Radio-Astronomical Imaging on Graphics Processors

Bram Veenboer, John W. Romein

Oude Hoogeveensedijk 4, 7991 PD, Dwingeloo, The Netherlands
{veenboer, 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 image-domain gridding (IDG) algorithm is a novel approach towards solving the most compute-intensive parts of creating sky images: gridding and degridding. It alleviates 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 a thorough performance analysis of this algorithm for an Intel Xeon CPU, Intel Xeon Phi, and GPUs from AMD and NVIDIA. We show that, by evaluating trigonometric functions in hardware, GPUs are both much faster and more energy efficient than a CPU or Xeon Phi. Furthermore, on GPUs, IDG is an order of magnitude faster and more energy efficient than traditional AW-projection. IDG on GPUs is the ideal candidate imaging technique for the SKA, as it meets the computational and energy constraints of the SKA Science Data Processor system.

Keywords: Radio Astronomy, Imaging, Gridding, GPUs

1. Introduction

At the present time, the world's largest radio telescope, the *Square Kilometre Array* (SKA) [1], is being developed. It will consist of hundreds of dishes and over 100,000 antennas. While it enables astronomers "to monitor the sky in unprecedented detail" [1], it can "easily push computing requirements into the exascale domain" [2]. However, even for existing telescopes such as the LOw-Frequency ARray (LOFAR) [3], 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 [4]. Only when the DDEs are corrected for, the high dynamic ranges imposed by the high sensitivity and large fields of view of the new generation of radio interferometers are achieved [5].

Creating sky images is especially computationally intensive: both efficient algorithms and processing methods are needed to meet the time and power constraints of instruments such as the SKA [6]. 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 [2].

Presently, the most widely used algorithm for gridding or degridding is W-projection [7]. This algorithm corrects for the so-called W-term in wide-field imaging, which causes artifacts around sources away from the phase center. The AW-projection [8] algorithm additionally corrects for DDEs (the A-term). Especially the correction for DDEs makes AW-projection expensive. The *Image-Domain Gridding* (IDG) [9] algorithm corrects for both the W-terms and the A-terms, and addresses the limitations of traditional methods currently in use by separating gridding from DDE correction. IDG is available as an open-source library [10] and is already being used to process the data for various radio telescopes, such as LOFAR [3] and the MWA [11].

We previously presented the first implementation of IDG in [12]: we analyzed its performance and energy efficiency on a reference CPU platform and GPUs from both AMD and NVIDIA. Although the performance on these GPUs was bound by (shared) memory bandwidth, we showed that both GPUs were much faster and more energy efficient than the CPU.

In this paper, we present the following new contributions: (1) we add an optimized implementation for a fourth architecture: the Intel Xeon Phi Knights Landing manycore processor; (2) we introduce our new GPU implementations that are no longer bound by shared memory bandwidth and are optimized for a wider variety of parameters. Consequently, IDG on GPUs now achieves even better performance for widely-used imaging use-cases; (3) we introduce two new execution schemes where both the CPU and the GPU are used to create very large sky images; (4) we demonstrate superior performance and energy efficiency of using this algorithm on GPUs over the traditional AW-projection imaging approach; (5) we provide an analysis of the impact of IDG on the future Square Kilometre Array radio telescope.

The rest of the paper is organized as follows: In Section 2, we provide the necessary background on radio-astronomical imaging. After discussing related work in Section 3, we describe the IDG algorithm in Section 4. In Section 5, we describe how the algorithm is most efficiently mapped onto various devices. In Section 6, we present experimental results, and uncover architectural and other features that are most relevant to performance and energy efficiency. In Section 7 we compare IDG to implementations of the W-projection and AW-projection algorithms, and in Section 8 we analyze the implications of our results for the SKA.

2. 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 the positions, intensity, and polarization of the sources. In this paper we focus on interferometers such as LOFAR [3] where the signals from separate receivers are combined to produce an image, thus not on single-dish telescopes like FAST [13].

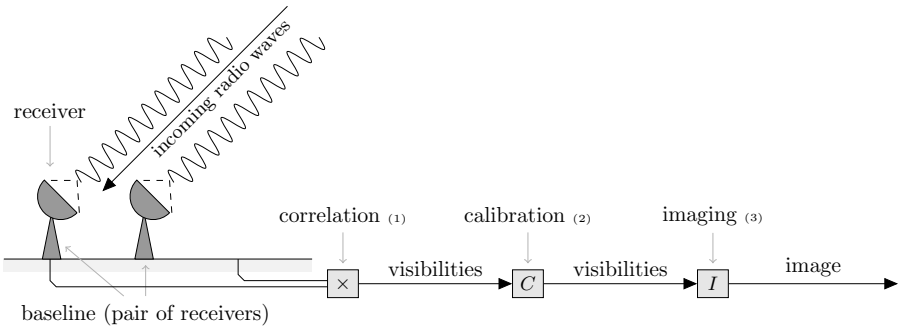


Figure 1: Incoming radio waves are received by a pair of receivers. Since the receivers are spaced apart, the signal is out of phase. The visibilities – the correlations of these signals – contain information on the amplitude and phase of the radio source.

As shown in Fig. 1, the creation of a sky image requires roughly three steps: (1) the digitized signals from pairs of distinct receivers are correlated to produce the so-called *visibilities*, (2) calibration is used to estimate and correct errors in the data, and (3) an *imaging* step converts visibilities into a *sky image*. Each visibility has an associated (u, v, w)-*coordinate* that depends on the location of the receivers with respect to the observed sky. Due to earth rotation, the (u, v, w)-coordinates of consecutive visibilities differ slightly. Therefore, every pair of receivers (called a *baseline*) contributes a track of measurements in the (u, v, w)-space, as detailed later in Fig. 9c.

Every antenna measures two orthogonal polarizations X and Y. Multiplying and integrating (correlating) the signals of antenna q and r for a short period of time (in the order of seconds) produces a single visibility for baseline (q, r). A visibility contains all four combinations XX, XY, YX and YY of a baseline, hence $V^{(q,r)} \in \mathbb{C}^{2\times2}$. All relevant parameters for imaging are summarized in Table 1.

The relation between visibilities and sky brightness, $B(l, m) \in \mathbb{R}^{2\times2}$, is given

Name	Symbol	Additional information
observation time	$t_{obs} \in \mathbb{R}$	
receivers	$R_{obs} \in \mathbb{N}$	# elements in observation
baselines (q, r)	$B_{obs} \in \mathbb{N}$	$B_{obs} = \binom{R_{obs}}{2}$
time	$1 \leq t \leq T_{obs}$	T_{obs} time steps
channel	$1 \leq c \leq C_{obs}$	C_{obs} (data) channels
visibility	$V \in \mathbb{C}^{2\times2}$	$T_{obs} \times C_{obs}$ per baseline
integration time	$t_{int} \in \mathbb{R}$	$t_{int} = \frac{t_{obs}}{T_{obs}-1}$

Table 1: Imaging parameters.

by the following measurement equation [14]:

$$V^{pq} = \iint_{\ell\, m} A_p(\ell, m) B(\ell, m) A_q^H(\ell, m) \frac{1}{n} e^{-2\pi i (u^{pq}\ell + v^{pq} m + w^{pq}(n-1))}\, d\ell\, dm, \quad (1)$$

where $\ell, m \in \mathbb{R}$ are the direction cosines of sky coordinates, $n = \sqrt{1-\ell^2-m^2}$, and $A_p(\ell, m), A_q(\ell, m) \in \mathbb{C}^{2\times 2}$ describe the aforementioned direction-dependent effects (DDEs), see [8].

A sky image (a map of sky brightness) is reconstructed from the measurement of many visibilities with distinct (u, v, w)-coordinates. In wide-field imaging, the w-coordinate of the visibility has to be taken into account. Consequently, there exists a three-dimensional Fourier-transform relation between the sampled data and the image [15]. A sky image can be constructed by performing an non-uniform discrete Fourier transform from visibilities to image space. This is a costly process, as the number of operations scales linearly with the number of visibilities and quadratically with the number of pixels in the image.

By using W-projection, the three-dimensional samples can be projected onto a uniform two-dimensional plane by *gridding* the visibilities [7]. In this operation a convolution kernel is applied to each of the visibilities, see the top-right panel of Fig. 3. In W-projection, gridding takes place in the frequency domain. After gridding the visibilities, an inverse FFT is applied to the grid to get the image.

Astronomical observations are affected by variable gain effects that are broadly classified as direction-independent effects (DIEs) and direction-dependent effects (DDEs) [8]. These gain effects can be estimated by a process known as *calibration*. The gains due to direction-independent effects (such as beam-patterns of the antennas) can be corrected for directly after calibration. The time-dependent direction-dependent gain effects (the *A-terms*, such as variations in the ionosphere) can only be corrected for during imaging.

In W-projection, the convolution kernels depend solely on the w-coordinate associated with the visibility. For AW-projection, these kernels additionally depend on time, frequency and baseline [8]. In practice, the convolution kernels are precomputed on an oversampled grid to accurately map the non-uniform visibilities onto a regular grid. The convolution kernels in W-projection and AW-projection gridding form a potentially large multi-dimensional data structure that scales quadratically in size with both the number of pixels in one dimension of the convolution kernel and the oversampling factor.

In the imaging step (see Fig. 2), imaging (gridding and inverse FFT) and prediction of visibilities (FFT and degridding) is typically repeated a couple of times to construct a sky image using a variant of the CLEAN algorithm [16].

3. Related work

Currently, the most widely used gridding approach is known as *W-projection* [7]. This algorithm corrects for the W-term by means of a convolution in Fourier space, but it does not correct for the DDEs (i.e., the A-terms). However, when

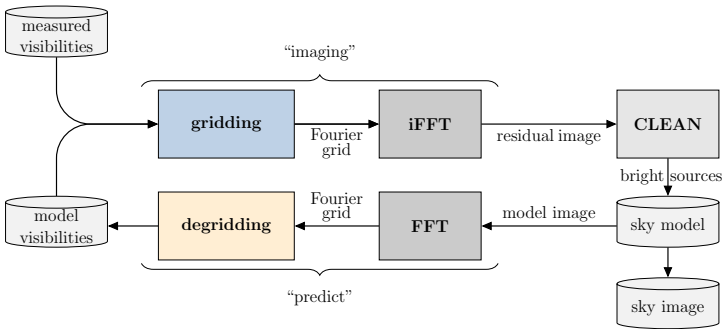


Figure 2: Imaging is an iterative process, with *image* and *predict* typically being the most time-consuming sub-parts of the pipeline. We implement the *gridding* and *degridding* steps using the Image-Domain Gridding algorithm.

antennas are spaced far apart from each other (to resolve the high spatial frequencies) and to observe large fields, the support of the W-terms can become large, making this technique inefficient and memory intensive [17]. One approach to reduce the support of the W-terms is to split the image into facets [15]. Furthermore, W-projection gridding can be extended by *W-stacking* [18, 19] or *W-snapshots* [18] to limit the support size of W-terms, at the cost of having to sort the visibilities.

The computational challenge increases further when the correction for DDEs is taken into account [17]. The correction of these so-called A-terms can be done in a similar manner to the W-term correction – called *A-projection*. Applying both corrections results in the so-called *AW-projection* gridding [8]. AW-projection is computationally expensive because the AW-terms have to be recomputed frequently. This can be alleviated by combining DDE correction with faceting such that a piecewise constant A-term is applied to each facet [20]. This method has the disadvantage of taking DDEs into account in a discontinuous manner in the image domain.

The *Image-Domain Gridding* (IDG) [9] algorithm corrects for both the W-terms and the A-terms and addresses the limitations of traditional methods currently in use by separating gridding from DDE correction. We presented the first implementation of IDG for CPUs and GPUs in [12], and in [21], we showed how we implemented IDG for an FPGA.

Most state-of-the-art imagers make use of one or more of the various gridding algorithms and their implementations: e.g., CASA [22] and LOFAR's AW-Imager [5] use W-projection and AW-projection, while WSClean [19] uses W-stacking only. We integrated IDG into WSClean such that all its features (data handling, deconvolution, etc.) are maintained, while the existing inversion (gridding) and predict (degridding) functionality are replaced by IDG.

The first efficient implementation of W-projection gridding on GPUs has been reported in [23], and has been further improved since [24, 25]. To the best of our knowledge IDG is currently the only GPU-accelerated imager that

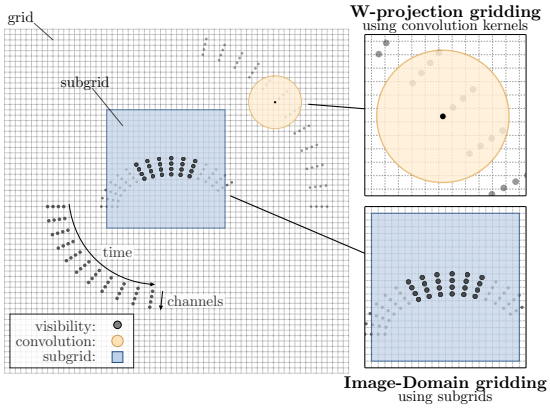


Figure 3: In traditional W-projection and AW-projection gridding, visibilities are gridded using convolutions in the frequency domain (top-right) as opposed to correcting the W-term and A-term effects in the image domain (bottom right). For the latter, neighboring visibilities (indicated by thick dots)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

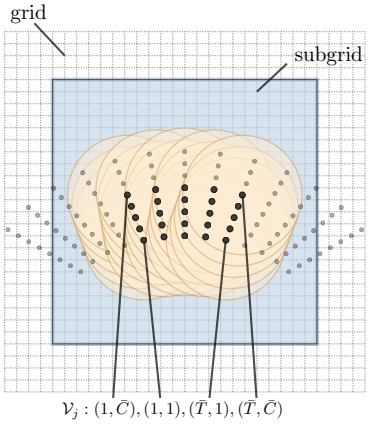


Figure 4: A subset of visibilities ($\mathcal{V}$, black dots), including their associated AW-projection convolution kernels (blue circles), is covered by a subgrid.

implements gridding and degridding with correction of DDEs.

4. Algorithm & Implementation

At the center of the Image-Domain Gridding (IDG) [9] algorithm are so-called *subgrids*, which represent low-resolution versions of the sky brightness for a small subset of visibilities, (see Figs. 3 and 4). IDG maps visibilities to subgrids by performing a direct Fourier Transform at subgrid resolution. This has the added benefit of allowing for cheap application of W-terms and A-terms.

4.1. The execution plan

Before gridding or degridding starts, an *execution plan* is generated that specifies the subgrid locations and associated visibilities. The process of positioning the subgrids to cover all visibilities is implemented in the form of a greedy algorithm that distributes the visibilities over subgrids. As depicted in Fig. 4, not only the visibilities need to be covered by the subgrids, but also the surrounding support of their associated AW-projection convolution kernels [9]. Thus, for each baseline, starting with the first integration period, and having $\bar{C}$ channels that can be covered by an $\bar{N}\times\bar{N}$ subgrid, we include as many integration periods as possible (each with $\bar{C}$ channels) until the support of the next integration period is no longer covered by the subgrid. We use $\bar{T}$ to denote the number of integration periods on 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 (u, v, w)-coordinates,

```
1  complex<float> subgrid[P][N̄ × N̄];
2  for i = 1..N̄ × N̄ do
3  |  float offset = compute_offset(s, i);
4  |  for t = 1..T̄ do
5  |  |  float index = compute_index(s, i, t);
6  |  |  for c = 1..C̄ do
7  |  |  |  float scale = wavenumbers[c];
8  |  |  |  float phase = offset - (index × scale);
9  |  |  |  complex<float> phasor = cexp(phase);
10 |  |  |  for p = 1..P do
11 |  |  |  |  complex<float> visibility =
   |  |  |  |    visibilities[t][c][p];
12 |  |  |  |  subgrid[p][i] += cmul(phasor, visibility);
13 |  |  |  end
14 |  |  end
15 |  end
16 end
17 apply_aterm(subgrid);
18 apply_taper(subgrid);
19 store(subgrid);
```

Algorithm 1: Gridding pseudocode that is executed for every subgrid s in the gridder kernel.

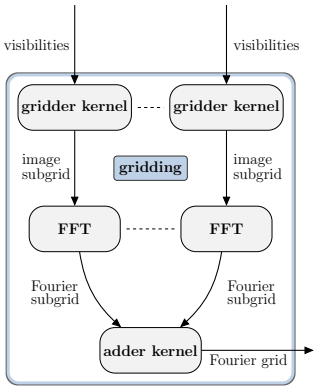


Figure 5: The Image-Domain gridding routine is a drop-in replacement for the gridding step in Fig. 2. Subgrids are independent of each other and can be computed in parallel. The adder kernel adds the subgrids onto the grid.

a *task*. The set of all n tasks is called the *work* and is generated by an execution plan.

4.2. Gridding and degridding

The work is processed by the gridder and degridder kernels in batches, using Algorithm 1 and Algorithm 2 for every task, respectively.

A direct summation of visibilities to the subgrid is computationally feasible as in practice the size of a subgrid $\bar{N} \times \bar{N}$ is 4-6 orders of magnitude smaller than the size of the grid $N \times N$. After direct summation, A-term correction is applied to each pixel of the subgrid. (The details of the A-term correction, which are not critical for performance, can be found in [9].) Since we have performed the corrections in the image domain, the subgrid has to be Fourier-transformed before the result is added to the larger $N \times N$ grid (i.e., four $\bar{N} \times \bar{N}$ FFTs per subgrid, one for each of the four polarizations).

The entire process of gridding and degridding is illustrated in Figures 5 and 6, respectively. 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 s; (2) the subgrids are Fourier-transformed, this step will be referred to as the *subgrid-fft*; (3) the transformed subgrids are added to the grid by the *adder kernel*.

The *cexp*(*phase*) evaluation in Line 9 of Algorithm 1 comprises one evaluation of *cos*(*phase*) and one evaluation of *sin*(*phase*). *cmul* denotes a complex multiplication, which comprises four real-valued multiply-add operations. Thus

```
1  apply_taper(subgrid);
2  apply_aterm(subgrid);
3  complex<float> visibilities[T̄][C̄][P];
4  for t = 1..T̄ do
5  |  for c = 1..C̄ do
6  |  |  float scale = wavenumbers[c];
7  |  |  for i = 1..N̄×N̄ do
8  |  |  |  float index = compute_index(s, i, t);
9  |  |  |  float offset = compute_offset(s, i);
10 |  |  |  float phase = (index × scale) - offset;
11 |  |  |  complex<float> phasor = cexp(phase);
12 |  |  |  for p = 1..P do
13 |  |  |  |  complex<float> pixel = subgrid[p][i];
14 |  |  |  |  visibilities[t][c][p] += cmul(phasor,
   |  |  |  |    pixel);
15 |  |  |  end
16 |  |  end
17 |  end
18 end
19 store(visibilities);
```

Algorithm 2: Pseudocode that is executed for every subgrid s in the degridder kernel.

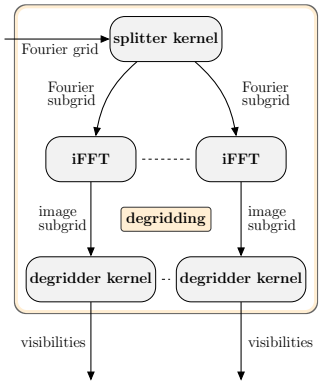


Figure 6: Degridding is the inverse operation of gridding, visibilities are computed taking a grid as input.

for every evaluation of *cexp*(*phase*) in Line 9, 17 real-valued multiply-add operations are performed, one in the computation of *phase* and 16 in the complex multiplication of *phasor* with visibilities and addition to the subgrid in Line 12. The offset (the position of the subgrid with respect to the center of the grid), the index (the position of a pixel in the subgrid) and the wavenumber (frequency dependent scaling factor) terms are used to compute a phase shift in Line 8. Before subgrids are stored (Line 19), correction for DDEs (Line 17) and a tapering function (Line 18, to suppress aliasing, see [9]) are applied.

In IDG, these convolution kernels and the tapering function are two-dimensional arrays where the size in number of pixels in one dimension is given by N_W. Consequently, the minimum number of pixels of the subgrid in one dimension $\bar{N} > N_W$. In practice, we use larger subgrids (e.g., $\bar{N} = 32$) so that multiple visibilities and their associated AW-kernels are covered (see also Fig. 4).

The *degridding* step is similar to the gridding step, but proceeds in reverse order: First, subgrids are extracted from the grid by the *splitter kernel*, 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 as shown in Algorithm 2.

4.3. Complexity

We now determine the complexity of Image-Domain Gridding. To this end we first establish the complexity of the steps that are performed in the gridding operation. The complexity of the gridder kernel for a single subgrid follows from

Algorithm 1:

$$\mathcal{O}_{gridder}(\bar{T}\bar{C}\bar{N}^2),$$

The complexity of a 2D FFT applied to a subgrid is:

$$\mathcal{O}_{fft}(\bar{N}^2 log\bar{N}^2),$$

and the complexity of adding a subgrid to a grid is:

$$\mathcal{O}_{adder}(\bar{N}^2).$$

If we assume an average of $\bar{V} = \bar{T}\times\bar{C}$ visibilities per subgrid, the complexity of gridding S subgrids is given by:

$$\mathcal{O}_{gridding}(S\bar{N}^2(\bar{V} + log\bar{N}^2 + 1)).$$

We will refer to $\bar{V}$ as the *visibility density* from now on. For a dataset with V_{obs} visibilities, S is given by: $S = V_{obs} \times \bar{V}^{-1}$, therefore:

$$\mathcal{O}_{gridding}(V_{obs}\bar{N}^2(1 + \bar{V}^{-1} log\bar{N}^2 + \bar{V}^{-1})).$$

The visibility density mostly depends on the size of the subgrid ($\bar{N}$), the size of the convolution kernel (N_W) and the (u, v, w)-coordinates associated with the visibilities.

The splitter kernel, 2D FFT, and degridder kernel in degridding have the same complexity as the 2D FFT, adder kernel and gridder kernel in gridding, respectively. The complexity of degridding is therefore the same as the complexity of gridding:

$$\mathcal{O}_{degridding} = \mathcal{O}_{gridding}.$$

5. Implementation details

We demonstrated the first implementations of the IDG algorithm on an Intel Xeon CPU, as well as on GPUs from both AMD and NVIDIA in [12]. We use well-known optimization techniques such as multi-threading, vectorization, data-reuse, prefetching, memory-coalescing, latency hiding, and loop transformations. Since then we applied additional optimizations throughout, and added optimized kernels for the Intel Xeon Phi architecture. In the remainder of this section, we provide a high-level overview of the most important optimizations and we will highlight significant changes and additions with respect to our previous work.

5.1. Single-precision gridding

It is not trivial to determine whether single-precision floating-point is sufficiently accurate to obtain the maximum achievable dynamic range in a sky image [26, 27, 28].

In a configuration as typically used in radio astronomy, IDG (using single-precision) is shown to provide comparable accuracy to classical W-projection

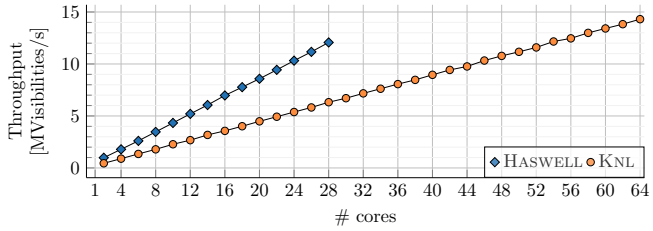


Figure 7: On Haswell and Knl, subgrids are distributed over logical cores and are processed in parallel. Throughput scales linearly with the number of physical cores used.

gridding [9]. Moreover, most compute architectures provide double the theoretical peak performance when using single-precision arithmetic instead of double-precision arithmetic. Therefore, we use single-precision floating-point operations (flops) for the computations and data-structures in all our implementations and omit the term "single-precision" from here on.

5.2. Intel Xeon and Intel Xeon Phi

We implemented IDG for Intel Xeon CPUs (Haswell architecture) and for the Intel Xeon Phi (Knights Landing) manycore processor in C++.

The Haswell architecture implements the AVX2 instruction set, which supports 8-element single-precision vector instructions, including fused-multiply add (FMA). The Knights Landing architecture additionally supports the AVX-512 instruction set, which doubles the vector length to 16 elements. The peak floating-point performance for Haswell and Knights Landing is only achieved when these vectorized FMA instructions are used. We therefore write the innermost loops of the gridder and degridder kernels in the form of a SIMD reduction using the desired (vector) intrinsics.

The Haswell architecture has a three-level cache hierarchy combined with DRAM, whereas Knights Landing has two cache levels and features 16 GB of high-bandwidth MCDRAM next to DRAM. As we will show in Section 6.2, these architectural differences do not significantly affect the performance of IDG.

On Knights Landing we found it beneficial for performance to reduce the number of visibilities per subgrid when creating the execution plan. This exposes more parallelism (more subgrids are created) so that the load can be better balanced over all the cores in this many-core processor.

Both Haswell and Knight Landing support hyper-threading, which allows multiple threads (two for Haswell, up to four for Knights Landing) to be executed on the same physical core. We use the number of threads that gives the best performance.

On both devices tasks (groups of subgrids with their associated visibilities) are distributed over all logical cores (threads) using OpenMP. In other words, each thread computes a subset of the subgrids. Given a sufficiently large number of subgrids (which is typically the case), this method scales linearly with the number of logical cores as we show in Figure 7.

5.3. GPU

We implemented IDG on GPUs in their native programming languages: CUDA for NVIDIA GPUs [29] and OpenCL for AMD GPUs [30]. Apart from differences in syntax and terminology, these programming languages offer the same basic functionality. In the remainder of this paper we adhere to the CUDA terminology.

GPUs have a number of *Streaming Multiprocessors* (SMs) with a number of *CUDA cores* each. Every SM additionally contains a register file, load-store units, dedicated caches and a software-managed cache (shared memory). We used shared memory buffers throughout the gridder and degridder kernels to process input data in batches. At this point, we also transpose the data to optimize the memory access pattern.

Compared to our previous GPU kernels (see [12]) we applied the following additional optimizations: (1) in the gridder kernel, every thread now computes multiple pixels at once so that every visibility loaded from shared memory is used to update multiple pixels; (2) the loop over channels in the degridder kernel (Line 5 in Algorithm 2) is unrolled, such that $index$ and $offset$ can be reused multiple times; (3) applying the A-term and tapering function is moved to a separate kernel. Although this requires an additional pass from and to device memory, the number of registers used in the gridder and degridder kernels is reduced which increases occupancy and improves performance.

The GPUs that we consider are connected to a host machine and have dedicated device memory, we thus have to copy any input and output data between host and device. We use double-buffering to overlap I/O and kernel execution.

5.4. Sine/cosine

The sine/cosine evaluation on Line 9 of Algorithm 1 (the gridder kernel) and on Line 11 of Algorithm 2 (the degridder kernel) is the most important factor that impacts gridding and degridding performance on the various architectures.

On both the Haswell and the Knights Landing architectures, the evaluation of sine/cosine is performed in software. We use Intel's Short Vector Math Library (SVML) or Vector Math Library (VML), whichever is fastest. Alternatives for the sine/cosine-computations, such as Libmvec (a open-source vector math library [31]) or a custom lookup table (based on integer arithmetic and configurable precision), perform less well.

On NVIDIA GPUs, the streaming multiprocessors contain a number of special-function units (SFUs) that implement the computation of both transcendental functions and interpolation in hardware [32]. The SFU provides fast approximations for sine/cosine with a maximum error of 2 units in the last place (ulps) [33]. In contrast, on AMD GPUs, the sine/cosine evaluations are performed by the same ALUs that also compute the FMAs, at a quarter of the rate while the maximum error is implementation-defined [34, 35]. Both NVDIAs and AMDs native instructions provide sufficient accuracy for IDG.

5.5. Scaling to large images

We implemented three distinct imaging schemes: *GPU-only*, *hybrid*, and *unified*.

In the *GPU-only* scheme, all operations are performed on the GPU. It supports images that fit in device memory. Every pixel requires 32 bytes (4 polarizations $\times$ 8 bytes for every complex floating-point number). An image of 40,000 $\times$ 40,000 pixels, for instance, thus is about 48 GB in size, which is more than most GPUs currently available provide.

The *hybrid* imager performs the gridder and subgrid-fft on the GPU and the addition to the grid on the host. As the image is kept in host memory, the maximum size of the image is not bound by the size of the device memory.

NVIDA GPUs from the Pascal generation and newer support Unified Memory, which provides a single memory address space between any processor (CPU or GPU) in the system. This is implemented using on-demand page migration. When the GPU addresses a memory page that is not resident in device memory, a page fault is generated and the corresponding page is migrated from host to the device, possibly evicting another page. The grid is allocated on the host, while the adder kernel is executed on the GPU. Instead of having to explicitly copy the parts of the grid accessed in the adder kernel, the pages are automatically migrated on demand.

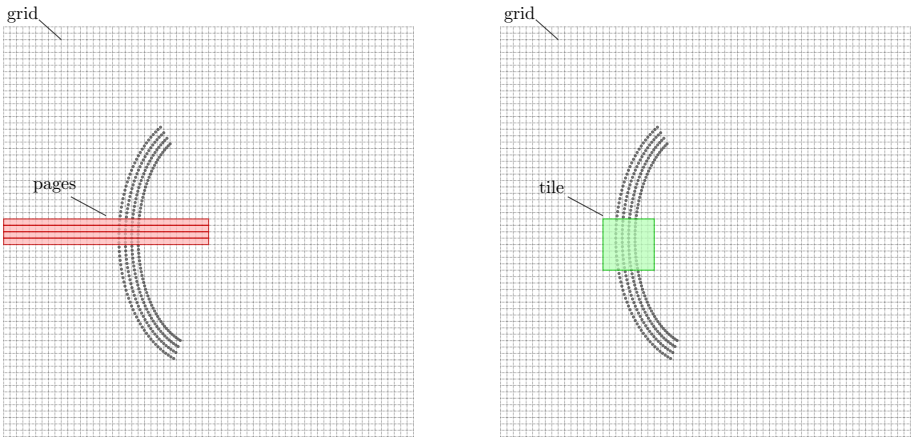
This mechanism is an excellent match for the irregular yet localized memory accesses encountered when adding subgrids onto the grid. There is no need to keep track in software which parts of the grid are being updated and should be copied to or from GPU memory, as this is all transparently resolved by the CUDA Unified Memory runtime greatly simplifying the application. We will refer to the implementations that use this feature as the *unified* imager. We use a tiled memory layout for the grid such that pixels that are close together in the grid are also close together in memory, reducing the number of pages migrated when accessing the pixels corresponding to a subgrid, see also Fig. 8.

6. Performance analysis

In this section, after describing the experimental setup, we analyze the performance of our implementation of the IDG algorithm (called IDG subsequently) for each architecture in detail.

6.1. Experimental setup

All experiments were executed using the hardware listed in Table 2. We refer to this hardware as HASWELL (a dual-socket system with Intel Haswell-EP processors), KNL (a system with one Intel Xeon Phi Knights Landing processor), VEGA (an AMD Vega Frontier Edition GPU) and PASCAL (an NVIDIA Tesla P100 GPU). VEGA is hosted by a dual-socket Haswell-EP system, where the GPU is connected using PCIe 3.0. HASWELL, KNL and VEGA are part of the DAS-5 cluster [36]. PASCAL is hosted by a Minksy system that is part of the



(a) Unified Memory (default) (b) Unified Memory with tiling

Figure 8: With on-demand page migration, the Unified Memory subsystem copies pages of memory (indicated in red) between CPU and GPU memory. In the default setting (on the left), pixels that are (vertically) close together in the grid are covered by different pages. This leads to inefficient use of the migrated pages as many pixels are not used. By tiling the grid, pixels that are close together in the grid are also close together in memory. Therefore, the number of page migration required for accessing neighboring pixels (e.g. in a tile, indicated in green), is significantly lower than in the default setting.

Table 2: The Intel Haswell-EP CPU, Intel Knight Landing Xeon Phi, AMD Vega GPU and NVIDIA Pascal GPU used in our experiments. We refer to these device as Haswell, Knl, Vega and Pascal

Model	Architecture	Peak[a)] (TFlop/s)	Mem size (GB)	Mem bw (GB/s)	TDP (W)
Intel Xeon E5-2697v3	Haswell-EP	2.60	$\leq$1536	136	290
Intel Xeon Phi 7210	Knight Landing	5.32	$\leq$384	102	215
NVIDIA Tesla P100	Pascal	10.6	16	732	300
AMD Vega FE	Vega	13.1	16	483	300

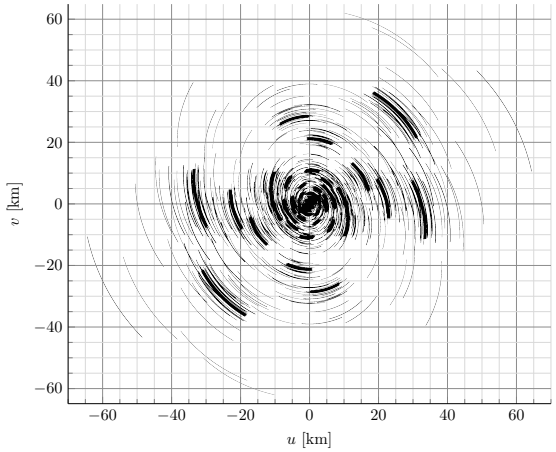
[a)] Single-precision floating-point performance, turbo-mode enabled.

Parameter	Value
R_{obs}	120
B_{obs}	7,140
T_{obs}	8,192
C_{obs}	16
t_{int}	0.9

(a) Observation parameters.

Parameter	Value
N_W	9
$\bar{N}$	32
N	8,192

(b) Imaging parameters.



(c) SKA-1 Low (u, v, w)-coverage.

Figure 9: We use (u, v, w)-data generated using a subset of proposed SKA1-Low station coordinates [41]. Every baseline (pair of stations) contributes one track in this plane. The resulting (u, v)-plane is filled with both 'short baselines' and 'long baselines' and therefore representative for a wide range of imaging use-cases.

JURON cluster [37]. A Minsky system is based on the IBM Power 8 architecture, and supports the high-bandwidth NVLink bus between the CPU and the GPU.

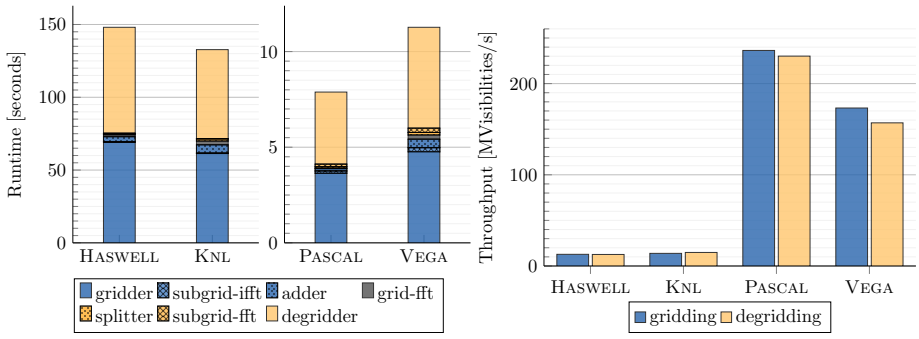
For Haswell and Knl, we used Intel's compiler (version 19.0) together with MKL (version 2019.0); for Vega, we used the AMD APP SDK 3.0 OpenCL runtime and GPU driver version 2527.4; for Pascal, we used CUDA 9.2.88 and GPU driver 410.48.

We use LIKWID [38] to measure the energy usage of both the package (CPU cores and caches) and the DRAM for Haswell and Knl. For Vega, we measure the energy consumption of the full PCIe device, using PowerSensor [39]. Pascal uses a mezzanine connector unsuitable for PowerSensor and we therefore resort the NVIDIA Management Library (NVML) to measure energy consumption.

The (u, v, w)-coordinates in a dataset determine the subgrid configuration, and hence the number of visibilities per subgrid and the total number of subgrids. We created a representative benchmark using proposed antenna coordinates for the SKA1-Low telescope – the phase 1 subset of the SKA covering the low frequency spectrum [40]. We use the dataset and imaging parameters as shown in Table 9a and 9b, respectively. The (u, v)-plane for this dataset is illustrated in Fig. 9c. The A-terms (in this benchmark, all set to identity) are updated every 256 time steps. Therefore, $\lceil \bar{T} \rceil = 256$. We reduce $\bar{T}$ when this increases performance. Furthermore, $\bar{C} = C$.

6.2. Performance comparison

For a single imaging cycle shown in Fig. 2 we present, respectively, the execution time for all kernels and overall throughput (measured in MVisibilities/s



(a) Runtime distribution.

(b) Throughput comparison.

Figure 10: Distribution of runtime for one imaging cycle and comparison of throughput for gridding and degridding.

$= 10^6$ Visibilities/s) in Fig. 10a and Fig. 10b.

For Haswell and Knl we additionally show in Fig. 7 that throughput scales linearly with the number of cores used. On Haswell, hyper-threading has a negligible impact on performance. On Knl, we use two threads per core as this gives the highest throughput.

Both GPUs complete the task roughly an order of magnitude faster than Haswell and Knl. For all architectures, runtime is dominated by the gridder and degridder kernels (in all cases, more than 90% of the runtime). Since the impact on the execution time of all other kernels is limited, we focus on the gridder and degridder kernels for the remainder of this analysis.

6.3. Roofline analysis

In Fig. 11 we show roofline plots [42], where an operation (an *op*) is defined as one of the following: $+, -, *, \sin(), \cos()$. The dashed lines correspond to the upper bound for peak performance for the instruction mix of 17 FMA instructions and 1 sine/cosine evaluation as found in the gridder and the degridder kernels, see [12] for details.

Both the kernels on Haswell and Knl, given the limitations of hardware *and* the supporting mathematical library, achieve close to optimal. Furthermore, since Knl is compute bound, we found no advantage of using high-bandwidth MCDRAM over DRAM.

For our GPU implementations, unrolling over pixels (in the gridder) and over visibilities (in the degridder) (see Section 5.3) increases the operational intensity. In contrast to the results shown in [12], the GPU kernels are therefore no longer bound by the bandwidth of shared memory. For Vega, the limiting factor, like on the Haswell and Knl is the evaluation of sine and cosine.

Only on Pascal the measured performance approaches the theoretical peak performance: 85% and 80% for the gridder and degridder kernel, respectively. The difference between measured performance and theoretical peak performance

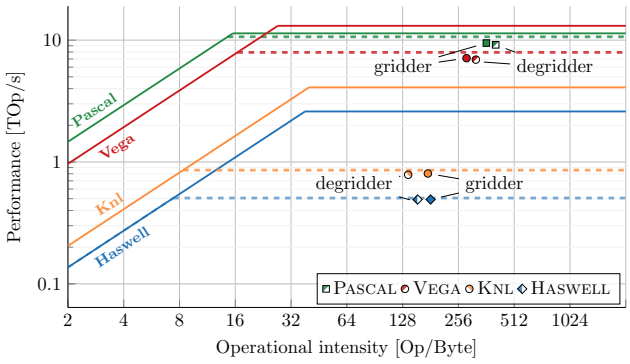


Figure 11: Roofline analysis: one operation is $+, -, *, \sin()$ or $\cos()$. Peak performance is only achieved if non-masked FMA instructions (two operations) are used exclusively. While the operation count is known exactly, the data movement is measured.

is mainly due to memory latencies that are not (fully) hidden. These latencies are the result of a suboptimal memory access pattern for the A-terms and (u, v, w)-coordinates. This effect is more pronounced in the degridder kernel, where (u, v, w)-coordinates are not shared between threads. The occupancy in the gridder and degridder kernels is too low to hide all memory latencies, but the high register usage prevents the GPU from achieving a higher occupancy.

While the performance of the gridder and degridder kernels on Haswell, Knl and Vega is bound by sine/cosine evaluations, on Pascal it is pretty close to the theoretical peak and mostly limited by memory latencies. These results indicate that Pascal is a very suitable architecture for IDG as it offers a balanced mix of floating-point units, SFU units and shared-memory.

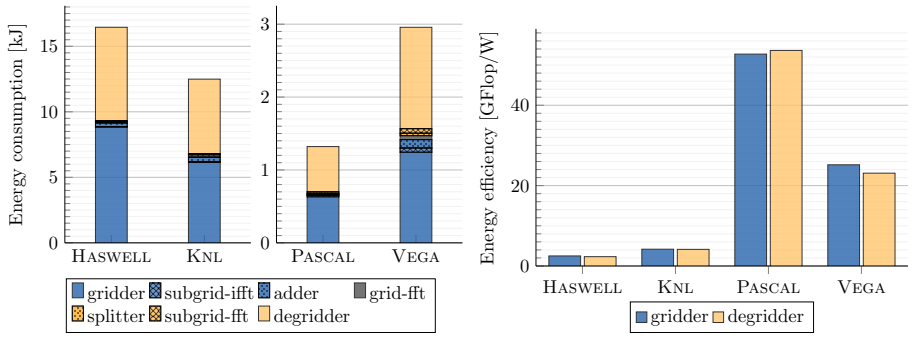
6.4. Energy efficiency

The total energy consumed during the execution of a single imaging cycle is shown in Fig. 12a. As most of the time is spent in the gridder and degridder kernels (Fig. 10a), the bulk of the energy is correspondingly spent in these kernels. Thus, also in terms of energy efficiency, both GPUs outperform Haswell by more than an order of magnitude. Knl proves to be more energy-efficient than Haswell, by consuming 30% less energy in this task.

In Fig. 12b we present the energy efficiency of the individual kernels. Pascal is the most energy-efficient GPU: for the gridder and degridder kernels, it achieves about 53 GFlops/W. Second, but still with about 25 GFlops/W for both kernels, comes Vega. The other architectures lag far behind the GPUs. Haswell achieves only about 2.5 GFlops/W, while Knl is slightly more efficient with about 4.1 GFlops/W.

6.5. Scaling to large images

As depicted in Fig. 2, both gridding and degridding are performed in order to execute an imaging cycle. Therefore, we define *imaging throughput* as



(a) Energy-consumption distribution.

(b) Energy-efficiency comparison.

Figure 12: Distribution of energy-consumption for one imaging cycle and comparison of energy-efficiency for the gridder and degridder kernels.

the throughput for gridding and degridding combined. We show results for our Hybrid and Unified imagers in Fig. 13, and compare them to GPU-only imaging.

Up to grid sizes of about $16{,}000 \times 16{,}000$ pixels the entire grid fits in GPU memory. The GPU-only imager achieves 235 and 230 MVisibilities/s for gridding and degridding, respectively, resulting in an imaging throughput of 116 MVisibilities/s. For the largest images that the GPU-ony imager can create, we have to reserve the majority of the device memory to store the grid. Consequently, the amount of memory available for other data (e.g. visibilities and subgrids) is limited and this causes a minor performance degradation as this is not sufficient to keep all SMs occupied all the time.

For images that do not fit in GPU memory we need to use either the hybrid or the unified imaging scheme. We observe that the hybrid imaging scheme performs lower than the GPU-only scheme. This can be attributed to the adder and splitter kernels that run slower on the CPU than on the GPU, as we show in Fig. 14a: in all cases the computation on the CPU takes longer than the computation on the GPU.

There is a minor performance difference between the GPU-only and the unified imaging routines for grid sizes up to $16{,}000 \times 16{,}000$ pixels. Since the page migrations are overlapped with computation, the performance impact is low. These results demonstrate that the performance of CUDA Unified Memory and NVLink is excellent.

For larger images, not all parts of the grid covered by subgrids fit in device memory. Consequently, the Unified Memory runtime spends more time moving pages from and to GPU device memory than for the smaller grids – resulting in a loss of performance that scales with the size of the image. We take a closer look at the runtime distribution for the Unified imaging in Fig. 14b.

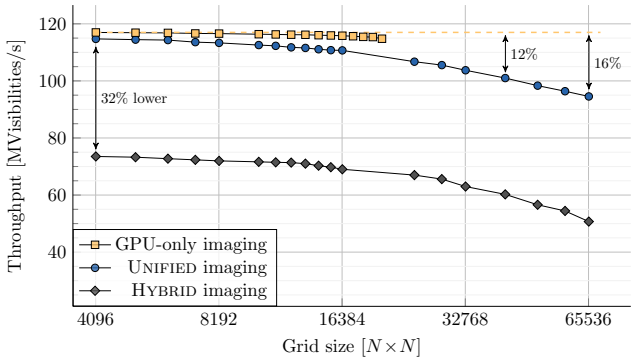
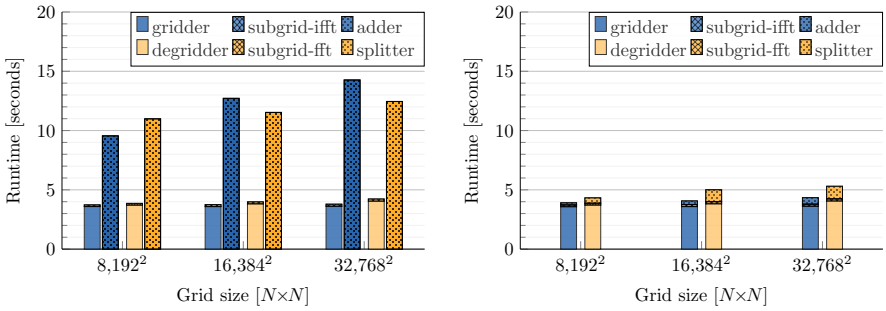


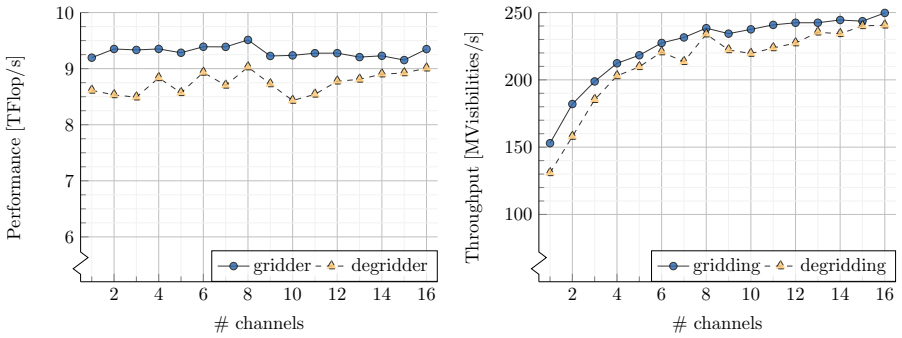
Figure 13: Throughput for the Hybrid and Unified imagers. While the Unified imager achieves almost identical throughput to the GPU-only imager, it also allows for larger images at a modest throughput decline (about 12% for 40,000 × 40,000 pixel images). The Hybrid imager performs about 32% lower than Unified, this is mainly due to the kernels that run on the host, see also Fig. 14a.



(a) Hybrid runtime distribution.

(b) Unified runtime distribution.

Figure 14: In Hybrid gridding (Fig. 14a), the gridder and subgrid-ifft are executed on the GPU, while the host in the meantime executes the adder, and vice versa for degridding. For both gridding and degridding, the computation on the host takes longer than the computation on the GPU and thus limits throughput. In the case of Unified (Fig. 14b), the runtime for the adder and splitter kernels increases for larger images, as more tiles of the grid have to be migrated between host and device memory. For both imagers, the throughput is affected by a reduced visibility density for large images.



(a) Kernel performance.

(b) Routine throughput.

Figure 15: The gridder and degridder performance is relatively consistent for all values of $\bar{C}$. The degridder kernel on average performs about 7% lower than the gridder kernel. For large values of $\bar{C}$, throughput is mostly determined by the time spend in the gridder and degridder kernels (see Fig. 10a). For small values of $\bar{C}$, the time spend in the other kernels (e.g. the subgrid-fft and the adder kernel for gridding) negatively affects throughput. In case of spectral-line imaging (where $\bar{C} = 1$), throughput is reduced to about 60% of the throughput for cases where $\bar{C}$ is larger.

6.6. Imaging a different number of channels

So far, we have shown results for $\bar{C} = 16$, but the gridder and degridder kernels also achieve good performance for different settings of $\bar{C}$, as we show for PASCAL in Fig. 15a. This is an appealing property, for instance for spectral-line imaging, where $\bar{C} = 1$. As shown in Fig. 15b, throughput is affected for small values of $\bar{C}$. As we showed in the complexity analysis of IDG in Section 4.3, this can be explained by looking at the visibility density ($\bar{V}$): for small values of $\bar{C}$, the runtime becomes dominated by the time spent in the subgrid-fft and in the adder kernel (for gridding) or splitter kernel (for degridding).

With a fixed setting of $\bar{N}$ and $C_{obs} > \bar{C}$, not all C_{obs} channels might fit on a single subgrid. In this case IDG will create multiple subgrids with at most $\bar{C}$ channels each to cover all C_{obs} channels. The throughput for processing of C_{obs} channels will therefore be comparable to the throughput for processing $\bar{C}$ channels. For large values of C_{obs}, the dataset is typically split into multiple so-called *subbands* that are processed independently, possibly even using multiple machines.

7. A comparison with AW-projection

In this section, we compare IDG with the W-projection implementation introduced in [23]. The aim of this comparison is to estimate how IDG performs in comparison with traditional W-projection. We provide background in section 7.1 and refer to it as WPG from now on. One of the distinguishing features of IDG is the support for direction-dependent corrections (A-term correction) and we would like to compare its performance to AW-projection. However, as

no efficient AW-projection implementation was available, we extended WPG to include A-term correction to make a complete comparison. We refer to this code as AWPG.

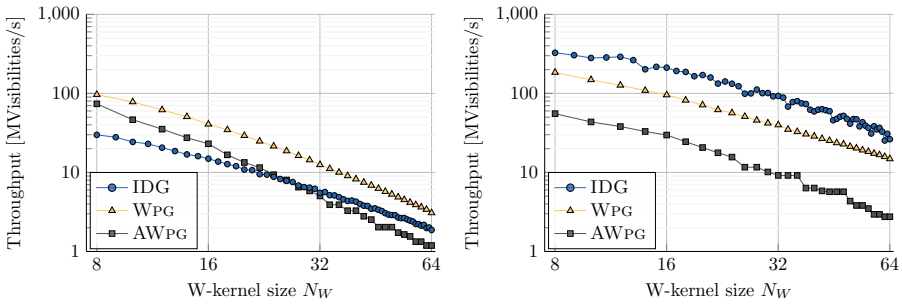
7.1. Background

The minimum size of the W-kernels $N_W \times N_W$ is determined by the observation settings (the instrument, the field of view, the target location, etc.) [8, 5]. Furthermore, the size of the W-kernel depends on the baseline length. For LOFAR, the W-kernel can be as large as 500×500 pixels for the longest baseline. In practice, *W-stacking* is used to limit N_W to small values in all situations (e.g., $N_W \leq 16$) [18, 19].

7.2. Experimental setup

In order to implement AWPG, we modified WPG taking the following properties for AW-projection gridding into account: (1) The AW-kernel is different for every baseline; (2) the AW-kernel changes after a (fixed) number of time steps; (3) the AW-kernel is constructed by combining the W-kernel with the A-term; (4) the A-term, like in IDG, is provided in the image domain; (5) a Fourier transformation is performed to get the AW-kernel into the Fourier domain. With these properties in mind, we modify WPG to create AWPG.

In the original WPG code, the W-kernel is generated only once for the entire observation. Due to modifications (1) and (2), this is not possible for AWPG. The amount of (device) memory required for the full AW-kernel is prohibitively large and we therefore interleave the computation of the AW-kernel with gridding. We implement the computation of the AW-kernel according to property (4) and (5), a Fourier transformation of the AW-kernel is performed before the gridder kernel is executed. Pseudocode for AWPG is shown in Algorithm 3. Like in IDG, the FFT is performed using an FFT library (e.g. using FFTW or Intel MKL). For the GPU implementation of AWPG, the Fourier transformations are performed on the GPU, using cuFFT.



(a) CPU throughput

(b) GPU throughput

Figure 16: On HASWELL, WPG is the fastest gridder (but does not correct for DDEs). IDG performs similar to AWPG for large kernel sizes. On PASCAL, IDG outperforms both AWPG and WPG, for all kernel sizes.

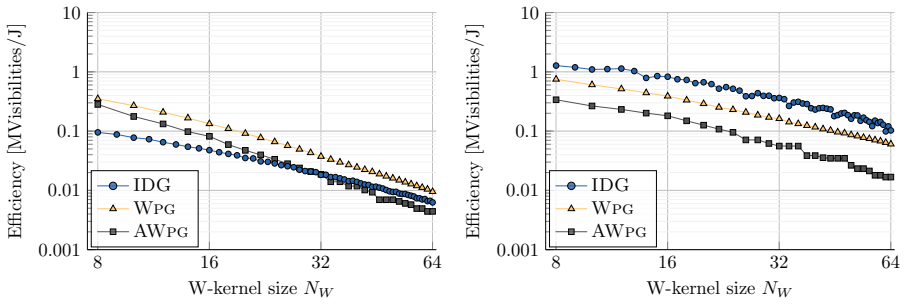
```
1  for bl = 1..B_obs do
2  |  for ts = 1..T_obs..T̄ do
3  |  |  int N_W = compute_kernel_size(bl, ts);
4  |  |  complex<float> wkernel = compute_wkernel(N_W);
5  |  |  complex<float> awkernel = compute_awkernel(bl, ts, N_W, wkernel);
6  |  |  awkernel = apply_2d_fft(awkernel);
7  |  |  for t = 1..T̄ do
8  |  |  |  for c = 1..C_obs do
9  |  |  |  |  for k = 1..(N_W × oversampling)^2 do
10 |  |  |  |  |  complex<float> weight = awkernel(k);
11 |  |  |  |  |  for p = 1..P do
12 |  |  |  |  |  |  int idx = compute_index(bl, t, c, k);
13 |  |  |  |  |  |  complex<float> visibility = visibilities[t][c][p];
14 |  |  |  |  |  |  grid[p][idx] += cmul(weight, visibility);
15 |  |  |  |  |  end
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

Algorithm 3: This pseudocode for AWPG illustrates two main difference between AWPG and IDG: (1) in AWPG kernels are computed prior to gridding, while IDG computes them on-the-fly; (2) AWPG grids each visibility directly onto $N_W \times N_W$ pixels in the grid, while IDG grids onto subgrids of $\bar{N} \times \bar{N}$ pixels. In this pseudocode $\bar{T}$ denotes the length of a timeslot for which the same A-term is applied. Like in [23], we use an oversampling rate ($oversampling = 8$).

7.3. Performance comparison

With the above in mind, Fig. 16a and 16b show the performance on HASWELL and PASCAL respectively for various values of N_W.

On HASWELL, WPG outperforms IDG by quite a large margin, for all kernel sizes, but recall that WPG does not correct for DDEs (affecting image quality).



(a) CPU efficiency

(b) GPU efficiency

Figure 17: In relative terms, these energy-efficiency results match the throughput results in Fig 16. On HASWELL, WPG is the most energy-efficient gridder. IDG is more energy efficient than AWPG for large kernel sizes. Furthermore, also in terms of energy efficiency, on PASCAL, IDG is more energy efficient than both WPG and AWPG.

Both curves show a decline in throughput as the size of the kernel (and hence the amount of computations per visibility) increases. The throughput of AWPG is about 2–3× lower than the throughput of WPG. This is mainly caused by the additional time spend in the Fourier transformation to compute the AW-kernel. The performance of the FFT is highly sensitive to the size of the transformation. Therefore, we use a larger kernel size when this increases overall throughput. We observe that on HASWELL, IDG is faster than AWPG for larger values of N_W. On PASCAL, WPG on average performs about 4× better than AWPG. The highest overall throughput is achieved on PASCAL using IDG. On PASCAL, IDG outperforms AWPG by almost an order of magnitude.

Improvements to the GPU implementation of WPG can increase its performance twofold from roughly 28% of the peak floating-point performance (which we measured in our tests) to 55% in the best case [25]. This implementation of W-projection is available at [43]. Even if WPG and AWPG on PASCAL would be twice as fast, they would still be outperformed by IDG.

7.4. Energy efficiency comparison

We measured the energy consumption for WPG, AWPG and IDG on HASWELL and PASCAL and found some notable differences, see Fig. 17. On HASWELL, WPG consumes the most energy, followed by IDG (6% lower) and AWPG consumes the least amount of energy (12% lower than WPG). However, WPG offsets this higher energy consumption with its throughput. In terms of visibilities processed per Joule consumed, on HASWELL, WPG is therefore the most energy-efficient imager. On PASCAL, the instantaneous energy consumption of WPG and IDG approaches the Thermal Design Power (TDP) of the device, while AWPG consumes about 30% less energy. This is due to the FFTs that AWPG performs, which consume significantly less energy in comparison to the gridder kernel. Overall, IDG on PASCAL is the most energy-efficient imager.

8. Impact on the Square Kilometre Array

The SKA will require a large amount of computational power at high energy efficiency [44]. To meet these requirements, we need better energy efficiency and compute capabilities than current technology provides: as discussed in [6], simply waiting for the next generation of hardware alone will not be sufficient. A co-design between compute hardware and algorithms is needed to meet these demands. As we demonstrated in Section 6, the IDG algorithm runs highly efficient on an NVIDIA GPU. Furthermore, in Section 7 we showed that on this GPU, IDG outperforms an imager based on the AW-projection algorithm by roughly an order of magnitude. With these results in mind, we investigate the impact of this work on the SKA.

In the following sections, we first determine the rate at which visibilities need to be imaged for the high-priority science objectives (HPSOs) identified in [45]. Secondly, we analyze whether this data rate can be processed using IDG.

8.1. Required data rates

The SKA community uses a parametric model [46] to analyze processing requirements for the SDP compute platform. This model is available online at [46]. We use the numbers from the "2019-06-20-2998d59_hpsos.csv" analysis of imaging HPSOs to establish an estimated imaging visibility rate of around 1264 MVis/s. If we consider an average of 10 imaging cycles (see [47] for why this is needed) and take into account that SKA1 Low might only be doing imaging observations half the time [45], the required processing rate becomes 6.3 GVisibilities/s.

This data rate does not take baseline-dependent averaging (BLDA) into account. Using BLDA, the overall number of visibilities could reduce by an order of magnitude. This could lead to having few visibilities per subgrid for the shortest baselines. The results in Section 6.5 suggest that this has a negative effect on the throughput of IDG. For the remainder of this discussion we use the aforementioned processing rate *without* BLDA and assume that the negative impact on throughput in case of BLDA will be offset by the lower overall visibility rate.

8.2. Science Data Processor (SDP)

The SKA consortium plans to build two main SDP systems, one for SKA-1 Low and one for SKA-1 Mid. According to the most recent plans, these processing facilities will initially provide a total double-precision peak performance of 50 PFlop/s [48]. The compute power will be distributed equally among the two sites and will later be extended to a combined 260 PFlop/s. The power cap for the final system will be 5 MW per site.

Gridding and degridding is estimated to contribute about 13% to the total SDP computation [49]. This implies that 0.13×50 PFlop/s ≈ 6.5 PFlop/s of the total SDP compute budget is available for gridding and degridding and that these operations may consume up to $(\frac{50}{260}) \times 5$ MW $\times\, 0.13 \approx 125$ kW per site.

In this analysis, we use the performance and energy-efficiency results of the NVIDIA Tesla P100 GPU (Pascal). By the time that the SKA will be build, we assume that the latest generation of GPUs will be used. These will likely be faster and more energy-efficient compared to the GPUs available today. Since the theoretical peak-performance in *double-precision* for Pascal is 5.3 TFlop/s, $\frac{6{,}500 \text{ TFlop/s}}{5.3 \text{ TFlop/s}} \approx 1226$ Tesla P100 GPUs would be needed to provide 6.5 PFlop/s of compute power. Next, we use the measured throughput for IDG on Pascal to estimate how many GPUs are needed to process the data rate of 6.3 GVisibilities/s.

8.3. IDG for SKA

The average throughput for Unified on Pascal for large images (40,000 × 40,000 pixels) is about 0.20 GVisibilities/s (see Fig. 13). Since both gridding and degridding have to be performed every imaging cycle, the combined imaging throughput is 0.10 GVisibilities/s. The average GPU power consumption in this setting is 255 W.

This means that approximately $6.3/0.1 = 63$ Tesla P100 GPUs are needed to process all input data, only a fraction of the 1226 GPUs available. The total power consumption for all these GPUs adds up to $63 \times 255\text{W} = 16$ kW, which is well within the power budget of 125 kW per site.

Even given that imaging throughput is about halved for spectral-line imaging (see Fig. 15a), and taking more imaging cycles and/or unforeseen bottlenecks in other parts of the imaging pipeline into account, there is headroom to still meet the constraints. Moreover, future generation GPUs will likely be faster and more energy-efficient than Pascal, which will make it even easier to remain within the compute and power constraints.

Next, we extrapolate our results for AWpg on Pascal from Section 7. On Pascal, IDG gridding on average is about an order of magnitude faster and almost 7× more energy-efficient than AWpg gridding. Under the assumption that AWpg is extended to support degridding (with degridding about as fast as gridding) and that support for large images (e.g. 40,000 × 40,000 pixels) is added to AWpg, the number of GPUs required for SKA would be approximately $9.6/0.01 = 960$. The power consumption of these GPUs would be about $960 \times 175 = 168$ kW. While AWpg meets the SKA requirements in terms of GPUs needed, the power consumption would be excessive given the power budget of 125 kW.

Idg runs much more efficiently than 10% of the peak performance generally considered for SDP processing [46]. Our analysis reveals that even in the worst case (for spectral-line imaging) an imager based on IDG using NVIDIA Tesla P100 GPUs would meet SKA compute and power budget.

These results indicate that IDG solves the most demanding parts of imaging (gridding and degridding with A-term correction), bringing us a big step closer towards making imaging for the SKA a reality.

9. Conclusions

We demonstrated our implementation of the IDG algorithm on four distinct architectures: Intel Xeon (Haswell), Intel Xeon Phi (Knights Landing), NVIDIA Pascal and AMD Vega. In particular for the latter two, the graphics processors, the IDG algorithm maps elegantly onto the underlying hardware. Due to hardware support for sine/cosine evaluations, on NVIDIA GPUs, our code achieves up to 85% of the floating-point peak performance and an energy efficiency of over 50 GFlop/W.

The comparison with a traditional imaging algorithm illustrates that IDG on GPUs exceeds the performance and energy-efficiency of the simpler W-projection gridding, while providing the functionality of the more challenging AW-projection gridding.

By efficiently mapping the IDG algorithm onto GPUs, we have addressed the largest computational challenge in the imaging pipeline of the modern radio telescopes: our results show that IDG meets the performance and energy efficiency requirements needed for the future Square Kilometre Array.

Acknowledgements

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), the EU Horizon 2020 research and innovation programme under grant no 754304 (DEEP-EST) and by the NWO through NWO-M (DAS-5 [36]) and Open Competition (617.001.204) grants. The European Commission is not liable for any use that might be made of the information contained in this paper. The authors would like to thank Bas van der Tol, André Offringa and Matthias Petschow for their support.

References

[1] The SKA Organisation, Square Kilometre Array, https://www.skatelescope.org/ (2018).

[2] R. Jongerius, et al., An End-to-End Computing Model for the Square Kilometre Array, IEEE Computer 47 (9) (2014) 48–54.

[3] M. P. van Haarlem, et al., LOFAR: The LOw-Frequency ARray, Astron. Astrophys. 556 (2013).

[4] R. J. van Weeren, et al., LOFAR facet calibration, The Astrophysical Journal Supplement Series 223 (1) (2016) 2.

[5] C. Tasse, et al., Applying full polarization A-Projection to very wide field of view instruments: An imager for LOFAR, Astron. Astrophys. 553 (2013) A105.

[6] E. Vermij, et al., Challenges in exascale radio astronomy: Can the SKA ride the technology wave?, International Journal of High Performance Computing Applications 29 (1) (2015) 37–50.

[7] T. J. Cornwell, K. Golap, S. Bhatnagar, The Non-coplanar Baselines Effect in Radio Interferometry: The W-projection Algorithm, IEEE J. Sel. Topics Signal Process. 2 (5) (2008) 647–657.

[8] S. Bhatnagar, et al., Correcting direction-dependent gains in the deconvolution of radio interferometric images, Astron. Astrophys. 487 (1) (2008) 419–429.

[9] S. van der Tol, B. Veenboer, A. R. Offringa, Image-Domain Gridding: a fast method for convolutional resampling of visibilities, Astronomy & Astrophysics 616 (2018) A27.

[10] ASTRON Netherlands Institute for Radio Astronomy, Image-Domain Gridding, GitLab: astron-idg/idg (2019).

[11] S. J. Tingay, et al., The Murchison Widefield Array: The Square Kilometre Array Precursor at Low Radio Frequencies, Publications of the Astronomical Society of Australia 30 (Jan. 2013).

[12] B. Veenboer, M. Petschow, J. W. Romein, Image-Domain Gridding on Graphics Processors, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2017, pp. 545–554.

[13] R. Nan, et al., The five-hundred-meter aperture spherical radio telescope (FAST) project, Int. J. Mod. Phys. D 20 (06) (2011) 989–1024.

[14] O. M. Smirnov, Revisiting the radio interferometer measurement equation, Astronomy & Astrophysics 531 (2011) A159.
[15] T. J. Cornwell, R. a. Perley, Radio-interferometric imaging of very large fields - The problem of non-coplanar arrays, Astronomy and Astrophysics 261 (1992) 353–364.
[16] U. Rau, et al., Advances in Calibration and Imaging Techniques in Radio Interferometry, IEEE Proceedings 97 (2009) 1472–1481.
[17] A. Scaife, SDP Memo: The SDP imaging pipeline, Tech. rep., SKA Science Data Processor Consortium (2016).
[18] T. J. Cornwell, M. A. Voronkov, B. Humphreys, Wide field imaging for the Square Kilometre Array, Proc. SPIE 8500 (Aug. 2012).
[19] A. R. Offringa, et al., WSClean: an implementation of a fast, generic wide-field imager for radio astronomy, Mon. Not. R. Astron. Soc. 444 (1) (2014) 606–619.
[20] C. Tasse, et al., Faceting for direction-dependent spectral deconvolution, Astronomy & Astrophysics 611 (2018) A87.
[21] B. Veenboer, J. W. Romein, Radio-Astronomical Imaging: FPGAs vs GPUs, in: Euro-Par 2019: Parallel Processing, Springer International Publishing, 2019, pp. 509–521.
[22] S. Jaeger, The Common Astronomy Software Application (CASA), in: R. W. Argyle, P. S. Bunclark, J. R. Lewis (Eds.), Astronomical Data Analysis Software and Systems XVII, Vol. 394 of ASP Conference Series, 2008, pp. 623–627.
[23] 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, 2012, pp. 321–330.
[24] D. Muscat, High-Performance Image Synthesis for Radio Interferometry, Master's thesis, University of Malta (2014).
[25] B. Merry, Faster GPU-based convolutional gridding via thread coarsening, Astron. Comput. 16 (2016) 140–145.
[26] A. Griffin, A. Ensor, End-to-end Modelling of the Imaging Pipeline in Radio Astronomy, in: 2018 IEEE 10th Sensor Array and Multichannel Signal Processing Workshop (SAM), Vol. 8, IEEE, 2018, pp. 480–484.
[27] A. Griffin, A. Ensor, SDP Memo: Numerical Precision, Tech. rep., SKA Science Data Processor Consortium (2018).
[28] S. Salvini, SDP Memo: On the Precision Required in SDP Pipelines, Tech. rep., SKA Science Data Processor Consortium (2018).
[29] D. Luebke, CUDA: Scalable parallel programming for high-performance scientific computing, in: 2008 5th IEEE international symposium on biomedical imaging, 2008, pp. 836–838.
[30] J. E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, IEEE Comput. Sci. Eng. 12 (1-3) (2010) 66–73.
[31] C. Lauter, A new open-source SIMD vector libm fully implemented with high-level scalar C, in: 2016 50th Asilomar Conference on Signals, Systems and Computers, IEEE, 2016, pp. 407–411.

[32] S. Oberman, M. Siu, A High-Performance Area-Efficient Multifunction Interpolator, in: 17th IEEE Symposium on Computer Arithmetic, 2005, pp. 272–279.
[33] NVIDIA Corporation, NVIDIA CUDA C programming guide (2018).
[34] M. Drobot, Low Level Optimizations for GCN (May 2014).
[35] S. Lagarde, Inverse trigonometric functions GPU optimization for AMD GCN architecture, https://seblagarde.wordpress.com (Dec. 2014).
[36] H. Bal, et al., A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term, IEEE Computer 49 (5) (2016) 54–63.
[37] Jülich Supercomputing Centre, JURON (IBM-NVIDIA pilot), https://hbp-hpc-platform.fz-juelich.de (2016).
[38] J. Treibig, G. Hager, G. Wellein, LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments, Proceedings of the International Conference on Parallel Processing (2010) 207–216.
[39] J. W. Romein, B. Veenboer, PowerSensor 2: a Fast Power Measurement Tool, 2018 IEEE International Symposium on Performance Analysis of Systems and Software (2018) 111–113.
[40] B. Mort, A simple interferometer baseline coordinate generator, GitHub: SKA-ScienceDataProcessor/uvwsim (2015).
[41] SKA Engineering Change Proposal, https://skaoffice.atlassian.net (2017).
[42] S. Williams, A. Waterman, D. Patterson, Roofline: An Insightful Visual Performance Model for Multicore Architectures, Communications of the ACM 52 (4) (2009) 65–76.
[43] Spectral-line imager for MeerKAT, GitHub: SKA-SA/katsdpimager (2020).
[44] R. Nijboer, et al., Parametric models of SDP compute requirements, Tech. rep., ASTRON Netherlands Institute for Radio Astronomy, SKA SDP PDR deliverable (2015).
[45] B. R., et al., SKA1 Level 0 Science Requirements, Tech. rep., The SKA Organisation (2015).
[46] SDP parametric model, GitLab: SKA-ScienceDataProcessor/sdp-par-model (2019).
[47] R. Braun, et al., SKA1 science priority outcomes, Tech. rep., ASTRON Netherlands Institute for Radio Astronomy, SKA design document (2014).
[48] The SKA Organisation, Designing the Square Kilometre Array, https://cdr.skatelescope.org/ (2018).
[49] A. P., et al., SDP System Module Decomposition and Dependency View, Tech. rep., SKA Science Data Processor Consortium (2015).