*, vol. 8500, Aug. 2012.

[15] A. R. Offringa *et al.*, "WSClean: an implementation of a fast, generic wide-field imager for radio astronomy," vol. 14, 2014.

[16] S. Bhatnagar, T. J. Cornwell, K. Golap, and J. M. Uson, "Correcting direction-dependent gains in the deconvolution of radio interferometric images," *Astronomy & Astrophysics*, vol. 487, no. 1, pp. 419–429, Aug. 2008.

[17] S. Jaeger, "The Common Astronomy Software Application (CASA)," in *Astronomical Data Analysis Software and Systems XVII*, R. W. Argyle, P. S. Bunclark, and J. R. Lewis, Eds., ser. Astronomical Society of the Pacific Conference Series, vol. 394, Aug. 2008, pp. 623–627.

[18] Tasse, C., van der Tol, S., van Zwieten, J., van Diepen, G., and Bhatnagar, S., "Applying full polarization A-Projection to very wide field of view instruments: An imager for LOFAR," *Astronomy & Astrophysics*, vol. 553, A105, May 2013.

[19] J. W. Romein, "An efficient work-distribution strategy for gridding radio-telescope data on GPUs," in *Proceedings of the 26th ACM international conference on Supercomputing*, Jun. 2012, pp. 321–330.

[20] D. Muscat, "High-Performance Image Synthesis for Radio Interferometry," PhD thesis, 2014.

[21] B. Merry, "Faster GPU-based convolutional gridding via thread coarsening," *Astronomy and Computing*, vol. 16, pp. 140–145, Jul. 2016.

[22] R. Bracewell, "The Fourier Transform and Its Applications," *New York*, vol. 5, 1965.

[23] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[24] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing," pp. 836–838, May 2008.

[25] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. 2007.

[26] H. Bal *et al.*, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *IEEE Computer*, vol. 49, no. 5, pp. 54–63, May 2016.

[27] B. Mort, *A simple interferometer baseline coordinate generator*, GitHub: SKA-ScienceDataProcessor/uvwsim, 2015.

[28] S. Oberman and M. Siu, "A High-Performance Area-Efficient Multifunction Interpolator," in *17th IEEE Symposium on Computer Arithmetic*, 2005, pp. 272–279.

[29] Advanced Micro Devices, Inc., *Southern Islands Series Instruction Set Architecture*. 2014.

[30] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," *Proceedings of the International Conference on Parallel Processing Workshops*, pp. 207–216, 2010.

[31] J. W. Romein and B. Veenboer, "Powersensor: A tool to analyze energy efficiency," unpublished.