## Radio-Astronomical Imaging on Accelerators

Veenboer, Abraham Jacobus Petrus

2021

VRIJE UNIVERSITEIT

# Radio-Astronomical Imaging on Accelerators

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op maandag 20 september 2021 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Abraham Jacobus Petrus Veenboer

geboren te Heemskerk

| | |
|---|---|
| promotor: | prof.dr.ir. H.E. Bal |
| copromotor: | dr. J.W. Romein |

# Table of contents

## Summary

Imaging is generally considered the most compute-intensive and therefore most challenging part of the data processing pipeline of a radio telescope. To reach the high dynamic ranges imposed by the high sensitivity and large field of view of the new generation of radio telescopes, we need to be able to correct for direction-independent effects (DIEs) such as the curvature of the earth as well as for direction-dependent time-varying effects (DDEs) such as those caused by the ionosphere during imaging. Existing imaging algorithms such as W-projection correct for DIEs, and AW-projection additionally corrects for DDEs, but these algorithms have their limitations which make them computationally infeasible for future radio telescopes.

The novel Image-Domain gridding (IDG) algorithm was designed to avoid the performance bottlenecks of traditional AW-projection gridding by applying instrumental and environmental corrections in the image domain instead of in the Fourier domain [1]. We implement, optimize, and analyze the performance and energy efficiency of IDG on a variety of hardware platforms to find which platform is the best for IDG. We analyze traditional CPUs, as well as several accelerator architectures.

Over the last decade, graphics processors (GPUs) have emerged as a popular computing platform, as they offer substantially more compute power over traditional processors (CPUs) at higher energy efficiency. While GPUs are generally more difficult to program than CPUs, easy-to-use development environments such as CUDA [2] and OpenCL [3] make GPUs attractive accelerators.

Due to recent technology developments for configurable processors (FPGAs) these devices have now also entered the domain where GPUs are used. FPGAs now support the high-level OpenCL programming language, have support for floating-point operations in hardware, and tight integration with CPU cores allow them to be used as accelerators. Combined, these are game changers: they dramatically reduce development times and allow using FPGAs for applications that were previously deemed too complex.

Thorough performance analysis, in which we apply roofline analysis [4], shows that our parallelization approaches and optimizations lead to close-to-optimal performance on all these platforms. The analysis also indicates that, by leveraging dedicated hardware to evaluate trigonometric functions, GPUs are both much faster and more energy-efficient than CPUs.

Our FPGA implementation of IDG makes efficient use of DSP resources to implement arithmetic (floating-point) operations. Almost all DSP resources are used almost all cycles to perform useful operations at runtime. The achieved performance is bound by clock speed, something we cannot improve with the current development tools. FPGAs do not outperform GPUs for IDG, but still, provide significant acceleration over using traditional CPUs.

We also show that on GPUs, IDG is an order of magnitude faster and more energy-efficient than AW-projection. IDG alleviates the limitations of traditional imaging algorithms while it enables the advantages of GPU acceleration: better performance at lower power consumption. The hardware-software co-design presented in this thesis has resulted in a highly efficient imager. This makes IDG on GPUs an ideal candidate for meeting the computational and energy efficiency constraints of the Square Kilometre Array (SKA) [5].

IDG has been integrated with a widely-used astronomical imager (WSClean) and is now being used in production by a variety of different radio observatories such as LOFAR [6] and the MWA [7]. It is not only faster and more energy-efficient than its competitors, but it also produces better quality images.

Part I

# *Introduction and Background*

Chapter 1

# General Introduction

This chapter starts with an introduction of the field of radio astronomy and describes the main challenges in radio-astronomical imaging. This leads to a number of research questions, which we describe in Section 1.1. Section 1.2 provides an outline of the remainder of the thesis.

Radio astronomy is the field of astronomy where radio telescopes are used to study the universe at radio-frequency wavelengths. To this end, electromagnetic waves emitted by radio sources, up to billions of light-years away, are detected by the telescopes and processed into a map of the sky. Unlike optical telescopes, radio telescopes are not affected by the weather conditions on earth, as the low-frequency signals are not blocked by clouds, see also Figure 1.1.

In optical telescopes, the resolution is determined by the diameter of the primary mirror or lens in the instrument. This is also known as the 'aperture' of the telescope. Similarly, for a typical single-dish radio telescope the resolution is determined by the collecting area of the dish. Large telescopes are needed to create high-resolution images of the sky. The largest telescope currently in existence is FAST [8], which consists of a single dish of 500 meters in diameter. Since the resolution of a telescope scales inversely proportional with wavelength, the aperture of a radio telescope needs to be much larger than the aperture of an optical telescope to obtain the same resolution.

By using a technique called 'aperture synthesis', the signals received by an array of telescopes (receivers) can be combined to simulate a telescope with a larger collecting area than that of a single dish. This creates an 'astronomical interferometer' that is equivalent in resolution to a single telescope with a diameter equal to the maximum spacing between the receiver elements in the array. Every pair of receivers (every baseline) contributes to the aperture of this simulated telescope, but the aperture is sparse. Therefore observations typically span multiple hours so that due to earth-rotation, the orientation of baselines with respect to the observed sky changes (earth-rotation synthesis). This way the aperture gets filled, which increases sensitivity. This process of aperture synthesis and earth-rotation synthesis is illustrated in Section 2.1.

The measurements (visibilities) of multiple baselines are the input for a radio-astronomical imager that produces a 'sky image', which is a map of sources in the sky. The relation between visibilities and the sky brightness that we would like to obtain is given by a measurement equation, which in its simplest form is essentially a Fourier transformation. To efficiently perform this operation using a Fast Fourier Transformation (FFT), the non-uniform visibilities first need to be placed on a regular grid. This operation is called *gridding* and is typically one of the most time-consuming subparts of imaging.

The LOw-Frequency ARray (LOFAR) [6] radio telescope is currently the most sensitive low-frequency radio telescope and comprises tens of stations (each consisting

of many receivers) centered around The Netherlands with remote stations all over Western Europe, with a maximum baseline length of about 2000 km. The future Square-Kilometre Array (SKA) will have even more receivers and will be built as two arrays, one in South Africa (SKA1-Low) and one in Western Australia (SKA1-Mid).

While the design and construction of such instruments are challenging themselves, another challenge awaits when the instrument is put into use. A radio telescope such as the SKA produces enormous amounts of data (at a rate multiple times the global internet traffic) and it is impossible to store all this data. The measurements produced by the instrument, therefore, need to be processed in (near) real-time to create data products for use by astronomers around the world. This requires processing techniques optimized for both high performance and high energy efficiency.

Even for existing telescopes such as LOFAR, with in the order of 50,000 antennas grouped into about 50 stations, data processing remains challenging – in particular, when *direction-dependent effects* (DDEs) have to be taken into account [9]. Correcting for the DDEs improves the 'dynamic range' of the image, which is the ratio between the power of the brightest and dimmest source in the sky image. For instance, to construct the deepest extragalactic LOFAR surveys, a dynamic range of up to $\approx 1{:}10^5$ to $1{:}10^6$ is needed. The next generation of highly sensitive radio interferometers are capable of providing an even higher dynamic range ($\approx 1{:}10^6$ to $1{:}10^8$), but only when the DDEs are corrected for [10].

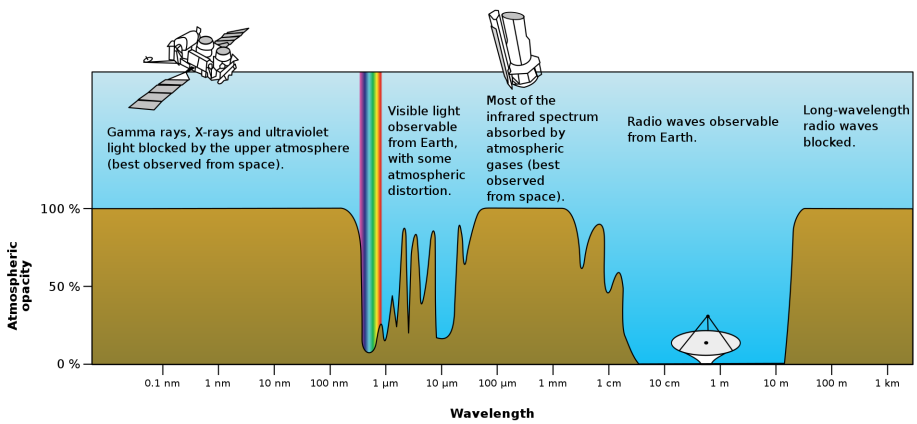



***Fig. 1.1:*** *Radio astronomy, like optical astronomy, studies the celestial objects (planets, stars, galaxies etc.) by "capturing" the light that they emit, but that, unlike optical astronomy, cannot be seen with our eyes. With their instruments (radio telescopes) radio astronomers detect radio emission from these objects.*

## 1.1 Research questions

Motivated by the challenges outlined above, the main research question of this thesis is:

> **RQ:** *What is the best, fastest, and most energy-efficient imaging algorithm for future radio telescopes?*

To answer this question, we first provide the necessary background on radio-astronomical imaging in Chapter 2, where we discuss the challenges for imaging in future radio telescopes, and we introduce state-of-the-art computing hardware (accelerators). In this thesis, we will analyze the performance and energy efficiency for radio-astronomical imaging on various accelerators. This analysis requires a way to measure power consumption. This leads to the first research question, which we will answer in Chapter 3:

> **RQ1:** *How do we compare the energy consumption of CPUs and accelerators?*

Next, we focus on the Image-Domain Gridding (IDG) algorithm. This algorithm is designed to solve the limitations of state-of-the-art imaging algorithms. An efficient implementation of this algorithm might be a good candidate for current and future radio telescopes. Therefore, in Chapter 4, we first explain how we implement this algorithm using various processing steps, and an *execution plan*. An implementation of IDG comprises of a set of compute kernels (one for each processing step) and a mechanism to execute these processing steps according to the execution plan.

We need to find out which processing steps are most suitable for acceleration and implement corresponding compute kernels on various hardware architectures:

> **RQ2:** *How do we efficiently implement Image-Domain Gridding on accelerators?*

For each architecture, we discuss the optimizations that we applied, and we analyze the performance and energy efficiency:

> **RQ3:** *What is the most efficient class of hardware architectures for IDG?*

We address CPUs in Chapter 5, to answer the following sub research question:

> **RQ3a:** *How efficient is IDG on CPUs?*

In Chapter 6, we look at Graphics Processing Units (GPUs) to answer:

> **RQ3b:** *How efficient is IDG on GPUs?*

Finally, in Chapter 7, we study Field Programmable Gate Arrays (FPGAs):

> **RQ3c:** *How efficient is IDG on FPGAs?*

Chapter 8 provides a condensed overview of the results from Chapter 5 through 7 and answers the following research question: We compare our implementation of IDG on CPUs and GPUs to W-Projection and AW-Projection in Chapter 9 to answer the following research question:

> **RQ4:** *How does IDG compare to traditional imaging techniques in terms of performance and energy efficiency?*

In Chapter 10 we use the best imaging solution (established in **RQ3**) and analyze its performance and energy efficiency in the context of the Square Kilometre Array to answer the final research question:

> **RQ5:** *Does IDG meet the performance and energy efficiency requirements for the Square Kilometre Array?*

To this end, we first establish the rate at which the SKA will produce measurements that have to be processed in (near) real-time and the processing resources and power budget available for imaging. Next, we extrapolate our results to see whether IDG meets these requirements.

## 1.2 Thesis outline

In this thesis, we propose to use an imager based on the novel Image-Domain Gridding (IDG) algorithm as a candidate solution for future radio telescopes such as the Square-Kilometre Array (SKA). The various chapters have been grouped into four parts:

1. Introduction and Background (Chapters 1 to 4)
2. Image-Domain Gridding Implementation and Analysis (Chapters 5 to 8)
3. Image-Domain Gridding in Context (Chapters 9 to 11)
4. Conclusion and Outlook (Chapter 12)

We will now provide an overview of the contents of this thesis.

### Chapter 2: Background

In this chapter we provide the necessary background on radio-astronomical imaging: we give a brief overview of various *gridding* algorithms, we discuss the challenges that the upcoming Square Kilometre Array poses and we sketch the history of computer chips, which ultimately led to the accelerators that we will evaluate in this thesis. We end this chapter with an explanation of *roofline analysis* [4]. We use this model throughout the thesis to help understand the achieved performance, and to identify bottlenecks.

### Chapter 3: Energy Efficiency Analysis

To assess the energy efficiency of various devices, we need tools to measure energy consumption. Several (software) tools exist to measure energy consumption using built-in power meters, but these vary in accuracy and time resolution. Therefore we created PowerSensor, an easy-to-use tool that performs energy consumption measurements at millisecond resolution. This chapter is based on:

**PowerSensor 2: A Fast Power Measurement Tool**
Romein, J. W., **Veenboer, B.**
In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 111-113, IEEE, 2018*

John W. Romein came up with the idea to build a custom tool, which is based on an Arduino microcontroller, Hall-effect current sensors, and a software library, while the author of this thesis helped with the realization and with the writing of the paper.



## Chapter 4: Image-Domain Gridding

Radio-astronomical imaging comprises a few steps, of which *gridding* is typically the most compute-intensive. During gridding, input samples (visibilities) are placed onto a regular grid by applying an operation that resembles a convolution to every sample. Depending on the imaging use case, this convolution kernel could be large, be different for different subsets of visibilities, and may need to be computed frequently to take direction-dependent effects (DDEs) into account.

S. van der Tol realized that there is an alternative: by moving the gridding operation to the image domain, the convolution kernels are not needed. In this setting, called 'Image-Domain Gridding' (IDG), gridding becomes the complex multiplication of visibilities with a correction term. While the number of operations performed per visibility for IDG is higher than that for traditional (Fourier-domain) gridding, correcting for DDEs in the image domain is computationally cheap.

Image-Domain Gridding started as a rough proof-of-concept CUDA code. The author derived a reference implementation to study its properties and next created the first highly optimized imaging routines based on this algorithm for CPUs as well as for GPUs. This work is published in:

**Image-Domain Gridding on Graphics Processors**
**Veenboer, B.**, Petschow, M., Romein, J. W.
in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), pages 545–554, IEEE, 2017*

In that paper, the concept of an 'execution plan' is introduced to orchestrate the execution of the IDG algorithm. This was an essential step in the process of getting from a proof-of-concept gridding kernel to an imager capable of producing sky images from real LOFAR data. Furthermore, this IDG implementation has become the foundation for the IDG library that is used in production today. In this paper, we also present a thorough performance analysis in which we apply a roofline analysis to find performance bounds.

In the meantime, S. van der Tol completed a formal derivation of the IDG algorithm and analyzed the accuracy compared to traditional gridding. The aforementioned implementations of IDG for CPUs and GPUs were indispensable for the performance

experiments and imaging examples that were part of this study. This study, which is not a part of this thesis, is published in:

**Image Domain Gridding**
van der Tol, S., **Veenboer, B.**, Offringa, A. R.,
In *Astronomy & Astrophysics (A&A), article A28, EDP Sciences, 2018*

## Chapter 5 and 6: IDG on CPUs and GPUs

In these two chapters we take a deep dive into our Image-Domain Gridding implementations for CPUs, and for GPUs, respectively. The contents of these chapters are based on an extended subset of [11] and are published in [12]:

**Radio-Astronomical Imaging on Graphics Processors**
**Veenboer, B.** Romein, J. W.
In *Astronomy & Computing, Elsevier, Volume 32, July 2020*

In this paper we introduce new features that make IDG a versatile imager for a variety of real-world imaging use cases. We also optimized IDG for several additional devices, including new generations of GPUs from AMD and NVIDIA, as well as the Intel Xeon Phi manycore processor.

## Chapter 7: IDG on FPGAs

Three major FPGA technology developments, hardware Floating-Point units, tight integration with CPUs, and support for the OpenCL programming language, make FPGAs an interesting platform for HPC applications. Combined, these are game changers: they dramatically reduce development times and allow using FPGAs for applications that were previously deemed too complex. We implemented IDG for FPGA and compared programming models, optimizations, performance, and energy efficiency in the following paper:

**Radio-Astronomical Imaging: FPGAs vs GPUs**
**Veenboer, B.**, Romein, J. W.
In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), pages 509-521, Springer, 2019* (**best paper award**)

## Chapter 8: CPUs versus GPUs versus FPGAs

In this chapter we do not present any new research, but we provide an overview of the performance bounds identified in Chapters 5 to 7. Next, we show normalized

performance and energy efficiency results to compare the CPU, GPU, and FPGA architectures in relative terms to find the most suitable architecture for IDG.



## Chapter 9: IDG versus AWPG

After having analyzed the performance and energy efficiency of IDG in quite some detail, we compare the performance and energy efficiency of IDG with the previous state-of-the-art imaging technique *AW-Projection* in Chapter 9. To this end, we explain how we implemented AW-Projection gridding (AWPG) as an extended version of the highly optimized W-projection gridding (WPG) implementation presented in [13], and we conduct various experiments to compare performance and energy efficiency. The contents of this chapter are also part of [12].

## Chapter 10: IDG for the Square-Kilometre Array

In Section 2.2 we explain the challenges that the Square Kilometre Array (SKA) poses with respect to the pipeline that produces high-resolution images. After having demonstrated how we tackle this challenge by using IDG on accelerators, we assess whether our best candidate solution is sufficiently fast and energy-efficient to serve as a SKA imager. The contents of this chapter is also published in [12].

## Chapter 11: IDG use cases

The research outlined above has resulted in more than just some publications, it has also led to a publicly available software package [14] which now finds widespread use throughout the radio-astronomical community. A high-profile science case, research into the Epoch-of-Reonization (EoR), benefits greatly from IDG, as shown in:

**Precision requirements for interferometric gridding in the analysis of a 21 cm power spectrum**
Offringa, André R., Mertens F., van der Tol, S., **Veenboer, B.**, Gehlot, B. K., Koopmans L. V. E., Mevius M.,
In *Astronomy & Astrophysics (A&A), article A12, EDP Sciences, 2019*

IDG can also be used for radio-astronomical calibration, as presented in:

**Estimating continuous direction-dependent gain screens from radio interferometric visibilities and a large skymodel**
van der Tol, S., **Veenboer B.**, Offringa, A. R., Rafferty, D.,Mevius M., Dijkema T. J.,
In *Astronomical Data Analysis Software and Systems (ADASS), 2019*

We have summarized our experiences with Image-Domain Gridding in practice in the following overview paper, which also formed the basis for Chapter 11:

**Radio-Astronomical Imaging with WSClean and Image-Domain Gridding**
**Veenboer, B.**, van der Tol, S., Offringa, A. R., Romein, J. W., Dijkema, T. J.,
In *General Assembly and Scientific Symposium (GASS) of the International Union of Radio Science (URSI GASS), 2020*

The work on high-resolution imaging [12] and on the use of FPGAs for IDG [15] has led to a collaboration where a near-memory accelerator architecture is used for the acceleration of high-resolution imaging. Although it is not a part of this thesis, it is published in:

**Near Memory Acceleration on High Resolution Radio Astronomy Imaging**
Corda, S., **Veenboer, B.**, Awan, A. J., Kumar, A., Jordans, R., Corporaal, H.,
In *Mediterranean Conference on Embedded Computing (MECO), IEEE, 2020*

This line of research is ongoing with an investigation on the use of (dynamic) reduced-precision computing (using FPGA hardware) for radio-astronomical imaging.

## Part IV: Closing Words

In the last part of this thesis we answer the main research question and we provide an outlook for future work.

Chapter 2

# Background

In this chapter we first explain in Section 2.1 the basic concepts of radio-astronomical imaging. Section 2.2 introduces the challenges for data processing in the Square Kilometre Array. In Section 2.3 we explain why we need accelerators to solve this challenge, Section 2.5 discusses related work, and in Section 2.6 we introduce various performance and efficiency metrics used throughout this thesis.

## 2.1 Radio-astronomical imaging

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. Radio telescopes such as LOFAR (see Fig 2.8) and SKA1-Low (see Fig 2.9) are comprised of arrays of (small) antennas, while other radio telescopes (such as the VLA, MeerKAT, and SKA1-Mid) are based on an array of dishes. A notable exception is the single-dish Five-hundred-meter Aperture Spherical radio Telescope (FAST). Antenna arrays are used for frequencies around 100 MHz, while dishes are used for frequencies of 1 GHz or higher. In the remainder of this thesis, we only consider array-based radio telescopes.

By using 'aperture synthesis', the signals received by the individual elements of the array are combined to create a sky image. The first step in this process is to correlate the signals of every pair of receivers (*baselines*) in the observation. The resulting *visibilities* for all baselines are combined in an imaging step that produces the sky image. A single visibility shows up in the image as a waveform, this is illustrated in Fig. 2.1. This example uses simulated data with one source in the center of the image. This source only becomes visible when multiple visibilities (from multiple baselines) are added to the image, see Fig. 2.2 and Fig. 2.3. By adding longer baselines (i.e. by imaging visibilities originating from stations spaced further apart), the resolution of the image increases as illustrated in Fig. 2.4. Finally, by observing over an extended period of time, coverage of the visibilities in *uv-space* improves, resulting in a better image (see Fig. 2.5). These images are courtesy of NRAO [16].

As shown in Fig. 2.6, the creation of a sky image requires roughly three steps: (1) the digitized signals from pairs of distinct receivers are correlated to produce the *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. 2.10.

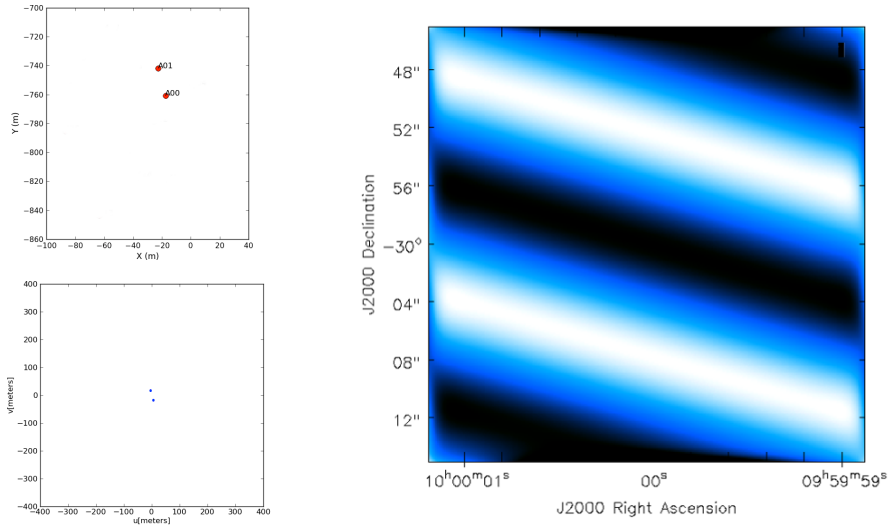



***Fig. 2.1:*** *The top left plot shows the position of two receivers (A00 and A01). These receivers form a baseline (and its mirror baseline) with corresponding 'uv-coordinates' – the bottom left plot. After imaging, a measurement (a 'visibility') of this baseline shows up as a waveform in the corresponding image shown – the plot on the right.*

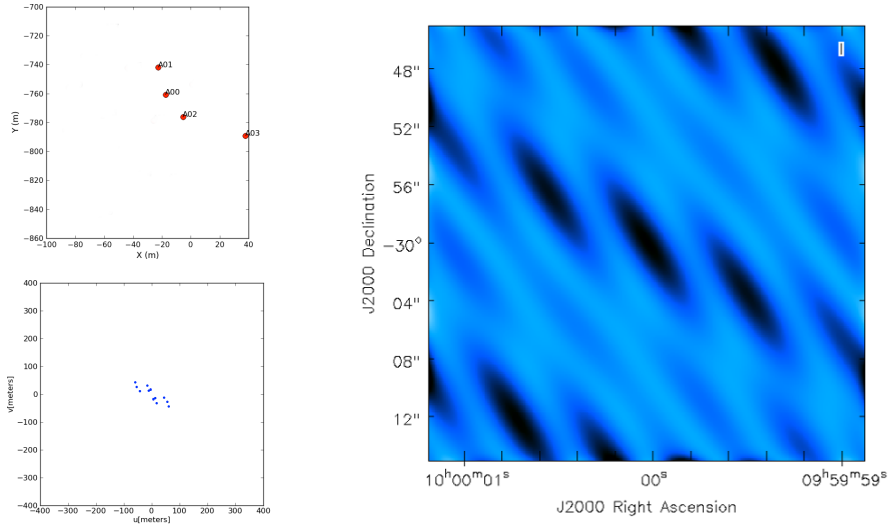



***Fig. 2.2:*** *Every baseline (pair of two receivers) adds a waveform to the image.*

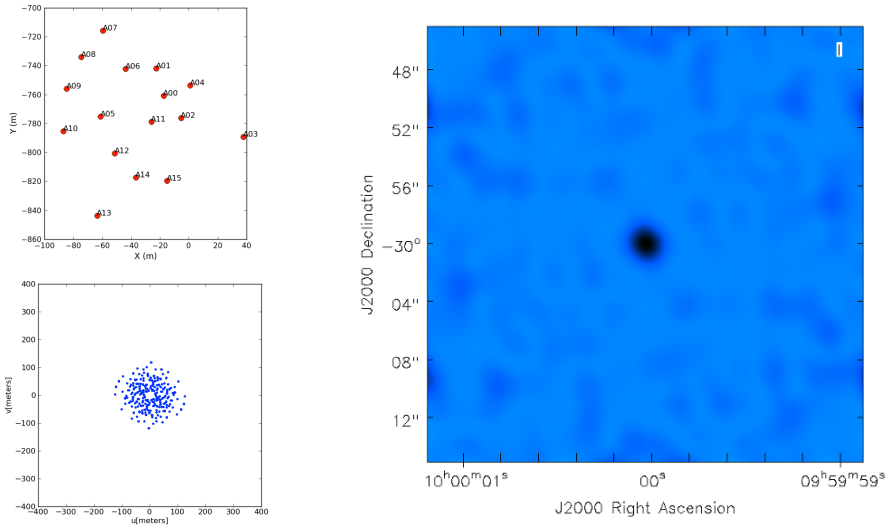



***Fig. 2.3:*** *Using 16 receivers, the source in the center of the image becomes visible.*

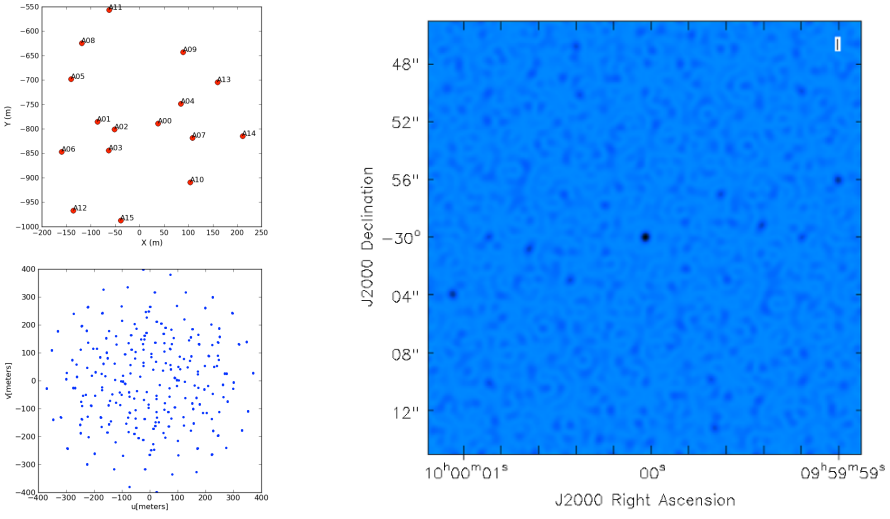



***Fig. 2.4:*** *With longer baselines (larger receiver spacings), the resolution of the image increases.*

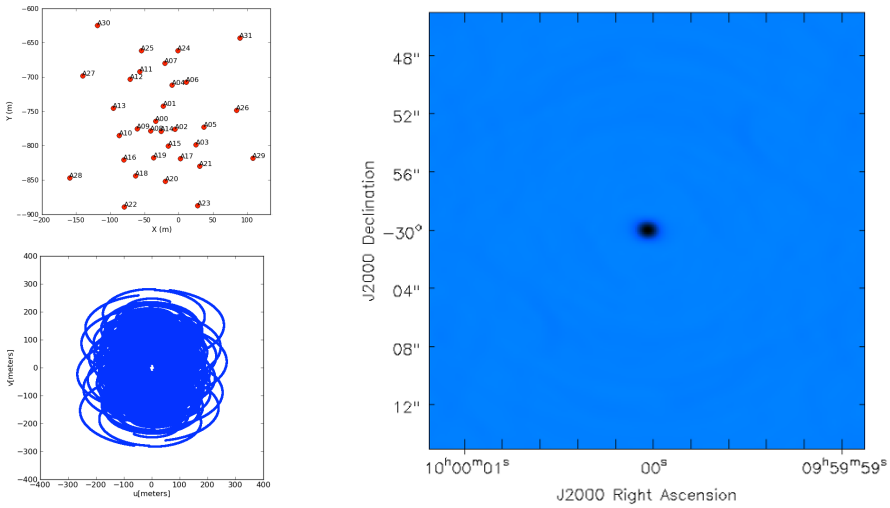



***Fig. 2.5:*** *For an observation over an extended period of time, the orientation of the baselines with respect to the sky changes. This leads to the tracks in uv-space as shown in the bottom left image and is known as 'earth-rotation synthesis'. Having a better coverage of the uv-space (more dense sampling), typically results in a better image.*

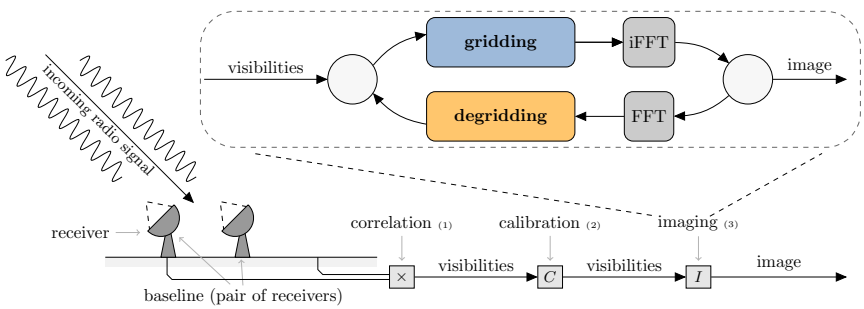



***Fig. 2.6:*** *Incoming radio signals are received by a pair of receivers. The correlator combines the signals into measurements that we call* visibilities. *The visibilities contain information on the amplitude and phase of the radio source. After calibration of the visibilities, an imaging step is used to create an image of the sky.*

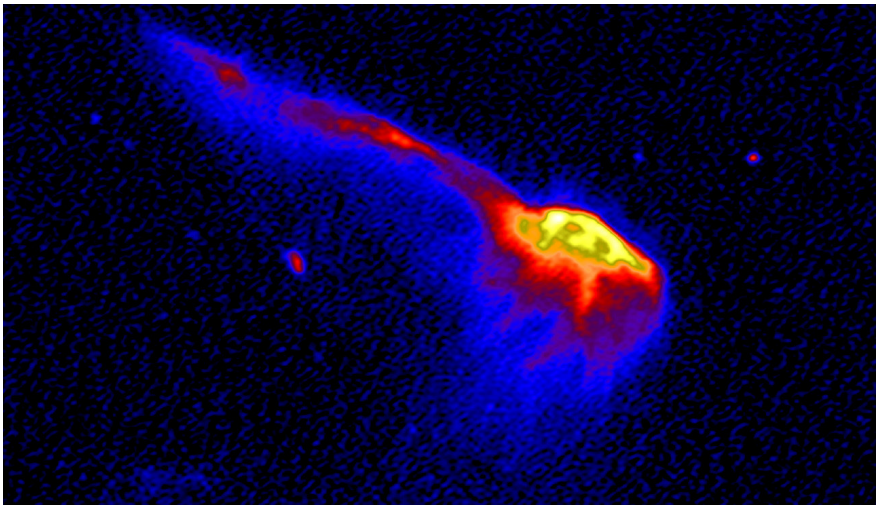


***Fig. 2.7:*** *This image of the 'Toothbrush' cluster of radio sources is created using ten hours of observational LOFAR data with High-Band Antenna (HBA) receivers at 120-181 Mhz [9].*

An example of a sky-image is shown in Fig. 2.7. Advanced calibration and processing techniques are needed to create *deep images*. These are images with high signal-to-noise such that faint radio sources are visible [9].

The (dipole) receivers in a typical radio telescope measure the interference pattern of a radio signal for two orthogonal polarizations $X$ and $Y$. Multiplying and integrating (correlating) the signals of a receiver pair $(q, r)$ for a short period of time (in the order of seconds) produces a single measurement. We will use the term *visibility* to denote all four combinations $XX$, $XY$, $YX$ and $YY$ of a receiver pair, hence $V^{(q,r)} \in \mathbb{C}^{2\times 2}$. All relevant parameters for imaging are summarized in Table 2.1.

The relation between visibilities and sky brightness, $B(l, m) \in \mathbb{R}^{2\times 2}$, is given by the following measurement equation [18]:

$$V^{pq} = \int_{\ell}\int_{m} A_p(\ell, m) B(\ell, m) A_q^H(\ell, m) \frac{1}{n} e^{-2\pi i (u^{pq}\ell + v^{pq} m + w^{pq}(n-1))}\, d\ell\, dm, \tag{2.1}$$

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 [19].

***Fig. 2.8:*** *Aerial photograph of the 'Superterp', the heart of the LOFAR core. The large circular island encompasses the six core stations that make up the Superterp. Two additional LOFAR stations (with a 300m diameter) are visible in the corners of the image. Each of these core stations includes a field of 96 low-band antennas and two sub-stations of 24 high-band antenna tiles each [6].*

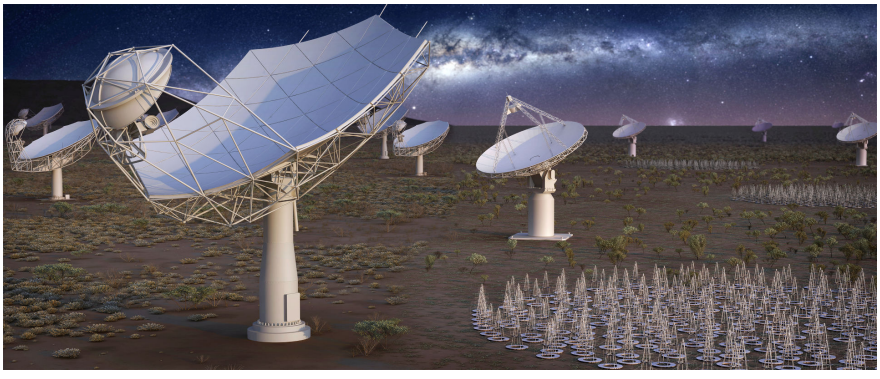


***Fig. 2.9:*** *Artist's impression of the Square Kilometre Array (SKA), which will consist of two instruments: SKA1-Low in Western Australia (the antennas) and SKA1-Mid in Southern Africa (the dishes) [5].*

| 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 | $V \in \mathbb{C}^{2\times 2}$ | $T_{obs} \times C_{obs}$ per baseline |
| integration time | $t_{int} \in \mathbb{R}$ | $t_{int} = \frac{t_{obs}}{T_{obs}-1}$ |

***Table 2.1:*** *Imaging parameters.*

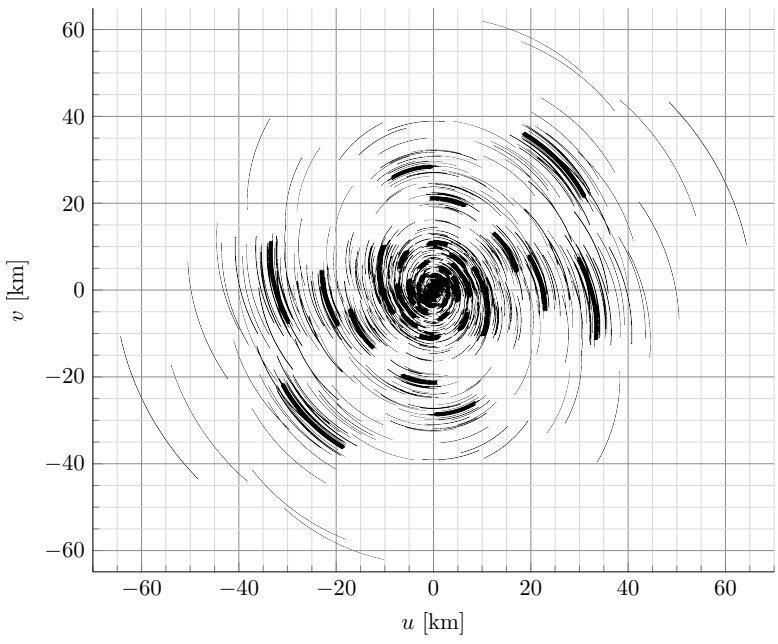



***Fig. 2.10:*** *$(u, v, w)$-coverage generated using proposed SKA1-Low receiver coordinates [17]. Every baseline (pair of receivers) contributes one track in this plane. The resulting $(u, v)$-plane is filled with both 'short baselines' and 'long baselines' and therefore representative for a wide range of imaging use cases.*

It follows from this equation that the sky brightness $B(\ell, m)$ can be reconstructed from the measurement of visibilities from different baselines ($\forall_p \forall_q \; V^{pq}$) with distinct $(u, v, w)$-coordinates. A sky image is created by reconstructing the sky brightness for a region of the sky in the observation direction.

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 [20]. A sky image can be constructed by performing a 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 [21]. In this operation a convolution kernel is applied to each of the visibilities, see the top-right panel of Fig. 4.1. In W-projection, gridding takes place in the frequency domain. After gridding the visibilities, an inverse FFT is applied to the grid to obtain the image.

Astronomical observations are affected by variable gain effects that are broadly classified as direction-independent effects (DIEs) and direction-dependent effects (DDEs) [19]. 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 [19]. 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. 2.6), imaging (gridding and inverse FFT) and prediction of visibilities (FFT and degridding) are typically repeated a couple of times to construct a sky image using a deconvolution algorithm such as the CLEAN algorithm [22]. One such iteration of imaging, deconvolution, and prediction is called a major cycle or an *imaging cycle*. The CLEAN algorithm forms the basis for most

deconvolution algorithms in radio interferometry. It comprises many iterations (minor cycles) in which peaks in the image are detected and added to a model image.

Many different deconvolution approaches have been developed, such as multi-scale multi-frequency deconvolution [23], deconvolution based on compressive sensing/sampling [24, 25], sparse recovery (MORESANE) [26], or Bayesian statistical inference (RESOLVE) [27]. These algorithms all try to bring new deconvolution features to allow for reconstruction of both bright and diffuse radio sources for highly sensitive radio telescopes such as LOFAR, the Australian Square Kilometre Array Pathfinder (ASKAP), and the Karoo Array Telescope (MeerKAT) [26].



With the advent of increasingly more advanced deconvolution algorithms, the number of (faint) sources that can be reconstructed typically increases. Consequently, more imaging cycles might be required to create the sky image.

Offringa et al. [28] compare the computational performance of various CLEAN algorithms to their improved version of multi-scale CLEAN. They show that the time spent in the minor cycles is improved, but that the required time per cycle approximately scales with the number of pixels in the image. Since cleaning is faster than inversion and prediction in most scenarios, it is less pressing to speed up cleaning further compared to improving the performance of the inversion and prediction tasks [28].

### 2.1.1 W-projection gridding

Currently, the most widely used gridding approach is known as *W-projection* [21]. 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 antennas are spaced far apart from each other, the support of the W-terms can become large, making this technique inefficient and memory-intensive [29]. Figure 2.11 through 2.14 illustrate how the W-Projection gridding algorithm works. One approach to reduce the support of the W-terms is to split the image into facets [20]. Furthermore, W-projection gridding can be extended by *W-stacking* [30, 31] or *W-snapshots* [30] to limit the support size of W-terms, at the cost of having to sort the visibilities. W-stacking is illustrated in Fig. 2.15.

### 2.1.2 AW-projection gridding

The computational challenge increases further when the correction for DDEs is taken into account [29]. The correction of these A-terms can be done similarly to the

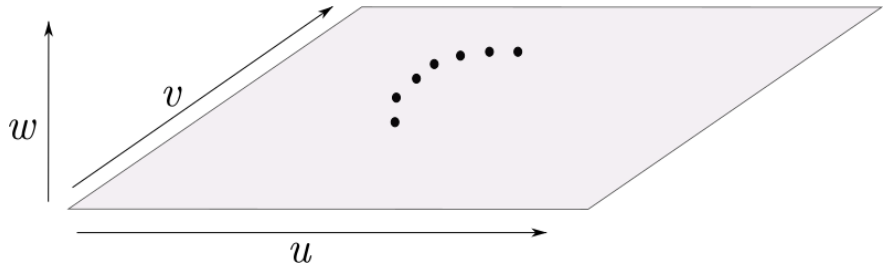



***Fig. 2.11:*** *A schematic representation of visibilities (indicated with black dots) and an $u, v$-plane (light gray rectangle).*

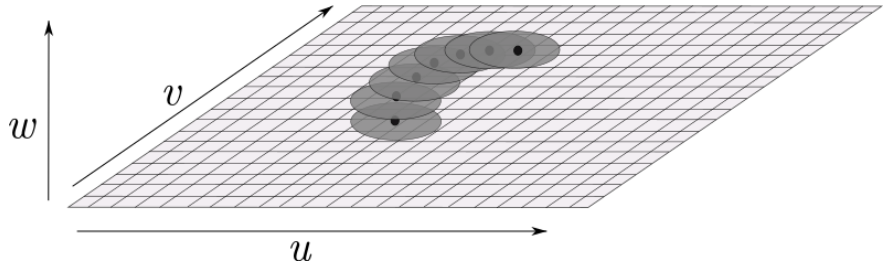



***Fig. 2.12:*** *The visibilities are placed onto a regular grid by applying a convolution kernel (dark gray circles) to each visibility.*

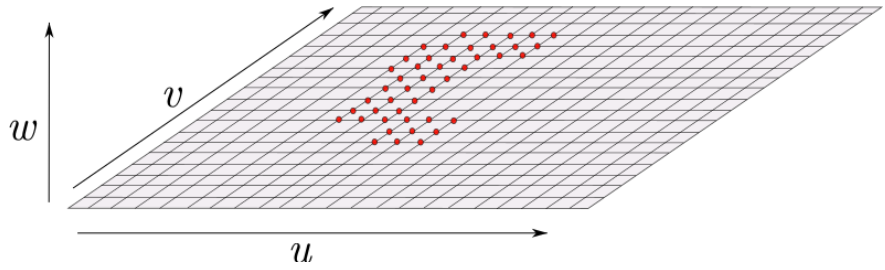



***Fig. 2.13:*** *The result of the convolution (product of visibility and convolution kernel term) is added to the grid. The updated pixels are indicated in red. The number of pixels that are updated (the* support *of the convolution function) depends on the imaging parameters.*

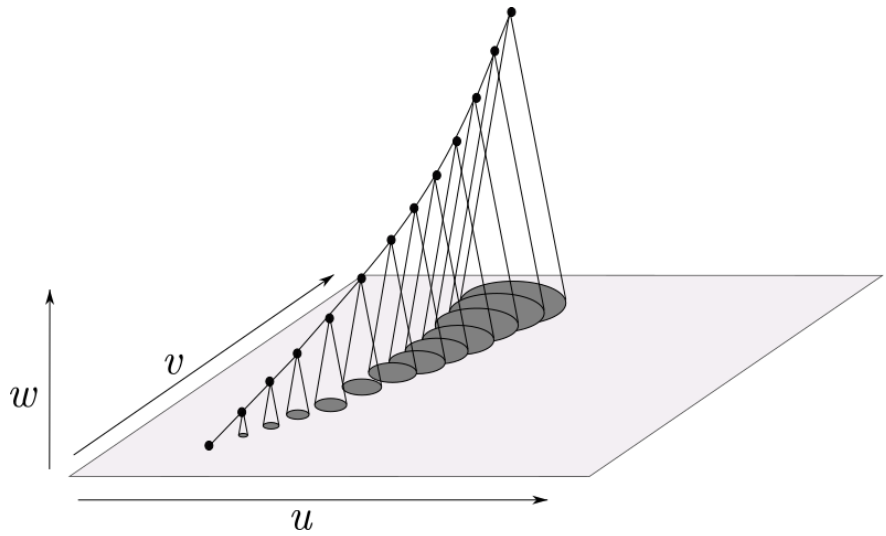



***Fig. 2.14:*** *The required size of the convolution kernel depends on the $(u, v, w)$-coordinate associated with each visibility. In this example, a baseline draws a track with rapidly changes $w$-coordinate (it gets bigger over time) and thus requires increasingly larger convolution kernels.*

W-term correction – called *A-projection*, see also Fig. 2.16. Applying both corrections results in *AW-projection* gridding [19]. 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 piece wise constant A-term is applied to each facet [32]. This method has the disadvantage of taking DDEs into account in a discontinuous manner in the image domain.

### 2.1.3 Image-domain gridding

A radio-astronomical imaging application needs to be able to correct for both W-terms and A-terms to be able to attain the high-quality images required to make discoveries in radio astronomy happen. The high cost of computing the convolution kernels in W-projection and AW-projection was the main motivation for the development of the *Image-Domain Gridding* algorithm (IDG) [1]. IDG effectively performs the same operation as classical gridding and degridding with AW-projection, except that it is designed to circumvent the computation of convolution kernels altogether.

The IDG algorithm corrects for both the W-terms and the A-terms in the image domain rather than in the Fourier domain and addresses the limitations of traditional

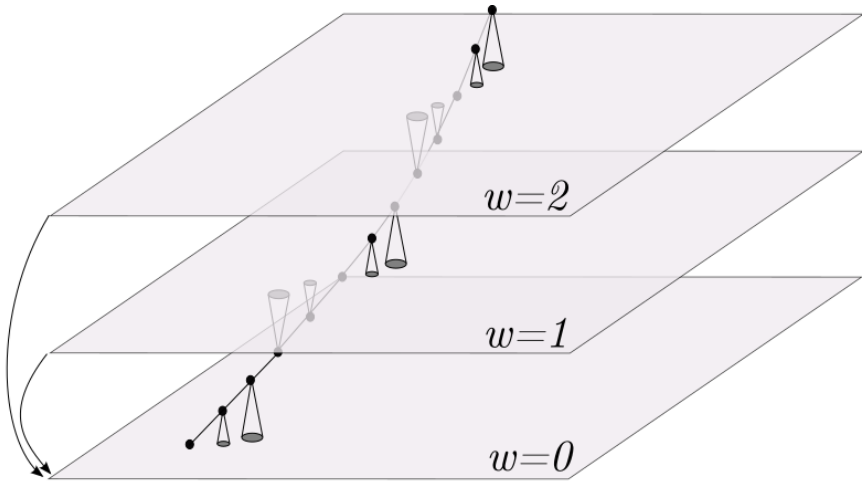



***Fig. 2.15:*** *With W-stacking, visibilities are grouped based on their w-coordinate and assigned to one of the potentially many* w-layers. *This example shows three different w-layers. Gridding proceeds as with W-projection, albeit that visibilities are gridded on the w-layer that they are assigned to. After gridding, the w-layers are combined into a single grid.*

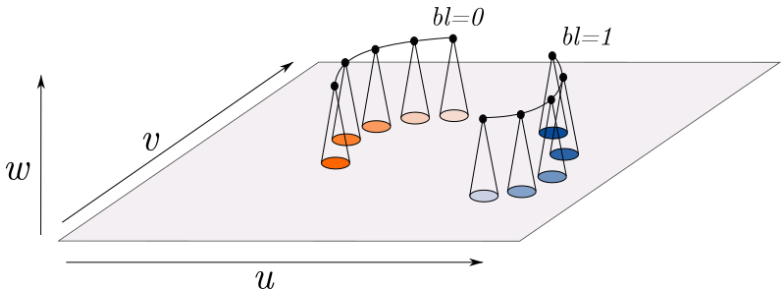



***Fig. 2.16:*** *Direction-dependent effects need to be corrected for different for distinct baselines, as each baseline 'sees' the sky in a (slightly) different way. This is illustrated with two baselines, each with its convolution kernels (one in orange, and one in blue). In this example the colors fade over time, illustrating that the convolution kernels can also change over time, for instance, to compensate for changes in the ionosphere during the observation.*

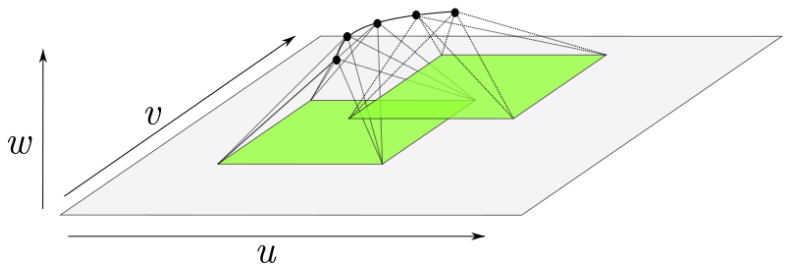



***Fig. 2.17:*** *In IDG, visibilities are mapped to subgrids (drawn in green), which are low-resolution versions of the grid. Instead of applying a convolution to each of the visibilities, every pixel in the subgrids is computed as a sum of visibilities multiplied by a correction term. We explain the IDG algorithm in more detail in Chapter 4.*

methods currently in use – in particular, the challenging DDEs. In this thesis, we study the (energy) efficiency of accelerated imaging using IDG on a variety of hardware platforms.

At the center of IDG are 'subgrids', onto which visibilities are gridded, rather than gridding them onto the grid. This is illustrated in Fig. 2.17.

### 2.1.4 Imaging applications

Most state-of-the-art imagers make use of one or more of the various gridding algorithms and their implementations: e.g., CASA [33] and LOFAR's AWImager [10] use a variant of W-projection and AW-projection, while WSClean [31] 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.

Another imaging method combines DDE correction with faceting by applying a piecewise constant A-term to each facet [32]. This method has the disadvantage of taking DDEs into account in a discontinuous manner in the image domain. IDG does not have this limitation as it effectively applies the DDEs in an interpolated manner.

## 2.2 The SKA challenge

The SKA will require a large amount of computational power at high energy efficiency [34]. To meet these requirements, we need better energy efficiency and compute capabilities than current technology provides: as discussed in [35], simply waiting for

the next generation of hardware alone will not be sufficient. A co-design between computing hardware and algorithms is needed to meet these demands.

The SKA community uses a parametric model [36] to analyze processing requirements for the Science Data Processor (SDP) compute platform. This model is available online at [36]. 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 [37] for why this is needed) and take into account that SKA1 Low might only be doing imaging observations half the time [38], the required processing rate becomes 6.3 GVisibilities/s.



We roughly distinguish two imaging use cases: *continuum imaging* and *spectral line imaging*, which differ mainly in the way that visibilities corresponding to different frequency channels are combined into a single or multiple sky-images. Furthermore, depending on the science case, sky images could either have a large field of view, a high spatial resolution, or both [39]. The latter case requires up to 100,000 $\times$ 100,000 pixel images.

Thus in order to meet SKA imaging requirements, a candidate imaging solution should (1) be able to process visibilities at a sufficiently high rate; (2) reach this data rate for continuum imaging as well as for spectral-line imaging; (3) be able to create very large sky-images; (4) remain within the power limits imposed for a computing platform for the SKA.

## 2.3 Accelerators

Gordon Moore observed that the number of components per chip roughly doubled every two years [40]. Moore divided the advances into three technical drivers: smaller transistors (20% per year), an increase in the number of transistors per area (25% per year) and, improved device and circuit designs (33% per year). The growth in chip area by about 20% (per year) through the 1970s slowed to about 10% (per year) in the 1990s and has now leveled off. Due to voltage scaling, transistors have become faster over the years (due to higher clock frequencies) while they consume less power. However, this form of scaling has also slowed over the years, transistors can be either fast or consume a low amount of power, but not both [41].

Thus while chips are not getting much bigger and transistors are not getting that much faster (without excessive power consumption), the remaining factor to increase performance is an improved design. Over time, the performance of CPUs has improved by increasing clock frequency, increasing the number of CPU cores,

and by architectural improvements such as support for vector units. The maximum clock frequency has hit a plateau at about 4 GHz. Higher clock frequencies (up to to about 5 GHz) are only attained when one or a few CPU cores are utilized and can be maintained only for a short period of time due to thermal or power constraints. Modern CPUs are highly complex chips and highly optimized multi-threaded and vectorized software is needed to fully utilize all compute capabilities.

In Fig. 2.18 we show the energy efficiency of major computer chip architectures of the last decade in terms of single-precision GFlop/W. Intel CPUs (shown in blue) has become increasingly more energy-efficient over the years, but energy efficiency does not scale according to Moore's Law. Intel has attempted to increase the energy efficiency of their CPUs by introducing the Xeon Phi series of many-core processors (shown in orange). The first commercially available processor in this series was Knights Corner, which is a PCIe *accelerator* card that runs its own operating system and is programmed much like a regular CPU.

In the meantime, manufacturers of graphics processing units (GPUs) started to support high-level programming languages to use GPUs for tasks other than gaming. The first GPU manufactured by NVIDIA to support the Compute Unified Device Architecture (CUDA) programming language was based on the Tesla architecture. As shown in Fig. 2.18 in green, this was the start of a series of computer chips with superior energy efficiency compared to CPUs. In 2016, Intel released the next Xeon Phi accelerator, Knights Landing. Like its predecessor, this chip has more cores than typical CPUs, but these cores operate on a lower clock frequency. While Knights Landing is significantly more energy-efficient compared to regular CPUs, it cannot keep up with the superior energy efficiency of contemporary GPUs and Intel announced in 2017 that its upcoming successor (Knights Hill) would be canceled.

In the same year, Intel acquired Altera. Altera is a well-established manufacturer of Field-Programmable Gate Arrays (FGPAs). The recent Arria and Stratix FPGAs perform floating-point operations natively in hardware (rather than through the use of logic) and they can be programmed by using the high-level OpenCL programming language. As Fig. 2.18 illustrates, the FPGAs (in brown) approach the energy efficiency of contemporary GPUs and therefore are an interesting accelerator platform.

Looking at the energy efficiency trends from Fig. 2.18 (the dashed lines), we do not expect that upcoming CPUs will surpass the energy efficiency of GPUs or FPGAs. GPUs and FPGAs have emerged as very interesting accelerators, and we expect that we will need to continue to investigate how such accelerators can be used to achieve both high performance and high energy efficiency.

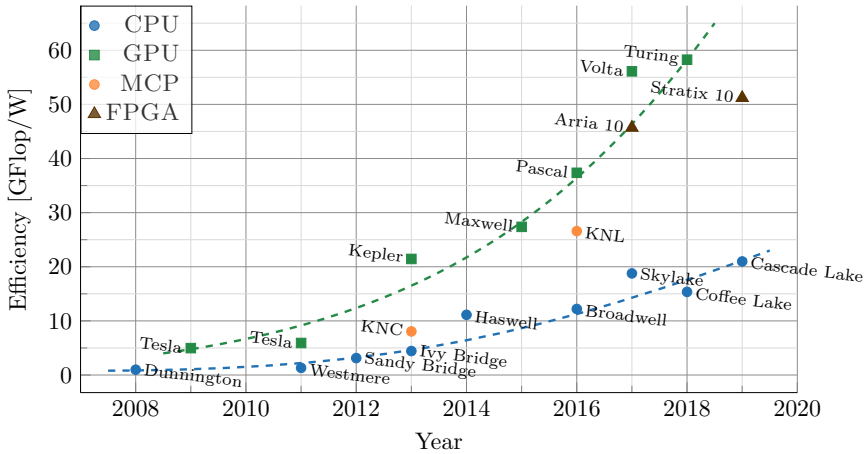



***Fig. 2.18:*** *Energy efficiency for various CPUs, GPUs, manycore processors (MCP), and FPGAs manufactured in the past decade. The energy efficiency is computed by taking the peak single-precision floating-point performance as advertised by the vendor, divided by thermal design power (TDP). For FPGAs we compute the peak floating-point performance by multiplying the number of DSPs (which support floating-point operations) by the peak operating frequency.*

## 2.4 Performance analysis

Runtime is a very common performance metric and is typically used to compare distinct candidate solutions in absolute terms. For radio-astronomical imaging, we are especially interested in how much data can be processed in a given time to assess whether (pseudo) real-time constraints can be met, rather than how long it took to process a dataset of a given size. We, therefore, use throughput measured in the number of visibilities per second as a primary performance metric.

While higher throughput is better, we also consider the theoretical peak performance of the devices in comparison. One would expect that two devices with similar peak performance should theoretically also achieve similar throughput. In practice, however, this is not always the case as we will show in this thesis. Therefore, we use the *roofline model* [4] to assess the achieved performance and identify bottlenecks and optimization opportunities. This model provides insights on the achieved performance of an application with respect to the architectural bounds of a given device, such as memory bandwidth and floating-point performance.

The roofline model relates the following quantities: the peak floating-point performance of the device ($\text{perf}_{peak}$), the peak memory bandwidth of the device ($\text{bw}_{peak}$), and the *operational intensity* ($I$) of the application. The operational intensity is defined as the number of floating-point operations performed per byte loaded or stored from or to memory.

The roofline model uses these quantities to define an upper bound (a roof) on performance:

$$\text{roof} = \min\left(\text{perf}_{peak}, I \times \text{bw}\right)$$

which gives the theoretical maximum floating-point performance for a given operational intensity, with:

$$\text{perf}_{peak} = \text{n}_{\text{fpu}} \times \text{n}_{\text{opc}} \times \text{clock}$$

where clock is the clock speed, $\text{n}_{\text{fpu}}$ is the number of floating-point units (FPUs) on the device, and $\text{n}_{\text{opc}}$ the number of floating-point operations a FPU can issue every cycle.

Applications are plotted as points on a roofline plot, with the x-value corresponding to the operational intensity and the y-value corresponding to the measured performance. Below the slanted part of the graph (where roof $= I \times \text{bw}$) the application is *memory-bandwidth bound*, whereas below the horizontal part of the graph (where, roof $= \text{perf}_{peak}$) the application is *compute bound*.

We show an example roofline plot for an NVIDIA Titan X Pascal GPU in Fig. 2.19. The gray dots correspond to synthetic benchmark kernels where threads issue memory load instructions and floating-point arithmetic in a predefined mix to achieve a certain operational intensity. As expected, the measured performance for these kernels is very close to the theoretical limit that is indicated with the green roofline.

The green dots are examples to illustrate kernels that fall either in the memory-bound region of the plot or in the compute-bound region of the plot. In both cases (additional) optimization might help to improve performance. However, without increasing operational intensity, the $example_A$ kernel will remain bound by the bandwidth of the memory.

## 2.5 Related work

Reed et al [42] describe that ever-more-powerful scientific instruments continually advance knowledge. This led to the 2013 discovery of the Higgs boson, powerful astronomy instruments such as the Hubble Space Telescope, and high-throughput

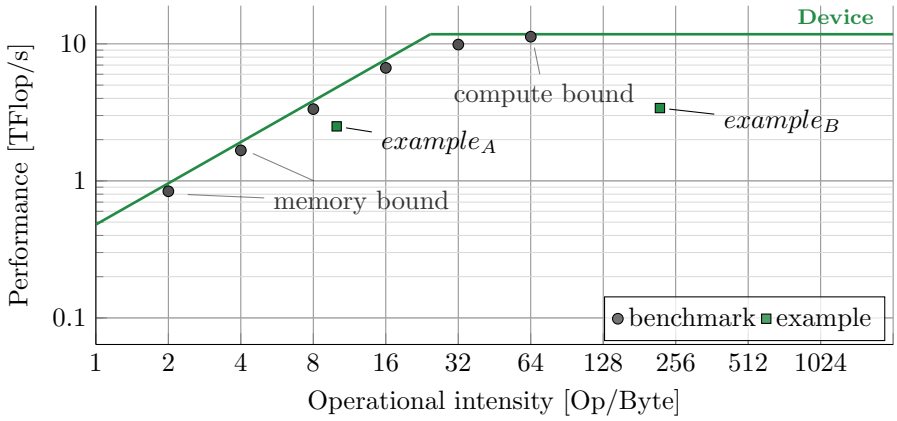



***Fig. 2.19:*** *Example roofline graph.*

DNA sequencing. Each such scientific instrument is critically dependent on computing for sensor control, data processing, international collaboration, or access. Successive generations of large-scale scientific instruments bring new capabilities, along with technical design challenges of advanced computing systems. Broadly speaking, the capability to generate data tends to grow more rapidly than the compute capabilities. Scientific research thus increasingly depends on both high-speed computing and data analytics.

Technology is changing rapidly. In the 1980s, supercomputing was dominated by Cray supercomputers. A decade later, massively parallel processors (MPP) and shared memory multiprocessors (SMP) were on the rise. In turn, clusters of commodity x86 hardware (built by Intel and AMD), as well as custom built processors (such as IBMs BlueGene) dominated the field of supercomputing [42]. Nowadays, these clusters are augmented with accelerators in the form of coprocessors, graphics processing units, and field-programmable gate-arrays (FPGAs).

Scientific research increasingly depends on high-speed computing and data analytics and interoperability and scaling of both are crucial to the future [42]. This directly links to the work presented in this thesis, as we investigate how we can use a novel imaging approach combined with the latest high-speed computing technologies to improve the output of the radio-astronomical instruments.

While computers are getting faster, their energy consumption also tends to increase. A study shows that computation per kilowatt-hour doubled every 1.57 years [43]. While this trend is encouraging, we still need to optimize for energy

efficiency to meet stringent energy efficiency goals. DARPA, for instance, targeted to build a 20 MW exaflop system which required a 56.8× performance improvement over the predecessor, while the power budget was only increased 2.4× [43].

The Green 500 list provides a ranking of the most energy-efficient supercomputer in the world [44]. This list shows that the energy efficiency of supercomputers tends to increase over the years and that the energy efficiency of accelerator-based, and custom-built systems dominate CPU-based systems [43]. With this in mind, we will be studying the energy efficiency of radio-astronomical imaging on accelerators and compare it to CPU-based reference systems.



Implementing a high-performance scientific application for emerging complex parallel computing systems is no trivial task. There is a large body of work on related topics, such as porting existing applications to these systems, directive-based programming, and performance portability [45]. In this thesis, we are mainly interested in achieving high performance at high energy efficiency, but we will also discuss the programming models that we use and compare the programmability of various (accelerator) architectures.

In the early days of accelerator computing, the theoretical performance available increased with newer generations of accelerators faster than the achieved performance [46]. This is called Realizable Utilization (RU). Today, accelerator programming has matured significantly, for instance through the advancements in programming eco-systems such as Compute Unified Device Architecture (CUDA) [2] which offers high-performance libraries for common operations (e.g. for Fourier transformations) and profiling tools. To meet the exascale requirements as outlined above, we try to maximize the RU for every accelerator that we study and present a thorough performance analysis to identify any bottlenecks.

## 2.6 Metrics

Throughout this thesis we use a number of different metrics to describe our results, these are defined in Table 2.2. Most of the performance and energy efficiency metrics are commonly used in the field of Computer Science.

Arguably the most common performance metric is Flop/s, which is also typically used to express the theoretical performance a computer chip. We count the number of floating-point operations in the different compute kernels and divide this number by the measured runtime of the kernel. We verify that the floating-point operation

counts are correct by comparing them with the actual number of operations performed as reported by performance counters built into the chips.

As we describe in Chapter 5 through 7, the flop count of operations such as $sine(x)$ is implementation-dependent. Counting all operations equally, e.g. $+, -, *, \sin()$ or $\cos()$ are all counted as a single operation, serves as a workaround but is not ideal when comparing devices that implement $sin()$ and $cos()$ differently.

We need an implementation and hardware agnostic performance metric and therefore introduce *Vis/s*: the number of input samples (visibilities in our case) processed per second. This metric does not depend on implementation details of individual operations and is easy to measure and to interpret.

For similar reasons as outlined above, Flop/W might be a common metric for energy efficiency, it is not a very useful one when describing the efficiency of IDG. Instead, we *Vis/J* to denote the number of visibilities that can be processed for every Joule consumed. For both Vis/s and Vis/J, higher is better.

Finally, we occasionally refer to the quality of a computed data product, either in terms of accuracy compared to a reference implementation or in terms of dynamic range.

***Table 2.2:*** *Performance, efficiency, and quality metrics used in this thesis.*

| Name | Unit | Description |
|---|---|---|
| runtime | s | wall-clock time in seconds |
| performance | Op/s | number of operations performed per second |
| performance | Flop/s | number of floating-point operations per second |
| throughput | Vis/s | number of visibilities processed per second |
| energy consumption | J | number of Joules consumed |
| energy consumption | W | instantaneous energy consumption in Joule/s |
| energy efficiency | Op/J | number of operations performed per Joule |
| energy efficiency | Flop/W | number of floating-point operations per Watt |
| energy efficiency | Vis/J | number of visibilities processed per Joule |
| accuracy | $\sigma$ | floating-point deviation wrt. a reference value |
| dynamic range | no unit | quality of the image (signal-to-noise ratio) |

Chapter 3

# Energy Efficiency Analysis

The contents of this chapter are based on the following paper:

**PowerSensor 2: A Fast Power Measurement Tool**
Romein, J. W., **Veenboer, B.**
In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 111-113, IEEE, 2018*

In this chapter we answer the following research question:

> **RQ1:** *How do we compare the energy consumption of CPUs and accelerators?*

## 3.1 Introduction

To assess the energy efficiency of an architecture, we need a way to measure power consumption. For CPU-based systems, we measure the power consumption of the CPU (or CPUs for multi-socket machines) and the DRAM using LIKWID [47]. LIKWID is a software tool that provides a convenient wrapper to read performance counters built in the CPU and exposes these to the user. By taking measurements right before and right after some region of code, we can measure the power consumption of individual kernels. We do not take auxiliary power consumption (for instance for storage or cooling) into account. Some GPUs manufactured by NVIDIA support the NVIDIA Management Library (NVML), which can be used in a similar fashion as LIKWID to measure power-consumption of the device. NVML however has a too low time resolution to accurately measure individual kernels. Furthermore, we would like to measure power consumption for (PCIe) devices that do not have support power measurements. Therefore, we created PowerSensor [48] to measure the instantaneous power consumption of PCIe cards and SoC development boards like GPUs, Xeon Phis, FPGAs, DSPs, and network cards, at sub-millisecond time scale. The remainder of this chapter is organized as follows: in Section 3.2 we introduce PowerSensor, we describe the different operations modes that PowerSensor supports, and we show a few examples of insights obtained with PowerSensor. In Section 3.3 we discuss related work and in Section 3.4 we conclude.



## 3.2 PowerSensor

PowerSensor is a low-cost, custom-built device that measures the instantaneous power consumption of GPUs and other (peripheral) devices at high time resolution. It consists of an Arduino Leonardo (or Arduino Pro Micro) microcontroller board, current sensors (ACS712) a PCIe riser card (to measure the power drawn from the motherboard), an optional LCD screen, and a USB cable that is connected to the host. Fig. 3.1 shows a typical use case of a PowerSensor attached to a GPU. In this scenario, we use three sensors that measure the power drawn through the PCIe slot (12V and 3.3V) and the external PCIe cable. As the microcontroller has only one ADC, it reads the sensors one after another and reports the measurements via USB to the host.

A small host library assists an application to determine its energy efficiency. The high time resolution (up to 8.62 kHz) provides much better insight into energy usage

than low-resolution built-in power meters (if available at all), as PowerSensor enables analysis of individual compute kernels, which typically run for milliseconds.

The tool has been used successfully to analyze several applications on PCIe devices like GPUs, a Xeon Phi, a 40 GbE network card, and an FPGA, as well as SoC development platforms like the Jetson TX1 and an EVMK2H DSP board [49, 50, 11]. The firmware, host library, support programs, and how-to-build-it-yourself manual are available for download [51].

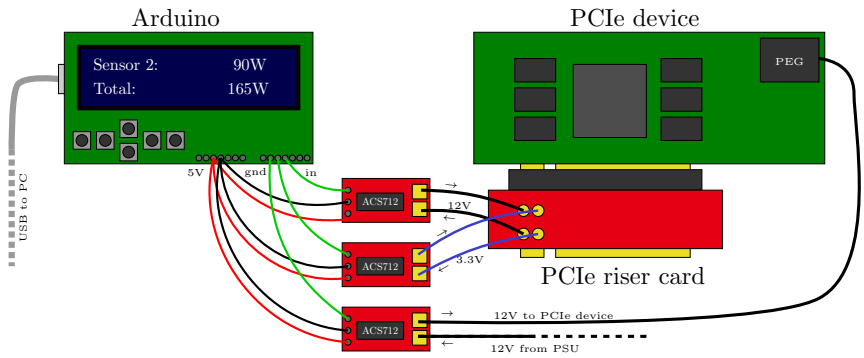



***Fig. 3.1:*** *A PowerSensor measures the instantaneous power use of a PCIe device.*

### 3.2.1 Operation modes

With ***interval-based measurements***, an application measures the power use of a device during some time interval. At the start and at the end, the application invokes a library function that returns an object that represents the instantaneous power state. With these two objects, the application asks the library how much energy or time was spent during the interval (in Joules, Watt, or seconds). An application can then determine its own energy efficiency by dividing the number of operations (obtained by profiling or analytically) by the measured energy use. The library interface (listing 3.2) is easy to use (listing 3.3).

As *GPU* kernels are typically launched asynchronously by enqueuing them to some stream, the PowerSensor state must be read by a callback function that is invoked whenever the GPU kernel starts or stops executing. Both CUDA and OpenCL support these callback functions.

In ***continuous measurements*** mode, PowerSensor writes a stream of consecutive measurements to file. The library starts a low-overhead thread that runs

```
class PowerSensor {
  public:
    ...
    State read();
};

double Joules(const State &first, const State &second);
double Watt(const State &first, const State &second);
double seconds(const State &first, const State &second);
```

***Listing 3.2:*** *PowerSensor host library interface.*



```
int main()
{
  PowerSensor sensor("/dev/ttyACM0");
  State start = sensor.read();
  ... // do work, e.g. on GPU
  State stop = sensor.read();
  cout << "It used " << Joules(start, stop) << 'J' << endl;
}
```

***Listing 3.3:*** *Example use of the host library.*

asynchronously with the application and writes tuples of the current time and wattage to file (in ASCII), 8,620 times per second. The library allows the application to put markers in this file, e.g., to annotate an event such as the start of a particular kernel execution. These markers can be cross-correlated with the power measurements. The file can be easily used to create time-vs.-power graphs by any plotting tool that allows ASCII input, as shown below.

Modifying the application source code to use the library is not obligatory; the included `psrun` utility can monitor the power use of a device during the execution of an unmodified application. This utility also supports the continuous measurements mode.

### 3.2.2 Installation and usage

The installation of this tool requires some basic skills in electronics, but is not excessively difficult. Once installed, its use is simple. The number and types of the used current sensors, the voltages of the power lines, and calibration weights are easily configurable using the `psconfig` utility.

The Hall-effect current sensors must be calibrated for the local magnetic field, by performing a null-level measurement with the connected device turned off. To measure very low currents, an ACS712 sensor board with an integrated voltage multiplier can be used. A PCIe riser card of sufficient quality is needed to maintain the PCIe signal integrity; we use the Adexelec PEXP16-EX.

As we do not measure voltages, the power supply voltage must be stable, also under varying load. The sensor and ADC error tolerances add up to 3.7%, but with proper calibration, our measurements are typically within 1% of built-in GPU power meters and lab equipment.

The time resolution is 116µs times the number of attached sensors, and is limited by the ADC conversion time.

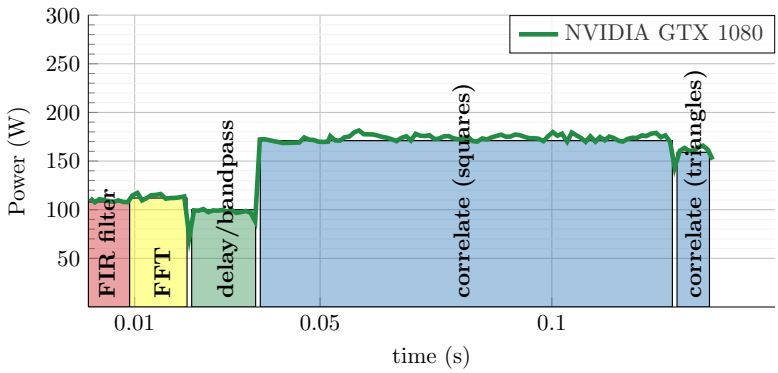


### 3.2.3 Examples

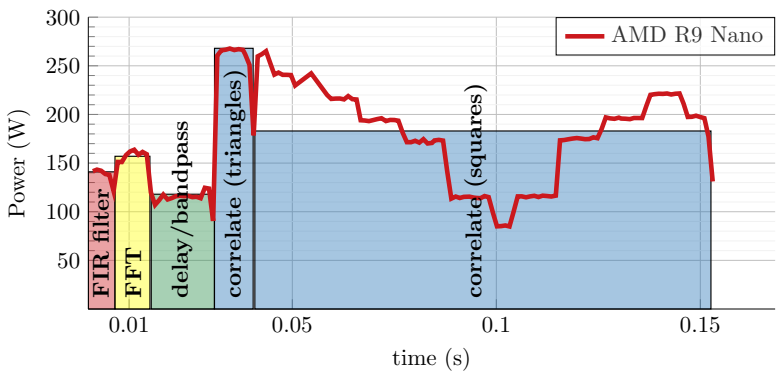
PowerSensor gives insight into an application's power efficiency, as illustrated by Figure 3.4. This radio-astronomical pipeline filters, corrects, and correlates the signals from 960 receivers [50]. The figure shows the instantaneous power consumption of three different devices. The correlation between energy use and executed kernels is clearly visible. The shaded area below the curve corresponds to the total energy used by a kernel. On the NVIDIA GTX 1080 GPU (Fig. 3.4a), some kernels draw much more power than others.

The AMD R9 nano GPU (Fig. 3.4b) has a Thermal Design Power (TDP) of 175W. However, the graph shows temporary power consumption as high as 275W. After 13ms, the device starts throttling, stepwise reducing power usage to as low as 85W to compensate for the excess power usage, then jumping back to 220W. The long-term average power consumption is indeed 175W. The performance of one kernel depends strongly on the power usage of the other kernels: the correlate-triangles function runs at a high clock frequency and thus a high power consumption because the first three functions did not use their full power budget. A high-time-resolution tool like PowerSensor is indispensable to analyze this behavior.

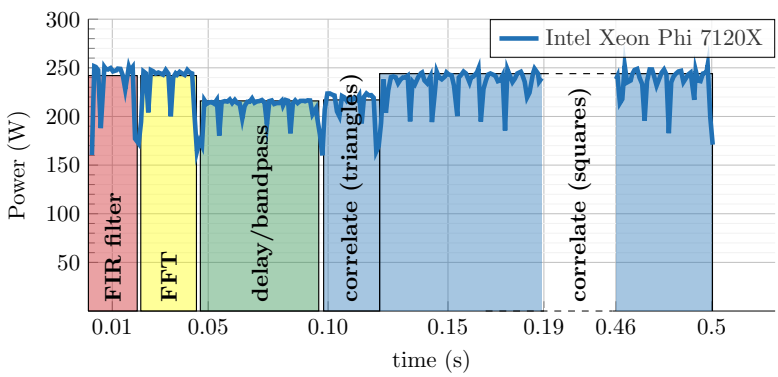
The graph for the Xeon Phi 7120X (Fig. 3.4c) shows repetitive power dips at a 100Hz rate. The Linux kernel periodically interrupts all cores, during which much less energy is drawn than when the application performs heavy vector computations. We discovered this behavior with PowerSensor; we did not notice it when profiling the application with VTune Amplifier.




**(a)** NVIDIA GTX 1080




**(b)** AMD R9 Nano




**(c)** Intel Xeon Phi 7120X

***Fig. 3.4:*** *Continuous measurement of the power consumption for various PCIe devices using PowerSensor. With the high time-resolution of PowerSensor, the correlation between kernel execution and energy consumption is clearly visible. Some kernels (e.g. correlation) consume more energy than others (e.g. delay and bandpass correction). Furthermore, features such as TDP limits (in case of AMD R9 Nano) and kernel interrupts (in case of Intel Xeon Phi 7120), are visible.*

## 3.3 Related work

There are many power-measurement tools that bear some resemblance. They are all elegant in some aspects, but none of them combines all advantages of high time resolution, simplicity, low cost, availability, and full library support. PowerInsight [52] measures both voltages and currents, but has lower time resolution and does not support the interval-based mode described above. PowerMon 2 [53] uses a well-designed but difficult-to-obtain custom PCB; it also cannot handle 150W PCIe power cables. Ilsche et al. present a highly accurate but costly and complex method [54]. Others have built their power measurement tools, for example, to validate a power estimation framework for GPUs (GPUSimPow) [55], or to analyze the power behavior of the Xeon Phi [56], but these tools are not publicly available.



## 3.4 Conclusion

The ability to perform power measurements at high time resolution is indispensable to study the power efficiency of individual compute kernels on accelerators like GPUs or the Xeon Phi. PowerSensor is an easy-to-use tool that performs these measurements at millisecond time scale. PowerSensor can be used for peripheral devices like PCIe cards but also for SoC development boards. PowerSensor consists of commodity components. PowerSensor reports measurements back to the host processor, via USB. An application can link to a simple library to analyze its energy efficiency (e.g., of GPU compute kernels), but the power can also be measured without modifying the application. The high time resolution, low cost, ease of use, and public availability make PowerSensor a useful tool for power measurements of (peripheral) devices.

> **RQ1:** *How do we compare the energy consumption of CPUs and accelerators?*

The energy consumption of CPUs is measured in software. Some accelerators have built-in power meters, but these are typically limited in time resolution. We designed PowerSensor to overcome this limitation. We use these tools throughout the thesis to analyze the energy efficiency of individual compute kernels.

Chapter 4

# Image-Domain Gridding

The contents of this chapter are based on the following papers:

**Image-Domain Gridding on Graphics Processors**
**Veenboer, B.**, Petschow, M., Romein, J. W.
in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), pages 545–554, IEEE, 2017*

**Radio-Astronomical Imaging on Graphics Processors**
**Veenboer, B.** Romein, J. W.
In *Astronomy & Computing, Elsevier, Volume 32, July 2020*

In this chapter we answer the following research question:

> **RQ2:** *How do we efficiently implement Image-Domain Gridding on accelerators?*

## 4.1 Introduction

The Image-Domain Gridding (IDG) algorithm is a novel imaging approach and is described and analyzed from a mathematical perspective in [1]. In [11], we analyzed its performance on an Intel Xeon CPU and GPUs from both AMD and NVIDIA. Section 4.2 provides a high-level overview of the IDG algorithm. We describe how this algorithm elegantly maps onto parallel devices by providing a work division strategy in Section 4.3. In [15] we added a discussion on the precision requirements and the complexity of the IDG algorithm. These topics are addressed in Section 4.4 and Section 4.5. To make IDG accessible for use by astronomers, we integrated IDG into the widely used WSClean [31] imaging application. Our IDG implementations are currently being used to process the data for various radio telescopes, such as LOFAR [6] and the MWA [7].



## 4.2 Algorithm

W-projection and AW-projection apply a convolution kernel to each of the visibilities, see the top-right panel of Fig. 4.1. In W-projection, these kernels depend on the $(u, v, w)$-coordinate associated with the visibility. For AW-projection, these kernels additionally depend on time and baseline. In practice, the convolution kernels are precomputed on an oversampled grid. The convolution kernels in W-projection or 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 quadratically with the oversampling factor. The classical convolution theorem [57] states that the Fourier transform of a product is the convolution of the Fourier transforms: $\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$ IDG [1] makes use of this theorem to replace the convolution operation with a multiplication, followed by a Fourier transform. This way, both W-correction, and A-correction are performed in the *image domain*. Consequently, oversampled convolution functions are not needed.

At the center of the Image-Domain Gridding (IDG) [1] algorithm are *subgrids*, which represent low-resolution versions of the sky brightness for a small subset of visibilities, (see Figs. 4.1 and 4.2). IDG maps visibilities to subgrids by performing a direct Fourier Transform at subgrid resolution. This operation is similar to a direct evaluation of the measurement equation (see Chapter 2), but at subgrid-resolution instead of at the resolution of the full grid. 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

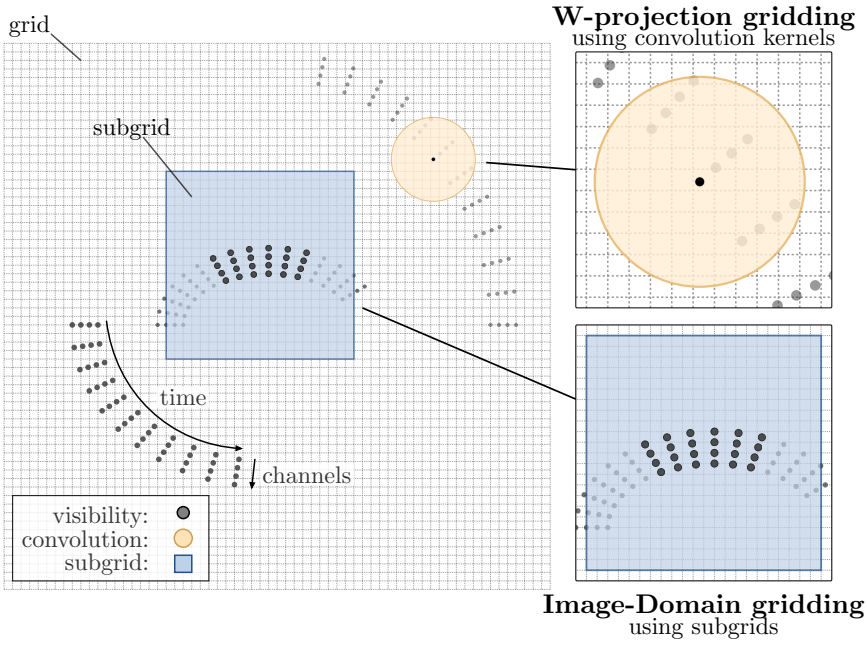



***Fig. 4.1:*** *In traditional W-projection and AW-projection gridding, visibilities are gridded using convolutions in the uv-domain (top-right) as opposed to correcting the W-term and A-term effects in the image domain (bottom right). For the latter, neighboring visibilities (indicated with thick dots) are gridded on small 'subgrids'. After gridding, a 2D FFT brings subgrids to the frequency domain, see also Fig. 4.3.*

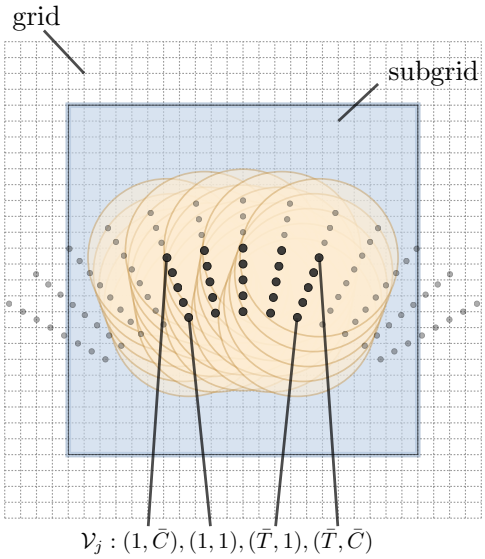



***Fig. 4.2:*** *A subset of visibilities ($\mathcal{V}$, black dots), including their associated AW-projection convolution kernels (yellow circles), is covered by a subgrid.*

orders of magnitude smaller than the size of the grid $N \times N$. This approach has the added benefit of allowing for cheap application of W-terms and A-terms.

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 [1].) 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 Image-Domain gridding and degridding is illustrated in Figure 4.3. The first step in gridding is to place visibilities onto subgrids. This step is performed by the *gridder kernel*, which applies Algorithm 1 for every subgrid $s$.

The $cexp(phase)$ evaluation in Line 9 of this algorithm comprises one evaluation of $cos(phase)$ and one evaluation of $sin(phase)$. *cmul* denotes a complex multiplication, which comprises four multiply-add operations. Thus for every evaluation of $cexp(phase)$ in Line 9, 17 real-valued multiply-add operations are performed, one in the computation of *phase* in Line 8 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 relative 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

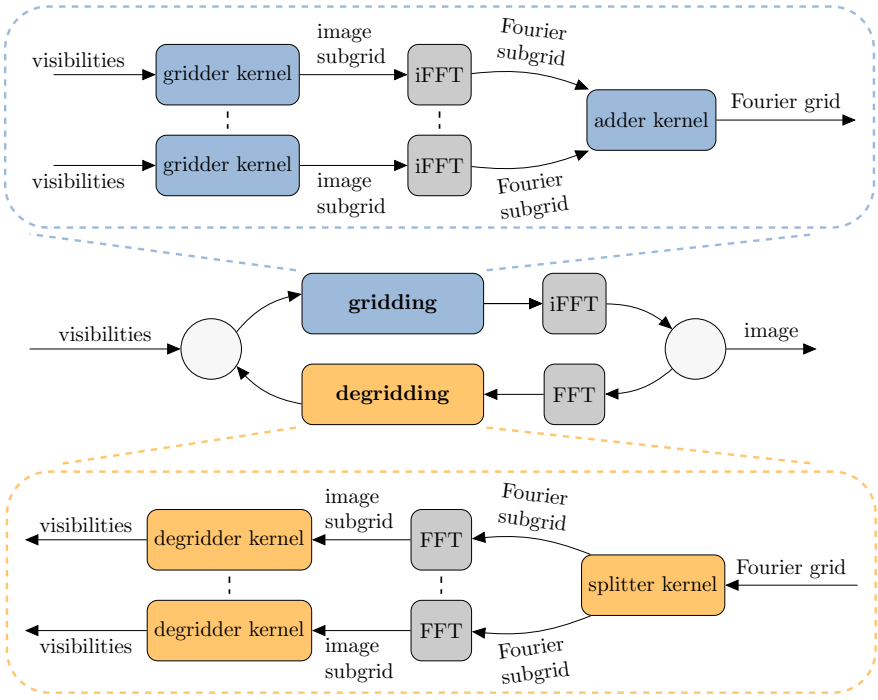



***Fig. 4.3:*** *Gridding consists of three steps: (1) the visibilities are gridded onto subgrids by the* gridder kernel *which applies an operation that resembles a DFT to each of the visibilities; (2) subgrids are Fourier transformed by applying an inverse 2D FFT; (3) the subgrids are added to the grid by the* adder kernel*. Degridding is similar to the gridding, but proceeds in reverse order: (1) subgrids are extracted from the grid by a* splitter kernel*; (2) subgrids are Fourier transformed by a FFT kernel; (3) the* degridder kernel *computes visibilities.*

```
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 apply_ifft(subgrid);
20 store(subgrid);
```

***Algorithm 1:*** *Image-Domain Gridding pseudocode that is executed for every subgrid $s$ in the gridder kernel. This routine is a drop-in replacement for the gridding step shown in Fig. 2.6. The variables $\bar{T}$ and $\bar{C}$ denote the number of visibilities in time and frequency mapped to a subgrid, respectively. $P$ is the number of correlations per visibility. In case of IDG, we assume $P = 4$, for visibilities corresponding to all four combinations of polarizations $X$ and $Y$.*

to compute a phase shift in Line 8. Before subgrids are stored (Line 20), correction for DDEs (Line 17) and a tapering function (Line 18, to suppress aliasing, see [1]) are applied.

In IDG, these convolution kernels and the tapering function are two-dimensional arrays where the size in the number of pixels in one dimension is given by $N_W$. Consequently, the minimum size of the subgrid $\bar{N} \geq N_W$. 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. 4.2).

## 4.3 Execution plan

Before gridding or degridding starts, an *execution plan* is generated that specifies the subgrid locations and associated visibilities. If $\mathcal{V} = \{V^{(q,r)}(t,c)\}$ denotes the

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

***Algorithm 2:*** *Degridding is the inverse operation of gridding, visibilities are computed taking a grid as input. The degridder kernel computes visibilities for every subgrid $s$.*

visibilities from all time steps $T$ and baselines $(q, r)$, the positions of the subgrids induce a partitioning $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \ldots \cup \mathcal{V}_n$. 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. 4.2, not only the visibilities need to be covered by the subgrids, but also the surrounding support of their associated AW-projection convolution kernels [1]. 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.

If for $\bar{T}$ visibilities not all frequency channels can be covered by a single subgrid the frequency channels are split into *channel groups* and mapped to distinct subgrids. Each channel group has a configurable maximum number of frequency channels $\bar{C}$. For every channel group, the execution plan (mapping of visibilities to subgrids) is created as described above.

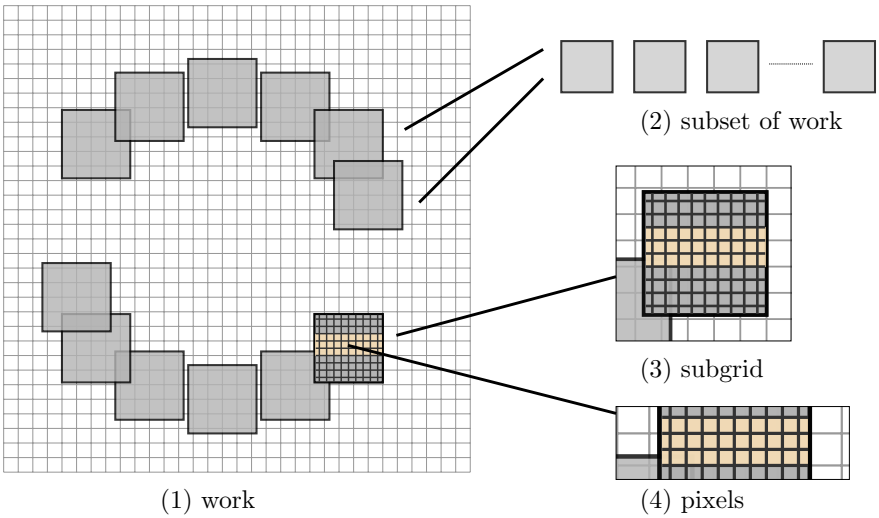



***Fig. 4.4:*** *Work division within IDG: (1) The* work *is partitioned into (2) subsets. (3) Every subset contains tasks, represented by individual subgrids. (4) For gridding, the smallest unit of computation is a single pixel. For degridding, the smallest unit of computation is a single visibility.*

Since A-term correction is applied on a per-subgrid basis, neighboring visibilities corresponding to different baselines are mapped to distinct subgrids to allow baseline-dependent A-term correction.

We might additionally require that $\bar{T} \leq \bar{T}_{\max}$ (where $\bar{T}_{\max}$ is architecture-specific) to limit the maximum number of time steps that are associated with a single subgrid. This approach controls the number of computations to be performed for each subgrid, and therefore provides a load-balancing mechanism.

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, a *task*. The set of all $n$ tasks is called the *work* and is generated by an execution plan. Subsets of the work are processed by the gridder and degridder kernels, using Algorithm 1 and Algorithm 2 for every task, respectively. This work division hierarchy is illustrated in Fig. 4.4, and we will refer to it later to show how the algorithm is mapped differently onto the distinct architectures.

## 4.4 Single-precision gridding

A common question for any scientific code is what level of precision is required. For any computation, as well as for any (intermediate) data structure, one might choose between different data types, for instance single-precision, or double-precision floating-point numbers and/or arithmetic.

In radio-astronomical imaging, the dynamic range of a sky image is an important quality metric. 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 [58–60].

Inaccuracies, which may or may not influence the final sky image, could potentially be introduced during gridding, where a pixel in the grid is repeatedly updated while the visibilities are added to that pixel. This might lead to rounding errors, as a contribution to a pixel might be (very) small, while the pixel itself has a large value. In IDG this is less of an issue than for traditional convolution-based gridding, because the number of updates per pixel is smaller due to the use of subgrids.

Most compute architectures provide at least double the theoretical peak performance when using single-precision arithmetic instead of double-precision arithmetic. This gives us a clear incentive to favor single-precision floating-point for IDG.

In a configuration as typically used in radio astronomy, IDG (using single-precision) is shown to provide comparable accuracy to classical W-projection gridding [1]. Therefore, we use a single-precision floating-point format for the computations and data structures in all our implementations and omit the term “single-precision“ from here on.



## 4.5 Complexity

We now determine the complexity of IDG, using the symbols listed in Table. 4.1. The first step in gridding is the gridder kernel. The complexity of this step 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:

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

***Table 4.1:*** *Symbols used in the complexity analysis of IDG.*

| Symbol | Description |
|---|---|
| $\bar{T}$ | number of visibilities in the time dimension on a subgrid |
| $\bar{C}$ | number of visibilities in the frequency dimension on a subgrid |
| $\bar{N}$ | number of pixels in one dimension of the subgrid |
| $\bar{V}$ | average number of visibilities per subgrid |
| $T_{obs}$ | total number of visibilities in the time dimension for one baseline |
| $C_{obs}$ | total number of visibilities in the frequency dimension for one baseline |
| $V_{obs}$ | total number of visibilities in observation |



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. 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.

For a dataset with $V_{obs}$ visibilities, $S$ is given by: $S = V_{obs} \times \bar{V}^{-1}$, therefore:

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

This expression illustrates that the workload scales linearly with the number of visibilities and quadratically with the size of the subgrid. Furthermore, the workload of the 2D FFT and adder kernel compared to the workload of the gridder kernel directly follows from the visibility density. We illustrate these properties with two examples.

First, we assume $\bar{T} = 128$ and $\bar{C} = 16$, and normalize $\mathcal{O}_{gridding} = 1$ for $\bar{N} = 32$. For $\bar{N} = [24, 48, 64]$, $\mathcal{O}_{gridding} = [0.56, 2.25, 4.0]$. Thus doubling the size of the subgrid results in a quadratic growth of gridding complexity. In all four cases, $\mathcal{O}_{gridder} \gg (\mathcal{O}_{fft} + \mathcal{O}_{adder})$: the workload of the 2D FFT and adder kernel are negligible with respect to the gridder kernel.

Next, we use $\bar{N} = 32$ and set $\bar{V} = [1, 2, 4, 8, 16]$ to find $\mathcal{O}_{fft} + \mathcal{O}_{adder} = [0.52, 0.26, 0.13, 0.06]$. This illustrates that with a low visibility density ($\bar{V} \leq 8$), the workload of the 2D FFT and adder kernel is noticeable.

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:

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

## 4.6 Conclusion



Image-Domain Gridding comprises two main steps: gridding *and* degridding. We explained how the *execution plan* helps to break these steps down into substeps that can be implemented by various distinct compute kernels. In Part II we demonstrate how we implement these kernels for (many-core) CPUs, as well as on Graphics Processing Units (GPUs) from both AMD and NVIDIA, and on Field Programmable Gate Arrays (FPGAs).

We made the IDG source code available online [14]. We have also integrated IDG with the WSClean imager [31] such that all its features (data handling, deconvolution, etc.) are maintained, while the existing inversion (gridding) and predict (degridding) functionalities are replaced by IDG. The gridding and degridding steps are split into smaller sub-steps (i.e. the gridder kernel, and the adder kernel) such that these sub-steps can be executed efficiently on a variety of (accelerator) hardware.

> **RQ2:** *How do we efficiently implement Image-Domain Gridding on accelerators?*

We use an *execution plan* to map input data to *subgrids* and split the computation into distinct kernels that can be implemented and optimized separately. This approach enables us to offload certain parts of the computation (e.g. the gridder kernel) to an accelerator, while others (e.g. the 2D FFT of the grid) are performed at the host.

Part II

# *Image-Domain Gridding Implementation and Analysis*

Chapter 5

# IDG on CPUs

The contents of this chapter are based on the following papers:

**Image-Domain Gridding on Graphics Processors**
**Veenboer, B.**, Petschow, M., Romein, J. W.
in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), pages 545–554, IEEE, 2017*

**Radio-Astronomical Imaging on Graphics Processors**
**Veenboer, B.** Romein, J. W.
In *Astronomy & Computing, Elsevier, Volume 32, July 2020*

In this chapter we answer the following research sub question:

**RQ3a:** *How efficient is IDG on CPUs?*

## 5.1 Introduction

In this chapter, we will describe our implementation of the Image-Domain Gridding algorithm for CPUs. We first provide an overview of the CPU architectures that we target in Section 5.2. Next, in Section 5.3 we describe the implementation of the various parts of the algorithm.

While we use well-known optimization techniques such as multi-threading, vectorization, and loop transformations, the use of these techniques to implement optimized Image-Domain Gridding CPU kernels is new. Most importantly, this implementation serves as a baseline to compare our implementations for accelerators (in Chapter 6 and 7) against.

We analyze the performance and energy efficiency of our IDG implementation for CPUs in Section 5.4.



## 5.2 Architecture

We consider two types of CPU series: Intel Xeon (Haswell architecture) and Intel Xeon Phi (Knights Landing architecture). Xeon is Intel's server-grade series of CPUs, while Xeon Phi is a many-core processor targeted at energy-efficient high-performance computing (see also Section 2.3).

The Haswell architecture implements the AVX2 instruction set, which supports 8-element single-precision vector instructions, including fused multiply-add (FMA). These FMA instructions double the throughput of computations like $a = a + (b * c)$ compared to issuing the multiply and add instructions separately.

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 can only be achieved when these vectorized FMA instructions are used.

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 DDR4 DRAM.

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.

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.

## 5.3 Implementation

All CPU kernels are implemented in C++ and are executed for every subset of the work: tasks are distributed over all logical cores using OpenMP. Given a sufficiently large number of subgrids (which is typically the case), this method scales linearly with the number of logical cores as we will show in Section 5.3.6.

IDG needs to exploit both thread-level and vector-level parallelism to fully utilize the CPU. Finding the right granularity for both levels is key to good performance and scalability. We now detail our strategy for various subparts of the algorithm (see also Figure 4.3).



### 5.3.1 Gridder kernel

The gridder kernel (as all other CPU kernels implemented in C++) is executed for every subset of the work. It distributes the tasks over all logical cores using OpenMP. In other words, each thread computes a subset of the subgrids according to Algorithm 1. We provide pseudo-code for this gridder kernel in Algorithm 3.

The most important performance optimizations are the following: (1) We prefetch and transpose all visibility data associated with a subgrid into a memory-aligned array to allow for fast, non-strided data access in the inner loops of the kernel. At this moment, for a better mapping to the instruction set, we also separate the real and imaginary part of the operands; (2) the sine/cosine-computations (Line 9 of Algorithm 1) are precomputed, see Section 5.3.5 for details; (3) we collapse the time and channel loops (Line 4 and 6); (4) the computation in the remaining inner-loop is written in the form of an reduction over visibilities as illustrated in Algorithm 4; (5) we use intrinsics (in the reduction) to force the compiler to use the desired (vector) instructions.

### 5.3.2 Degridder kernel

We distribute the work in the same manner as in the gridder kernel: each thread processes a subset of the work by applying Algorithm 2 to every task. In other words, each thread computes the visibilities for a subset of the subgrids. The kernel

```
1  #pragma omp parallel
2  for s = 1..S do
3      complex<float> subgrid[P][N̄×N̄];
4      float offset[N̄×N̄] = compute_offset(s);
5      float vis_real[P][T̄×C̄] = transpose_visibilities_real(s, visibilities);
6      float vis_imag[P][T̄×C̄] = transpose_visibilities_imag(s, visibilities);
7      for i = 1..N̄×N̄ do
8          complex<float> pixel[P];
9          for t = 1..T̄ do
10             float index = compute_index(s, i, t);
11             for c = 1..C̄ do
12                 float scale = load_scale(c);
13                 float phase = offset - (index * scale);
14                 complex<float> phasor = cos(phase) + i sin(phase);
15                 for p = 1..P do
16                     complex<float> visibility = load(vis_[real|imag], t, c, p);
17                     pixel[p] += visibility * phasor; // complex multiply–accumulate
18                 end
19             end
20         end
21         for p = 1..P do
22             apply_aterm(pixel[p]);
23             apply_taper(pixel[p]);
24             subgrid[p][i] = pixel[p];
25         end
26     end
27     apply_ifft(subgrid);
28     store(subgrid);
29 end
```

***Algorithm 3:*** *This pseudo-code for the CPU gridder kernel illustrates that multiple subgrids are processed in parallel. The computation inside the loops over time ($\bar{T}$) and frequency channels ($\bar{C}$) resembles a discrete Fourier transform (DFT) of visibilities to an image. Operations such as computing the* offset *(the position of the subgrid in the grid), and loading and transposing of input data (e.g. visibilities) take place outside the critical path. In the loop over pixels, an array of phasor values is computed (one for every visibility). A pixel is computed as the (complex) dot product of phasor and visibilities. This operation is implemented as a reduction (see Algorithm 4) and vectorized using fused multiply-add instructions. The apply_aterm and apply_taper operations are performed only once per pixel and are therefore not performance-critical.*

```
1  #pragma omp simd reduction(+ : ...)
2  for v ∈ VISIBILITIES do
3      Re(pix_11) += Re(vis_11[v]) * Re(phasor[v]);
4      Im(pix_11) += Re(vis_11[v]) * Im(phasor[v]);
5      Re(pix_11) -= Im(vis_11[v]) * Im(phasor[v]);
6      Im(pix_11) += Im(vis_11[v]) * Re(phasor[v]);
7
8      // [... same for pix_12 and pix_21]
9
10     Re(pix_22) += Re(vis_22[v]) * Re(phasor[v]);
11     Im(pix_22) += Re(vis_22[v]) * Im(phasor[v]);
12     Re(pix_22) -= Im(vis_22[v]) * Im(phasor[v]);
13     Im(pix_22) += Im(vis_22[v]) * Re(phasor[v]);
14 end
```

***Algorithm 4:*** *The complex multiplication of visibility with phasor and addition to pixels (Line 12 in Algorithm 1) comprises four FMAs. The loop over polarizations (Line 10) is fully unrolled. Thus for every visibility, a total of 16 FMAs are executed. The* reduction *clause in this pseudocode instructs the compiler to process multiple visibilities at once using vector instructions.*



optimizations are similar to the optimizations for the gridder kernel. A notable difference is that we apply vectorization over pixels (Line 8 of Algorithm 2) instead of over visibilities, see the pseudo-code in Algorithm 5. This way, the computation of a visibility can be implemented as a complex dot product of *phasor* and pixels.

### 5.3.3 FFTs

All subgrids are Fourier-transformed before adding them to the grid and after splitting the subgrids from the grid, for gridding and degridding, respectively. (see Figure 4.3). This is a parallel process and most efficiently done by using a math library such as FFTW or Intel's Math Kernel Library (MKL).

In a full imaging cycle, the grid is Fourier transformed after all visibilities are gridded. Since the grid contains four polarizations, this effectively amounts to executing four distinct FFTs. These are (much) larger transformations than the many transformations applied to the individual subgrids.

We implemented the large FFT operation in two ways, using fine-grained parallelism, where multiple threads perform the transformation and using coarse-grained parallelism with one thread per polarization, and up to four polarizations in parallel. We show the runtime for different grid sizes in Fig. 5.1 for both FFTW and MKL.

```
1  #pragma omp parallel
2  for s = 1..S do
3      complex<float> subgrid[P][N̄×N̄] = load_subgrid(s);
4      apply_fft(subgrid);
5      apply_taper(subgrid);
6      apply_aterm(subgrid);
7      float pixels_real[P][N̄×N̄] = transpose_subgrid_real(subgrid);
8      float pixels_imag[P][N̄×N̄] = transpose_subgrid_imag(subgrid);
9      float offset[N̄×N̄] = compute_offset(s);
10     for t = 1..T̄ do
11         for c = 1..C̄ do
12             complex<float> visibility[P];
13             for i = 1..N̄×N̄ do
14                 float index[N̄×N̄] = compute_index(s, i, t);
15                 float scale = load_scale(c);
16                 float phase = (index * scale) - offset;
17                 complex<float> phasor = cos(phase) + i sin(phase);
18                 for p = 1..P do
19                     complex<float> pixel = load(pixels_[real|imag], p, i);
20                     visibility[p] += pixel * phasor; // complex multiply–accumulate
21                 end
22             end
23             for p = 1..P do
24                 visibilities[t][c][p] = visibility[p];
25             end
26         end
27     end
28 end
```

***Algorithm 5:*** *The CPU degridder kernel uses the same optimizations as the gridder kernel, but proceeds in the reverse order. First, a subgrid is loaded and A-term correction and tapering are applied. Afterwards, the pixels of the subgrid are transposed into seperate buffers. The phasor values are computed in the same manner as in the gridder kernel. Unlike in the gridder kernel where we unrolled the loops over time and frequency, we unroll the loop over pixels such that a visibility can be computed as the complex dot product of phasor and pixels.*

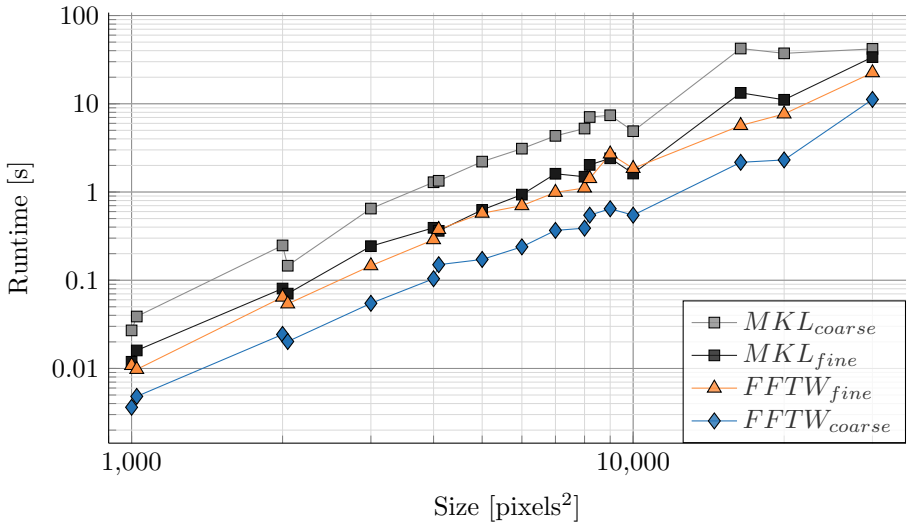



***Fig. 5.1:*** *Fourier transformation runtime for grids of different size, using MKL and FFTW in two modes: fine (1 polarization at once, multiple threads per FFT) and coarse (multiple polarizations at once, one thread per FFT).*



Runtime naturally scales with the size of the transformation. The results indicate that FFTW using coarse-grained parallelism is the best option, as it achieves the lowest runtime for all grid sizes.

### 5.3.4 Adder and splitter kernel

As subgrids might partially overlap in the grid, for the adder, parallelization over subgrids would require atomic additions to the pixels in the grid. To avoid prohibitive synchronization costs, we use a different parallelization strategy that avoids the need to use atomic operations: (1) threads are mapped onto rows of the grid; (2) for every row, the execution plan is used to find rows of subgrid pixels that intersect the current row in the grid; (3) the corresponding subgrid pixels are loaded from memory and added to the grid. For the splitter, overlapping subgrids are not a problem as the data from the grid is read-only. Therefore, multiple subgrids can safely be read in parallel.

### 5.3.5 Sine/cosine computations

The critical path of the gridder and degridder kernels consists of three parts: (1) computation of *phase*; (2) computation of *phasor*; (3) a complex multiplication and addition. Both the *phase* computation and the complex multiplication and addition are implemented using (vectorized) fused multiply-add operations. The *phasor* value on the other hand is computed by evaluating sine and cosine. The CPU architectures that we evaluate do not have hardware instructions to perform (vectorized) sine/cosine operations. We, therefore, have to perform these operations using software libraries. There is no simple relation between a sine/cosine evaluation and the number and type of instructions executed. We found that this differs significantly between different libraries, accuracy settings (if any) and might even be input dependent. This leads to an instruction mix consisting of fused multiply-add and sine/cosine operations. We define $\rho$ as the ratio between these two operations in the critical path of the computation. For the gridder and degridder kernels, $\rho = 17$: for every evaluation of sine and cosine, 17 FMAs are performed. We measured the runtime for different values of $\rho$ and for various methods to implement sine/cosine. We compute the performance by only taking the FMAs into account and show results in Figure 5.2.



In all cases, the achieved performance is lower than the theoretical peak (indicated with the horizontal blue line) and scales linearly with $\rho$. This illustrates that evaluating sine/cosine is a costly operation and (on Haswell ) has a noticeable impact on performance. For $\rho = 17$, performance is only about 20% of the theoretical peak in the best case. This best-case performance is achieved when we use an Intel compiler (version 2018.0) in combination with Intel's Vector Math Library (VML), which is part of the Intel Math Kernel Library (MKL, version 2018.0). These libraries provide an optimized implementation of common math functions. However, these functions are evaluated in software, e.g. by executing a particular sequence of instructions. Current x86 CPUs do not support the evaluation of sine and cosine in dedicated hardware. Surprisingly, using the MKL library in combination with a GNU compiler (version 7.3.0) results in lower performance than when an Intel compiler (version 2018.0) is used. On recent systems (with Glibc 2.22 or newer), the GNU compiler uses Libmvec (an optimized open-source vector math library [61]) to compute sine/cosine using vector instructions and otherwise falls back to a scalar evaluation of sine/cosine. The use of sine/cosine computations thus heavily reduces the processor's ability to perform multiply-add operations. It leads to a lower, $\rho$-dependent peak-performance bound as we will see in Fig. 5.2.

The libraries mentioned above have different specified calculation errors. This error is typically measured in Units of Least Precision (ULP), which is a unit to describe the distance between the smallest numbers that can be represented using floating-point numbers. MKL can be configured in different modes and we use Low Accuracy (LA), for the sine/cosine computation. This mode has an error of about 2 ULPs, but we saw no significant differences in IDG's output compared to running in High Accuracy (HA) mode, which has an error of 0.5 ULPs. We suspect that due to summing over many sine/cosine products (e.g. $\bar{N} \times \bar{N}$ times for the $phasor * visibility$ product in the gridder kernel), small errors in the sine/cosine computation cancel out.

We implemented a custom lookup table based on integer arithmetic and configurable precision. Given a finite number of lookup table entries, the lookup table will inherently be inaccurate due to (slight) mismatches between the argument and the lookup table indices. Interpolation is a well-known technique to improve accuracy, but it comes at the cost of having to execute more instructions for every lookup.

We measured the error of the lookup table by comparing the subgrid values after gridding, with subgrid values in an execution where VML was used to compute sine/cosine. This analysis shows that we could obtain (up-to floating-point accuracy) the same results without interpolation, by simply increasing the number of lookup table entries.

In this setting, the lookup table implementation performs better than the GNU compiler with VML, but less well than the Intel compiler with VML. Still, this implementation might prove useful in situations where neither Intel MKL nor a recent Libmvec is available. From now on, we will only report performance using the best-performing option.

### 5.3.6 Intel Xeon Phi

The Intel Xeon Phi Knights Landing architecture is mostly binary compatible with Xeon CPUs and we, therefore, use the CPU kernels optimized for the Haswell architecture as the starting point.

We make a few additions to the aforementioned CPU kernels for the Xeon Phi: (1) We extended the batched reduction in the gridder and degridder kernels to match the vector width of 16 floating-point elements as found in the AVX-512 instruction set. (2) we use the $\tilde{T}_{\max}$ setting when creating the execution plan to limit the number of visibilities per subgrid. This exposes more coarse-grain parallelism (more subgrids are

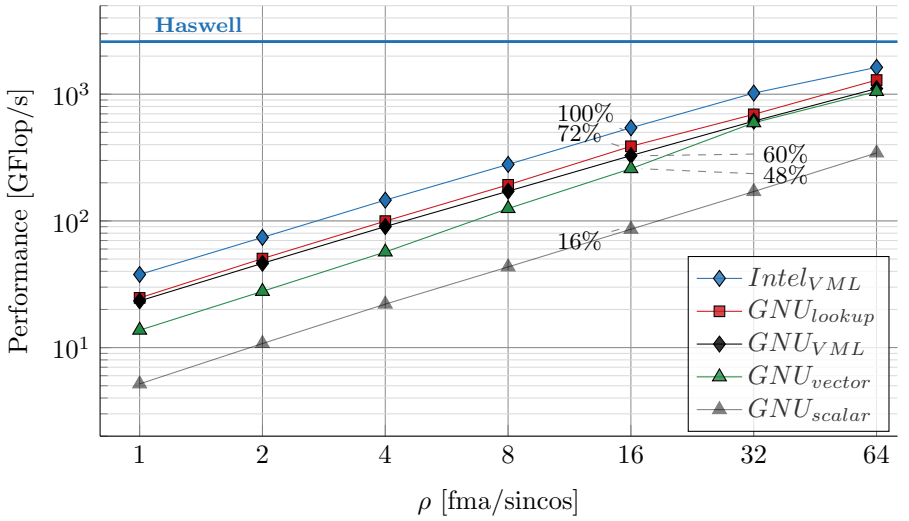



***Fig. 5.2:*** *Performance for different instruction mixes consisting of a number of FMA instructions as well as an evaluation of sine/cosine. For the gridder and degridder kernels, $\rho = 17$. The labels indicate performance relative to the fastest option ($Intel_{VML}$) at $\rho = 16$. In case of $\rho = 17$ and using VML, the CPU spends about 20% of the time on the FMAs.*

created) so that the load can be better balanced over all the cores in this many-core processor.

## 5.4 Results

After introducing our experimental setup in Section 5.4.1, we conduct a performance analysis in Section 5.4.2. Next, in Section 5.4.3 we show throughput and energy efficiency results, and in Section 5.4.4 we take a look at scalability.

### 5.4.1 Experimental setup

We perform experiments using the hardware listed in Table 5.1. We refer to this hardware as Haswell (a dual-socket system with two Intel Haswell-EP processors with 14 CPU cores each), Knl (a system with one Intel Xeon Phi Knights Landing processor with 64 CPU cores). We use the number of threads that give the best performance. Haswell is part of the DAS-5 cluster [62] and the Knl machine was

***Table 5.1:*** *The Intel Haswell-EP CPU (Xeon E5-2697v3) and Intel Knight Landing Xeon Phi (7210) used in our experiments.*

| Name | Architecture | Peak (TFlop/s) | Mem size (GB) | Mem bw (GB/s) | TDP (W) |
|---|---|---|---|---|---|
| HASWELL | Haswell-EP | 2.60 | ≤1536 | 136 | 290 |
| KNL | Knight Landing | 5.32 | ≤384 | 102 | 215 |

provided by Intel. We used the Intel compiler (version 19.0) together with Intel MKL (version 2019.0).

We use the following observation parameters: $R_{obs} = 120$, $B_{obs} = 7{,}140$, $T_{obs} = 8{,}192$, $C_{obs} = 16$ and $t_{int} = 0.9$. These parameters are mainly chosen to provide a dataset that is sufficiently large for benchmarking, but still manageable in terms of size (we generate the dataset in memory) and total execution time of the benchmark.

The number of stations is chosen to have at least a few baselines per available CPU core. While LOFAR has fewer stations and SKA will likely have many more receivers, our results can easily be extrapolated to either use-case as the performance is not dependent on the number of baselines. $T_{obs} = 8{,}192$ corresponds to about 2.5 hours of observation while observations typically last longer (e.g. 8 hours). Again, we argue that our results can easily be extrapolated. Finally, a real dataset will typically comprise many more frequency channels (grouped into subbands). We assume that the frequency channels can trivially be split into smaller sets and processed by IDG one after the other. Moreover, for these experiments, we are mainly interested in performance, energy efficiency, and throughput, not so much in absolute run-time.

The $(u, v)$-data that we generate using these parameters is shown in Fig. 2.10 and is representative for a wide range of imaging use cases. We set the imaging parameters as follows: $N_W = 9$, $\bar{N} = 32$ and $N = 8{,}192$.

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. The maximum number of channels per subgrid is not limited, thus $\bar{C} = C_{obs}$.

### 5.4.2 Performance

For a single imaging cycle, we present the execution time in Fig. 5.3. The execution time is computed as the sum of the runtime for individual kernel invocations. While KNL has double the theoretical peak floating-point performance compared to HASWELL, the runtime is only marginally lower.

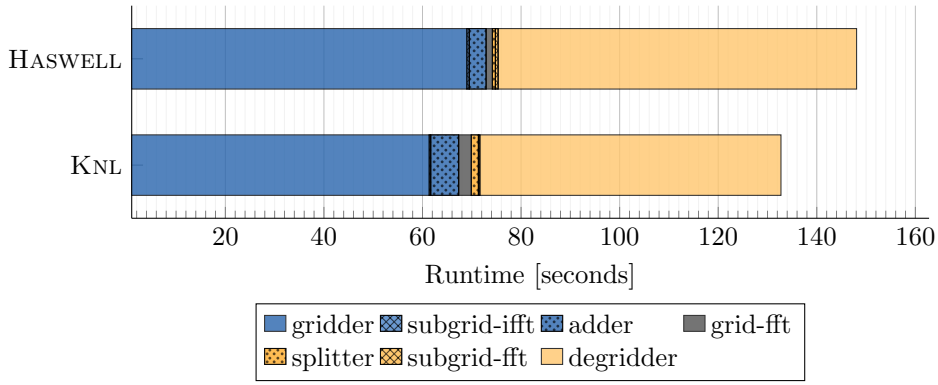



***Fig. 5.3:*** *Distribution of runtime for all kernels in an imaging cycle.*



Runtime is dominated by the gridder and degridder kernels, it accounts for 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 the performance analysis.

In Fig. 5.4 we show roofline plots [4], 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 Fig. 5.2 where we measured this value for Haswell. We used the same methodology to establish this limit for Knl.

Even without sine/cosine evaluations, the advertised peak performance on Knl (shown with the dotted line) can not be achieved in practice, as the clock speed is reduced when AVX-512 instructions are executed.

Both the gridder and the degridder kernels on Haswell and Knl perform much lower than the theoretical peak. However, given the limitations of hardware *and* the supporting mathematical library, the kernels on Haswell perform close to optimal. The performance of Knl is slightly further from the peak, for which we have two explanations: there is some load-imbalance (due to a varying number of visibilities per subgrid), and by the execution of (partially) masked vector instructions. Furthermore, since Knl is compute-bound, we found no advantage of using high-bandwidth MCDRAM over DRAM.

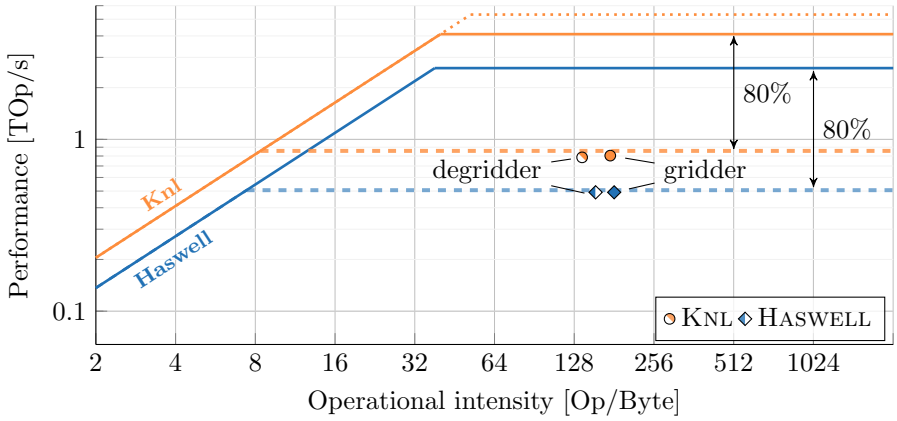



***Fig. 5.4:*** *Roofline analysis: one operation is* $+, -, *, \sin()$ *or* $\cos()$*. Peak performance is only achieved if non-masked FMA instructions (two operations) are used exclusively. While the operation count is known exactly, the data movement is measured.*

### 5.4.3 Throughput and energy efficiency

We call the combined throughput for gridding and degridding the *imaging throughput*. This metric provides insights into the processing rate for one imaging cycle and we show results in Fig 5.5. The throughput numbers are computed by dividing the number of visibilities processed (which is a known quantity) by the runtime.

As described in Chapter 3, we use LIKWID [47] to measure the energy consumption of Haswell and Knl. To this end, LIKWID performs measurements of CPU package and DRAM energy consumption in software, by querying performance counters. We sum the two measurements to get the total energy consumption of the platform. We did not use PowerSensor, because it would also include auxiliary energy consumption (since PowerSensor would need to be connected to the motherboard power cables), not just the energy consumption of the CPU package and the DRAM. We plot the energy efficiency in terms of visibilities processed for every Joule consumed in Fig. 5.6.

By having a higher peak performance (see Fig. 5.2) Knl achieves a higher imaging throughput than Haswell. Furthermore, Knl is almost 50% more energy-efficient.

### 5.4.4 Scalability

We evaluate gridding scalability in Fig. 5.7. As this graph illustrates, throughput scales linearly with the number of cores used, for both Haswell and Knl. While Haswell has better per-core performance, the higher core count of Knl leads to

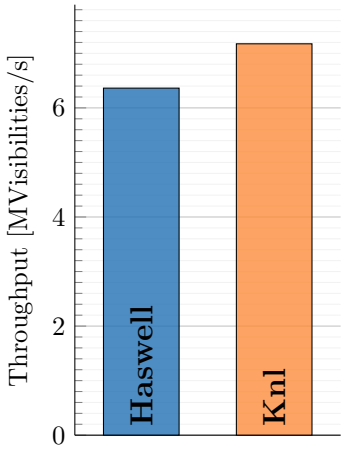



***Fig. 5.5:*** *Imaging throughput*

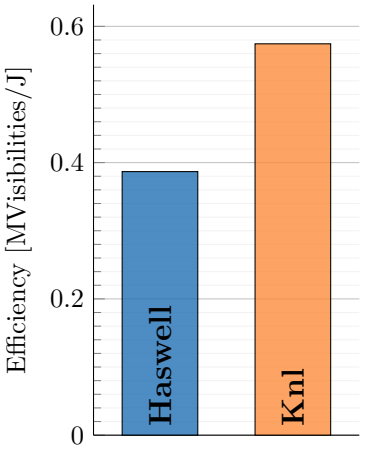



***Fig. 5.6:*** *Energy efficiency*



higher throughput when all cores are 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.

## 5.5 Conclusion

We discussed in Section 2.3 that the performance of processors has increased significantly over the years, mostly due to higher clock frequencies, increasing core-counts and more advanced instruction sets. We had to take the clock frequency as a given as it is dynamically adjusted based on the workload (e.g. the number of CPU cores used and the type of instructions executed).

The execution plan allows tuning of the amount of work per subgrid, such that the workload is properly distributed over across all available processor cores. Finally, we vectorized the most critical parts of the compute kernels to maximize the number of operations performed per clock cycle. Still, the achieved performance on these devices fell short of our expectations. We identified that this is caused by the evaluation of sine/cosine, which takes about 80% of the total kernel runtime.

Despite architectural differences, we found that optimizations on a regular Intel (Haswell) CPU and Intel Xeon Phi (Knights Landing) many-core processor were mostly identical. In both cases, the evaluation of sine/cosine in software is the most important performance limiter. We tried to use lookup tables to work around

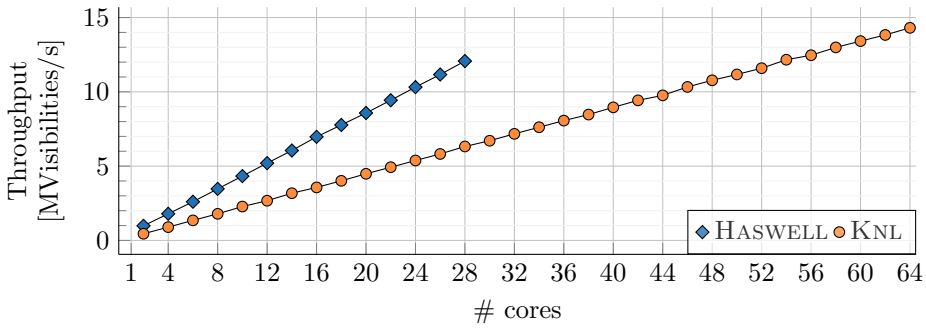



***Fig. 5.7:*** *Subgrids are distributed over logical cores and are processed in parallel. Consequently, throughput scales linearly with the number of physical cores used.*

this bottleneck, but found that the proprietary MKL library provides the best performance.

We showed that the higher peak performance of the Xeon Phi compared to the regular Xeon indeed translates to higher throughput. Furthermore, we showed that the Xeon Phi many-core processor is more energy-efficient than the Xeon processor.

The clock frequencies of contemporary CPUs have hit a plateau and they can not even be attained while executing the most advanced instruction set. The current trend seems to lean towards ever-increasing core counts. However, for IDG, this will not resolve the issue of sub-par sine/cosine performance. We will need accelerators capable of evaluating sine/cosine more efficiently to speed up IDG and bring radio-astronomical imaging to the exascale era.

**RQ3a:** *How efficient is IDG on CPUs?*

We demonstrated that our IDG implementation for CPUs is scalable to many CPU cores and to wide vector units, but also that attainable performance is fundamentally limited by the ability of the CPU to perform sine/cosine operations. Therefore, we conclude that on CPUs, IDG is not very efficient due to low performance of sine/cosine operations in software.

Chapter 6

# IDG on GPUs

The contents of this chapter are based on the following papers:

**Image-Domain Gridding on Graphics Processors**
**Veenboer, B.**, Petschow, M., Romein, J. W.
in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), pages 545–554, IEEE, 2017*

**Radio-Astronomical Imaging on Graphics Processors**
**Veenboer, B.** Romein, J. W.
In *Astronomy & Computing, Elsevier, Volume 32, July 2020*

In this chapter we answer the following research sub question:

**RQ3b:** *How efficient is IDG on GPUs?*

## 6.1 Introduction

In this chapter, we will be looking at Graphics Processors (GPUs) as candidate accelerators for Image-Domain Gridding. These devices have a significantly higher peak performance compared to CPUs (see also Section 2.3) but have a different programming model. After providing some background on GPUs in Section 6.2, we discuss our GPU implementations of IDG in Section 6.3. In Section 6.4 we present results and perform a performance and energy efficiency analysis. We demonstrated the first implementation of IDG on GPUs in [11]. We presented an extended subset of this work in [12] where we demonstrate new features such as support for some specific imaging use cases and additional performance optimizations. The contents of Section 6.3 and 6.4 are therefore mostly based on [12].

## 6.2 Background



Back in the 1950s, the first 'video cards' were being produced, with the primary goal of processing a stream of data from a central processor into images rendered on a display. Over the course of decades, these devices have evolved into increasingly more powerful devices. In 1999 NVIDIA introduced the first 'graphics processing unit' (GPU), a single-chip processor with integrated engines for 3D rendering. In the early 2000s both NVIDIA and another GPU manufacturer, ATI (which would later become a part of AMD), were competing to build increasingly more powerful GPUs.

At this point, people started using GPUs for tasks other than graphics because they were much faster than 'traditional' CPUs. At first, people used graphics programming languages such as OpenGL to use GPUs for general-purpose computations (GPGPU). While the first generations of GPUs were purely targeted at graphics computation, newer generations of GPU architectures added functionalities such as conditional branching (for instance to implement loops) and support for random-access memory writes. In 2007, NVIDIA introduced the Compute Unified Device Architecture (CUDA) development environment [2]. CUDA made GPUs much easier to program and became a widely adopted programming model for GPU computing.

Two years later, OpenCL was introduced as a framework that allows for the development of code for both CPUs and GPUs [3]. Today, NVIDIA GPUs are typically programmed using CUDA while AMD GPUs are programmed in OpenCL. Apart from syntactic differences and differences in terminology, these programming

languages offer the same basic functionality. In the remainder of this chapter, we adhere to the CUDA terminology.

GPUs have several of *Streaming Multiprocessors* (SMs) with a number of *CUDA cores* each. Every SM contains a register file, load-store units, dedicated caches, and a software-managed cache (shared memory).

On a 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. A thread block is executed in the form of *warps*. A warp behaves similar to a vector unit in a CPU: a single instruction is executed for multiple (distinct) data elements (in SIMD fashion). 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. See also Figure 6.1 for a schematic overview of the GPU architecture.

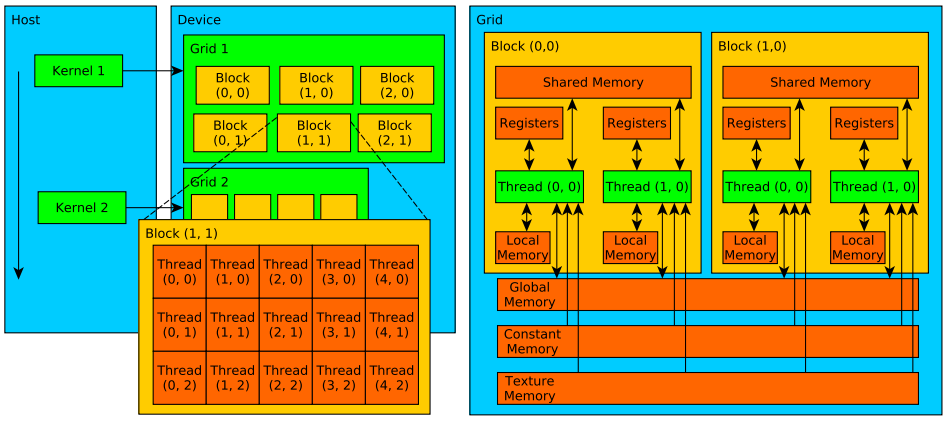



***Fig. 6.1:*** *A schematic overview of the architecture of a GPU using CUDA terminology [2]. The left part of the diagram illustrates the host (the CPU), which executes kernels on the device (the GPU). Kernels are executed by multiple threads organized in three-dimensional blocks, which can be organized in three-dimensional grids (middle part of the figure). The user explicitly defines the dimensions of the blocks and the grid. The GPU has different types of memory: registers and local memory private to each thread, memory shared by threads in a block, and several types of global memory. Shared memory has low latency and high bandwidth. Texture and constant memory feature a cache as well and are typically used for read-only data. Global memory can be read and written by the host and has the highest capacity of all these types of memory.*

## 6.3 Implementation

We detail our parallelization and optimization strategies for various parts of IDG in the following sections.

### 6.3.1 Gridder kernel

We implemented a GPU gridder kernel by applying a number of code transformations to the IDG gridder algorithm in Algorithm 1. First, subgrids are grouped into batches and the gridder kernel is launched once for every batch of subgrids. The GPU schedules these batches in the form of thread blocks onto the SMs such that multiple, independent subgrids are computed in parallel, see the pseudo-code in Algorithm 6.

The computation of *phase* (Line 10) is implemented using a fused multiply-add (FMA) operation of which the operands *index* and *offset* are computed outside the critical path and *scale* is cached in shared memory. We use native sine/cosine instructions for the computation of the *phasor* (Line 11), see Section 6.3.3 for details.



Next, in Algorithm 7, we show how we applied the *loop strip-mining* [63] technique to split loops into two parts: a fixed-size inner loop and a variable-size outer loop. We then moved the outer loop over frequencies (Line 10) outside to create a critical path without conditional branches, since the inner loop over frequency channels has a fixed number of iterations (Line 11), and the loop over polarizations (Line 15) is unrolled. The compiler consequently generates better code that runs more efficiently.

Tuning parameters $\kappa_t$ and $\kappa_c$ are chosen such that a batch of visibilities can be cached in shared memory. To this end, we add a shared-memory buffer of size $\kappa_t \times \kappa_c$ visibilities and let threads collaboratively load visibilities from device memory in a coalesced manner. We also apply *thread coarsening* [64] by partially unrolling the inner loop over pixels (Line 5) such that a visibility loaded from shared memory is used to update $\kappa_i$ pixels. These pixels are stored in registers and are only written to device memory after all visibilities (for the current $\kappa_c$ channels) are processed. The resulting code is shown in Algorithm 8.

In this kernel $\kappa_{threads}$, $\kappa_i$ and $\kappa_t$ are tuning parameters. We set their values empirically, e.g. $\kappa_{threads} = 128$, $\kappa_i = 4$ and $\kappa_t = 256$. A future implementation could use an autotuner such as Kernel Tuner [65] to find an optimal combination of parameters automatically.

```
1  s ← get_thread_block_id() ;                                   // s ∈ 1 to S
2  tid ← get_thread_id() ;                                       // tid ∈ 1 to κ_threads
3  complex<float> subgrid[P][N̄×N̄];
4  for i ← tid to N̄×N̄ by κ_threads do                            // map threads to pixels
5      float offset = compute_offset(s, i);
6      for t ← 1 to T̄ by 1 do
7          float index = compute_index(s, i, t);
8          for c ← 1 to C̄ by 1 do
9              float scale = load_scale(c);
10             float phase = offset - (index * scale);
11             complex<float> phasor = cos(phase) + i sin(phase);
12             for p ← 1 to P by 1 do                            // P=4, fully unrolled
13                 complex<float> visibility = load_visibility(s, t, c, p);
14                 subgrid[p][i] += phasor * visibility;
15             end
16         end
17     end
18 end
19 apply_aterm(subgrid);
20 apply_taper(subgrid);
21 apply_ifft(subgrid);
```

***Algorithm 6:*** *This pseudo-code shows the first step in the GPU gridder kernel implementation: subgrids $s$ are mapped to thread blocks (Line 1), with $\kappa_{threads}$ threads each. For every thread block, threads are mapped to pixels of a subgrid using their thread identifier (Line 2) such that multiple pixels are computed in parallel (Line 4).*

Applying the A-term and tapering function (Line 30 and Line 31 in Algorithm 8) takes place outside of the critical path, after which the pixels are written to device memory. The subgrid FFT is performed seperately, see Section 6.3.4.

### 6.3.2 Degridder kernel

We implemented the GPU degridder kernel such that subgrids are mapped onto thread blocks and visibilities onto threads. Shared memory is used as a cache for input data (batches of subgrid pixels) and for pre-computed data (*offset*). Pseudocode for this kernel is provided in Algorithm 9. This batched execution strategy can be configured by setting the size of the thread block ($\kappa_{threads}$), the number of pixels processed per iteration ($\kappa_i$) and the number of frequency channels computed per thread ($\kappa_c$). When $\bar{T}$ is small (e.g. less than twice the size of a warp), the kernel

```
1  s ← get_thread_block_id() ;                                   // s ∈ 1 to S
2  tid ← get_thread_id() ;                                       // tid ∈ 1 to κ_threads
3  complex<float> subgrid[P][N̄×N̄];
4  for i_1 ← tid to N̄×N̄ by κ_threads×κ_i do                       // map threads to pixels
5      for i_2 ← i_1 to κ_i by 1 do                               // pixel batch
6          float offset = compute_offset(s, i);
7          for t_1 ← 1 to T̄ by κ_t do
8              for t_2 ← t_1 to κ_t by 1 do                       // time batch
9                  float index = compute_index();
10                 for c_1 ← 1 to C̄ by κ_c do
11                     for c_2 ← c_1 to κ_c by 1 do               // channel batch
12                         float scale = load_scale(c);
13                         float phase = offset - (index * scale);
14                         complex<float> phasor = cos(phase) + i sin(phase);
15                         for p ← 1 to P do                      // P=4, fully unrolled
16                             complex<float> visibility = load_visibility(s, t_2, c_2, p);
17                             subgrid[p][i_1] += phasor * visibility;
18                         end
19                     end
20                 end
21             end
22         end
23     end
24 end
25 apply_aterm(subgrid);
26 apply_taper(subgrid);
27 apply_ifft(subgrid);
```

***Algorithm 7:*** *This GPU gridder improves on Algorithm 6 by additionally applying loop strip-mining to create fixed-size loops over pixels (Line 5), time (Line 8), and frequency channel (Line 11). This technique enables the compiler to generate better code that runs faster.*

```
1  s ← get_thread_block_id() ;                                   // s ∈1 to S
2  tid ← get_thread_id() ;                                       // tid ∈ 1 to κ_threads
3  complex<float> subgrid[P][N̄×N̄] = 0 ;                          // shared memory
4  complex<float> visibilities[κ_t][κ_c][P] ;                    // shared memory
5  for c_1 ← 1 to C̄ by κ_c do
6      for i_1 = tid to N̄×N̄ by κ_threads × κ_i do
7          complex<float> pixel[P][κ_i] = 0;
8          for t_1 ← 1 to T̄ by κ_t do
9              for j ← tid to κ_t×κ_c× P by κ_threads do
10                 visibilities[..][..][..] = load_visibility(..);   // from device memory
11             end
12             for t_2 ← 1 to κ_t by1 do
13                 float index = compute_index(s, i, t);
14                 for c_2 ← c_1 to κ_cby1 do                        // channel batch
15                     float scale = load_scale(c_1 + c_2);
16                     float phase = offset - (index * scale);
17                     complex<float> phasor = cos(phase) + i sin(phase);
18                     for i_2 ← 1 to κ_i by 1 do                    // pixel batch
19                         for p ← 1 to P by 1 do                    // P=4, fully unrolled
20                             complex<float> visibility = visibilities[t_2][c_2][p];
21                             pixel[p][i_2] += phasor * visibility;
22                         end
23                     end
24                 end
25             end
26         end
27         subgrid[..][..] += pixel[..][..];
28     end
29 end
30 apply_aterm(subgrid) ;                                        // outside of the critical path
31 apply_taper(subgrid);
32 apply_ifft(subgrid);
```

***Algorithm 8:*** *In this GPU gridder, we added pre-fetching of visibility data (Line 10 and 20). We have now found an elegant mapping from the IDG algorithm to the GPU: multiple subgrids are computed in parallel (by distinct thread blocks), multiple pixels of the subgrid are computed in parallel (by threads within a thread block), while input data is pre-fetched into shared memory buffers to maximize reuse, and memory accesses are coalesced.*

automatically uses $\kappa_c = 1$ to maximize the number of visibilities that are processed in parallel. To this end, in Line 9, threads are additionally mapped to frequency channels.

### 6.3.3 Sine/cosine

On NVIDIA GPUs, the SMs contain a number of special-function units (SFUs) that implement the computation of both transcendental functions and interpolation in hardware [66]. The SFU provides fast approximations for sine/cosine with a maximum error of 2 units of least precision (the spacing between two consecutive floating-point numbers) [67]. The SFU operates at half the rate of an FMA unit, see also Fig. 6.2. On AMD GPUs, sine and cosine are computed on the same execution units that also perform FMA instructions. Whereas an FMA instruction has a throughput of 1 cycle per instruction, sine/cosine instructions are performed at a quarterly rate. The maximum error is implementation-defined [68, 69]. Both NVIDIA's and AMD's native instructions provide sufficient accuracy for IDG.

We measure the performance of the sine/cosine evaluation (*cis*) for different ratios of sine/cosine evaluations and FMAs and show results for a NVIDIA Pascal GPU in Fig. 6.3 and for an AMD Vega GPU in Fig. 6.4 (See Section 6.4.1 for details about these GPUs). The NVIDIA GPU achieves superior performance by executing FMAs and sine/cosine evaluations using separate processing units (the SFUs).



### 6.3.4 Subgrid FFTs

Small subgrids (up to 32×32 pixels) could fit in shared memory and it would therefore be most efficient to compute the Fourier transformation before the data is written to global memory. Unfortunately, the cuFFT and clFFT libraries assume that the input data resides in global memory and processes 2D FFTs in two separate passes for rows and columns respectively, while intermediate results are also stored in global memory. We implemented a custom 32×32 FFT kernel that keeps intermediate data in shared memory. While this method improves performance at least twofold, we would need to implement a unique kernel for every possible different size of the subgrid. Moreover, as we show in Section 6.4, the impact of the time spent in the subgrid FFT with respect to the overall computation time is relatively low for the imaging use cases that we consider here. NVIDIA has demonstrated the cuFFTDx library, which is a device-callable library that retains and reuses data on-chip [71]. Like our custom FFT, cuFFTDx allows inlining the FFT into the gridder or degridder kernel. Since

```
1  apply_ifft(subgrid);
2  apply_taper(subgrid);
3  apply_aterm(subgrid);
4  s ← get_thread_block_id() ;                                   // s ∈ 1 to S
5  tid ← get_thread_id() ;                                       // tid ∈ 1 to κ_threads
6  complex<float> pixels[P][κ_i × κ_i] ;                         // shared memory
7  float offsets[κ_i] ;                                          // shared memory
8  for c_1 ← 1 to C̄ by κ_c do
9      for t ← tid to T̄ by κ_threads do                          // map threads to visibilities
10         complex<float> visibility[P][κ_c];
11         for i_1 ← 1 to N̄ × N̄ by κ_i do
12             for i_2 ← (i_1 + tid) to κ_i by κ_threads do      // map threads to pixels
13                 offsets[i_2] = compute_offset();
14                 pixels[i_2] = load_pixel() ; // from device memory to shared memory
15             end
16             for i_2 ← 1 to κ_i by 1 do                        // pixel batch
17                 float offset = offsets[i_2] ;                 // from shared memory
18                 complex<float> pixel = pixels[p][i_2] ;       // from shared memory
19                 for c_2 ← c_1 to κ_c by 1 do                  // channel batch
20                     float index = compute_index(s, i, t);
21                     float scale = load_scale(c);
22                     float phase = (index * scale) - offset;
23                     complex<float> phasor = cos(phase) + i sin(phase);
24                     for p ← 1 to P do                         // P=4, fully unrolled
25                         visibility[p][c_2] += pixel * phasor;
26                     end
27                 end
28             end
29         end
30     end
31     store(visibility);
32 end
```

***Algorithm 9:*** *We applied the same techniques as we detailed in Section 6.3.1 to implement the GPU degridder kernel: subgrids are mapped to thread blocks and we split, reorder and unroll certain loops to create an efficient mapping of threads to visibilities and pixels. The computation takes place in two stages: first threads are mapped to visibilities, such that $\kappa_{threads}$ each compute $\kappa_c$ visibilities each. Next the subgrid is processed in batches of $\kappa_i$ pixels, which are prefetched in a shared memory cache. As* offset *is not time or frequency dependent,* offset *is precomputed and stored in shared memory.*

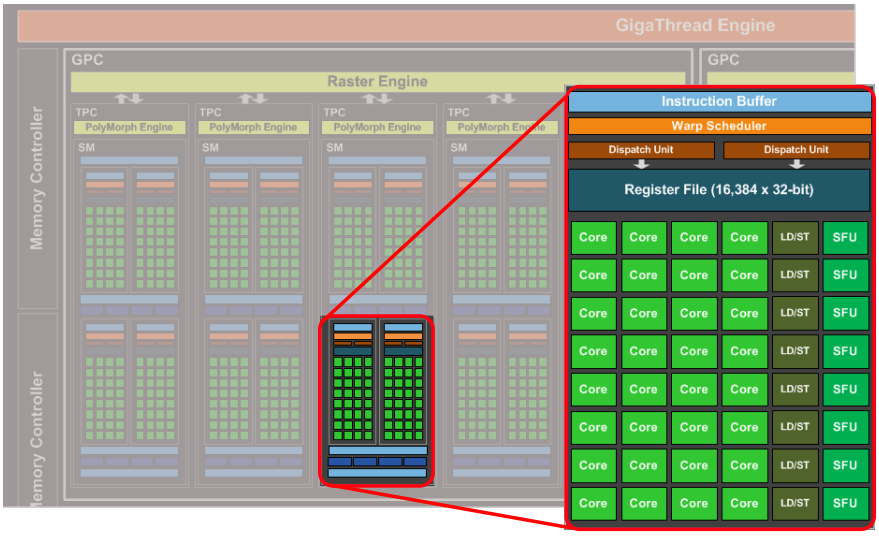



***Fig. 6.2:*** *This diagram of NVIDIA's GP104 chip illustrates that each Streaming Multiprocessors (SM) has a number of Special Function Units (SFUs) next to cores [70]. These units can be used to evaluate transcendental function (like sine/cosine) while the cores perform 'normal' floating-point operations such as multiplications and additions.*

6

cuFFTDx is not yet available and the custom FFT kernel is not practical in general, we use the cuFFT library for all our measurements. As we will show later, this is an acceptable approach for common imaging use cases where the FFT runtime has only a modest impact on the overall runtime.

### 6.3.5 Adder and splitter

We use a simple adder kernel where atomic adds are used to perform the addition of subgrids onto the grid. Unlike the adder kernel, no atomic operations are required for the splitter kernel. For every subgrid, the relevant part of the grid is copied into the subgrid buffer.

### 6.3.6 Asynchronous I/O and kernel execution

We use double-buffering such that the GPU can continue to execute kernels during data transfers. To this end, we use multiple worker threads on the host that process subsets of the work by issuing operations onto CUDA *streams*. CUDA *events* are used to synchronize between streams and to ensure sequential consistency between

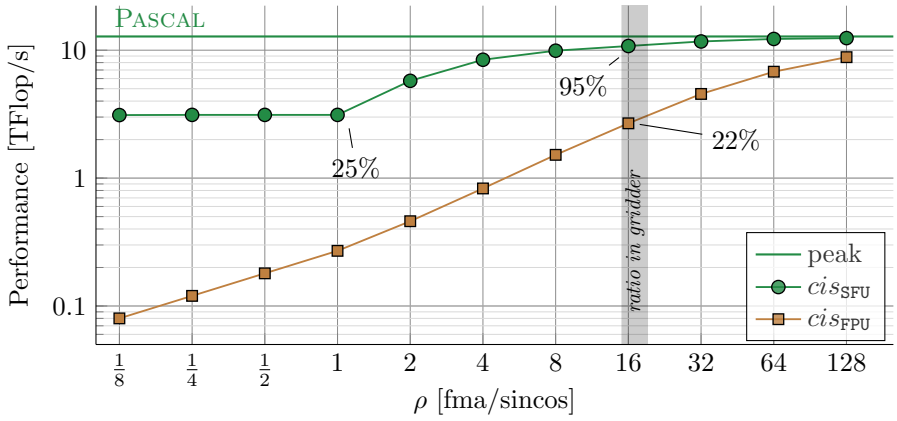



***Fig. 6.3:*** *This plot shows the maximum attainable performance on an NVIDIA GPU of a workload comprised of FMA instructions and sine/cosine evaluations. For $\rho \leq 2$, the performance is a quarter of the theoretical single-precision floating-point performance. This GPU has one SFU for every four FPUs. We can also identify this in the plot, as the achieved performance is 25% of the theoretical floating-point peak performance for $\rho \leq 1$. For an instruction mix of 17 FMAs and one sine/cosine evaluation ($\rho = 17$) as is the case in the gridder and degridder kernels, the operations on the SFUs are almost completely overlapped with computations on the floating-point units (FPUs). When sine/cosine is evaluated using FPUs, overall performance is significantly lower at about 22% of the peak.*

the operations within the kernel invocation; i.e., the kernel only starts when the input data is transferred. This technique allows I/O and kernel execution to overlap, as illustrated in Fig. 6.5. We use paged-locked host memory to enable DMA memory transfers between host and device memory. This increases the PCIe transfer speed while CPU overhead is reduced.

In our initial IDG paper ([11]) we presented benchmarks with small subgrids ($\bar{N} = 24$). In this setting, the balance between I/O (to transfer visibilities) and computations (to compute pixels) leaned toward the I/O side. In that scenario, throughput was limited by the bandwidth of the PCIe bus. This limit can be alleviated by using a faster interconnect, such as NVLink. However, as IDG matured we concluded that larger subgrids (e.g. $\bar{N} = 32$) are more common in practice. While this doubles the number of operations performed for every visibility, it also increases the maximum supported kernel size and typically leads to a better visibility density as more visibilities can be covered by a single subgrid (see also Section 4.5. Therefore, in [15] we moved to $\bar{N} = 32$. Using this setting, the PCIe bandwidth is only a

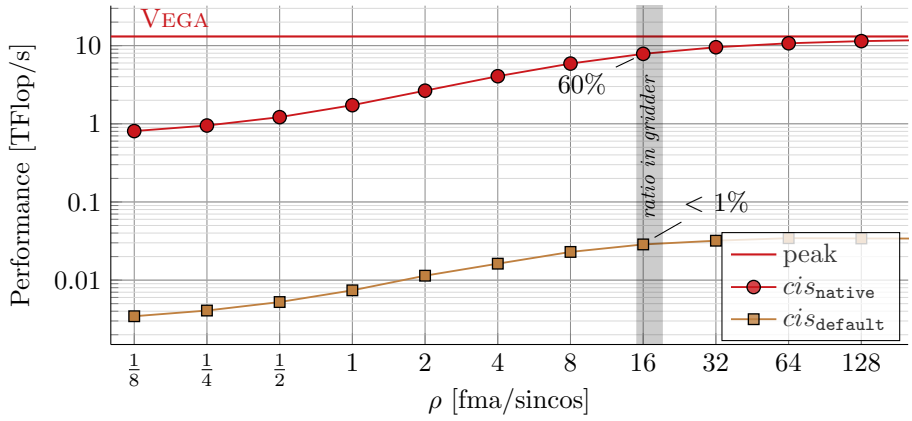



***Fig. 6.4:*** *The default sine/cosine implementation on AMDs Vega GPU architecture is very slow: less than 1% of the floating-point peak performance is achieved for* $\rho = 17$*. Native instructions are a much faster alternative.*

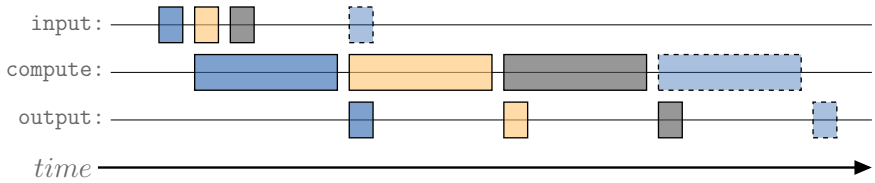



***Fig. 6.5:*** *In this double-buffering example, three threads (indicated with blue, yellow, and gray) each offload their computation to the GPU. This technique overlaps I/O with computations and helps to minimize GPU idle time.*

bottleneck for the fastest GPUs available today (e.g. NVIDIA Tesla V100 or NVIDIA Titan RTX).

### 6.3.7 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.

NVIDIA 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. 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. 6.6.

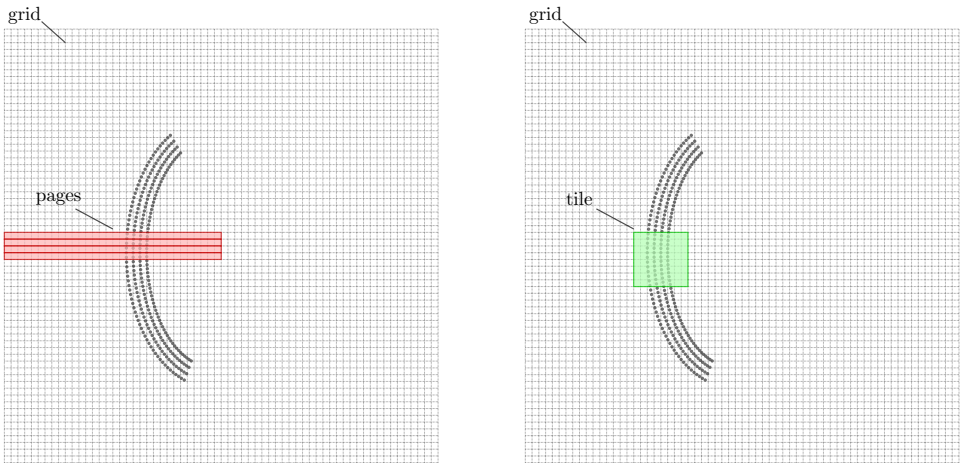


## 6.4 Results

We first provide the experimental setup in Section 6.4.1. In Section 6.4.2 we analyze the performance of our IDG implementations on an NVIDIA GPU and on an AMD GPU. We compare these GPUs in terms of throughput and energy efficiency in Section 6.4.3. In Section 6.4.4 and 6.4.5, we take a closer look at throughput for two imaging use cases; large images and spectral-line imaging.

### 6.4.1 Experimental setup

We use the same dataset and parameters as described in Section 5.4.1 to compare two GPUs: Vega (an AMD Vega Frontier Edition GPU) and Pascal (an NVIDIA Tesla P100 GPU), see Table 6.7) for details. Vega is hosted by a dual-socket Haswell-EP system, where the GPU is connected using PCIe 3.0. This is the same system as described in Section 5.4.1. Pascal is hosted by a Minksy system that is part of the JURON cluster [72]. A Minsky system is based on the Power 8 architecture




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

***Fig. 6.6:*** *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 migrated pixels are not updated. By tiling the grid, pixels that are close together in the grid are also close together in memory. Therefore, the number of page migrations required for accessing neighboring pixels (e.g. in a tile, indicated in green), is significantly lower than in the default setting.*



and supports the high-bandwidth NVLink 1.0 bus, which provides roughly three times more bandwidth than PCIe 3.0. 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.

### 6.4.2 Performance

In Figure 6.8 we show the execution time of a single imaging cycle. Like on the CPU, imaging runtime is dominated by the gridder and the degridder kernels. We analyze the performance of these kernels in Fig. 6.9. The solid lines correspond to the peak FMA performance and peak memory bandwidth as advertised by the manufacturers; the dashed lines correspond to the performance for the instruction mix of 1 sine/cosine evaluation and 17 FMAs operations using native math instructions, see Fig. 6.3 and 6.4. On Vega the performance is bound by the sine/cosine evaluations,

***Fig. 6.7:*** *The NVIDIA Pascal and AMD Vega GPUs used in our experiments.*

| Name | Architecture | Peak (TFlop/s) | Mem size (GB) | Mem bw (GB/s) | TDP (W) |
|---|---|---|---|---|---|
| PASCAL | Pascal | 10.6 | 16 | 732 | 250 |
| VEGA | Vega | 13.1 | 16 | 483 | 300 |

while on PASCAL the performance is fairly close to the theoretical peak performance of the floating-point units. 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. These results indicate that PASCAL is a very suitable architecture for IDG as it offers a balanced mix of floating-point units, special-function units, and shared memory.

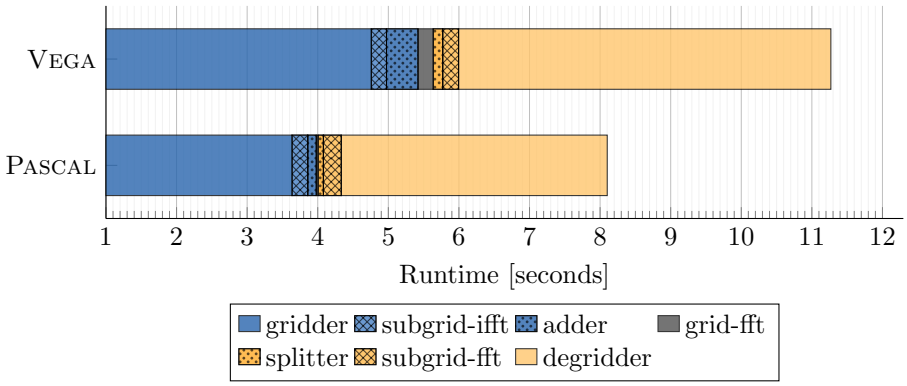



***Fig. 6.8:*** *Distribution of runtime for all kernels in an imaging cycle.*

### 6.4.3 Throughput and energy efficiency

We present throughput and energy efficiency results for PASCAL and VEGA respectively in Fig. 6.10 and in Fig. 6.11. For VEGA, we measure the energy consumption of the full PCIe device, using PowerSensor [48] (see also Chapter 3). PASCAL uses a mezzanine connector unsuitable for PowerSensor and we, therefore, resort to the NVIDIA Management Library (NVML) to measure energy consumption. By achieving better performance in the gridder and degridder kernels, PASCAL naturally also outperforms VEGA in terms of throughput. Furthermore, PASCAL is significantly more energy-efficient than VEGA.

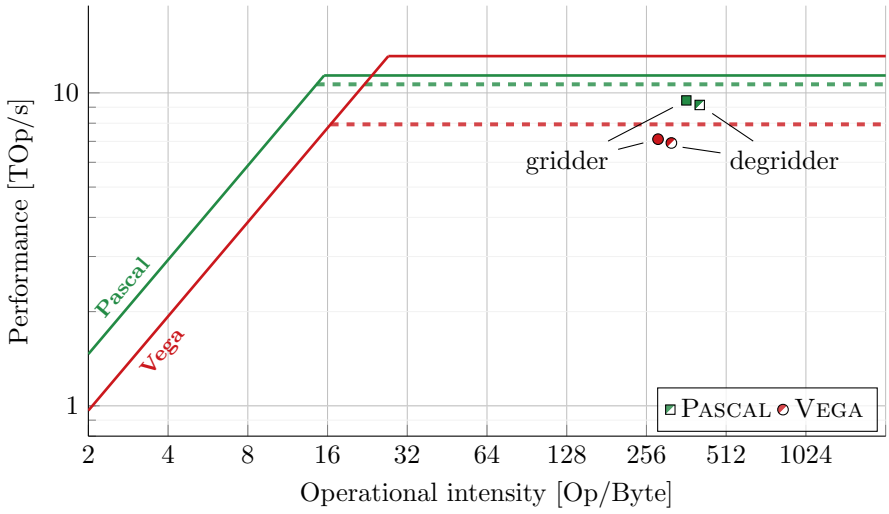



***Fig. 6.9:*** *Roofline analysis for* Pascal *and* Vega.

### 6.4.4 Creating large images

We show the imaging throughput for our Hybrid and Unified imagers on Pascal in Fig. 6.12, and compare them to GPU-only imaging. While the Unified imager achieves almost identical throughput to the GPU-only imager, it also allows for larger images at a modest throughput decline.

Up to grid sizes of about 16,000×16,000 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 overall imaging throughput of 116 MVisibilities/s. For the largest images that the GPU-only 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. 6.13a: in all cases the computation on the CPU takes longer than the computation on the GPU.

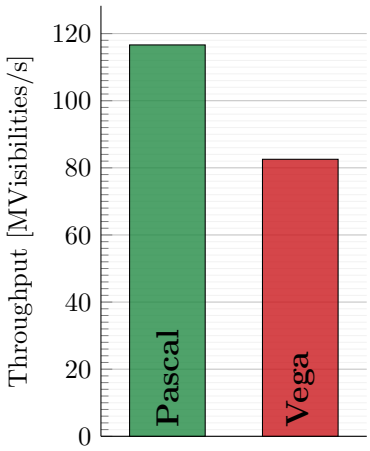



***Fig. 6.10:*** *Imaging throughput*

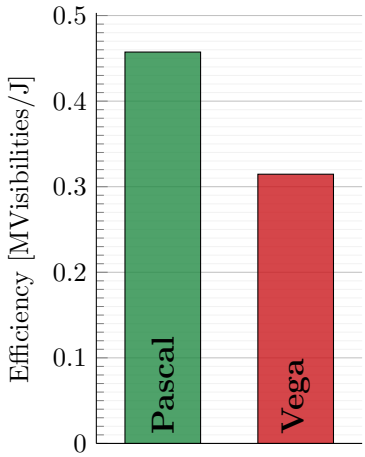



***Fig. 6.11:*** *Energy efficiency*

There is a minor performance difference between the GPU-only and the unified imaging routines for grid sizes up to 16,000×16,000 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. 6.13b.

### 6.4.5 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. 6.14a. This is an appealing property, for instance for spectral-line imaging, where $\bar{C} = 1$. As shown in Fig. 6.14b, throughput is affected for small values of $\bar{C}$. As we showed in the complexity analysis of IDG in Section 4.5, 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 the adder kernel (for gridding) or splitter kernel (for degridding).

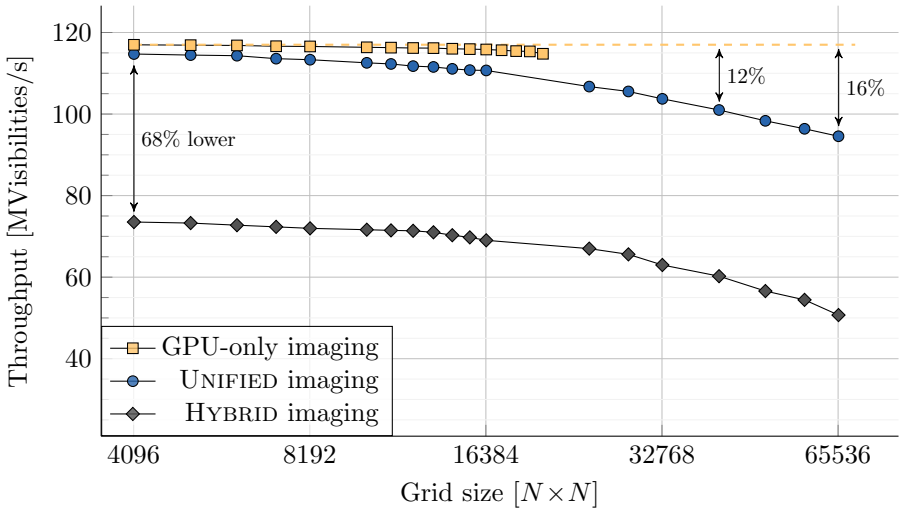


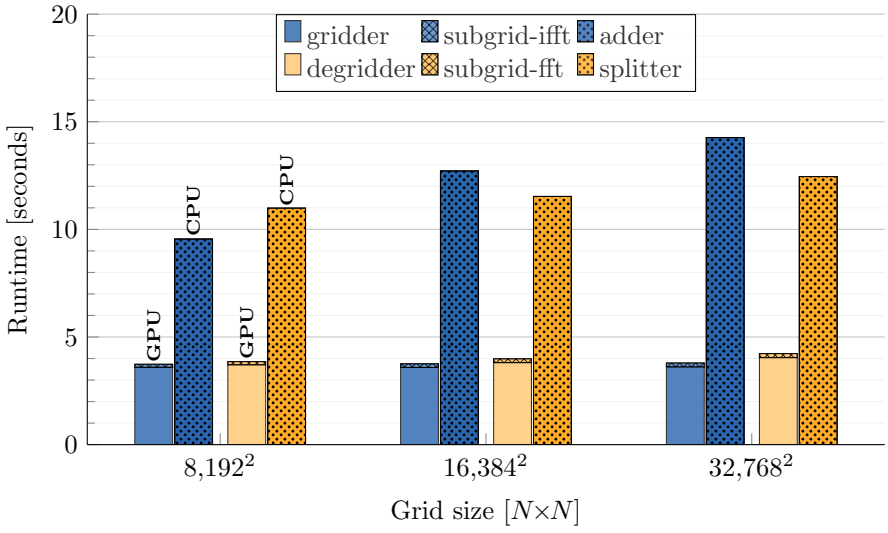

***Fig. 6.12:*** *Throughput for the* HYBRID *and* UNIFIED *imagers.*



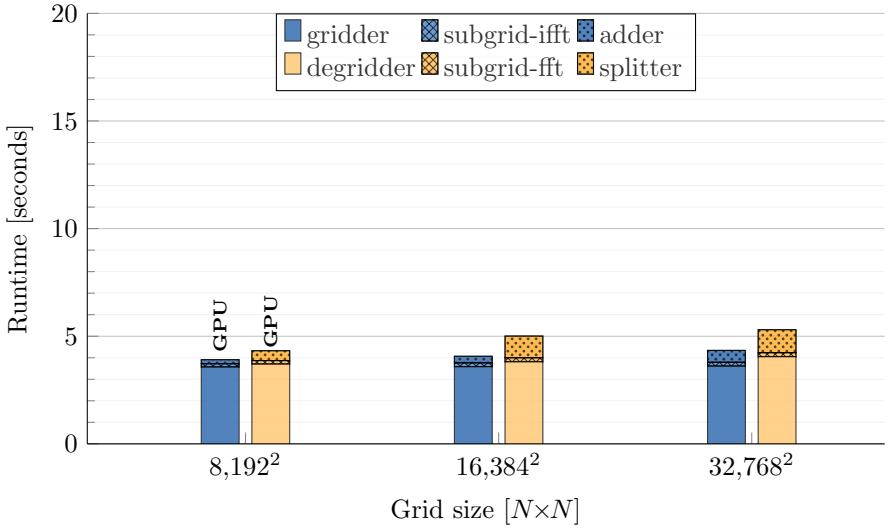
With a fixed setting of $\bar{N}$ and $C_{obs} > \bar{C}$, not all $C_{obs}$ channels might fit on a single subgrid. In this case, IDG will create multiple subgrids with at most $\bar{C}$ channels each to cover all $C_{obs}$ channels. The throughput for processing of $C_{obs}$ channels will therefore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

Typical datasets have $C_{obs} \gg 16$, split into subbands (e.g with $C_{subband} = 256$) that are gridded separately. For such datasets, and reasonably sized subgrids (e.g. $\bar{N} = 32$ up to $\bar{N} = 48$), not all visibilities can be mapped to a single subgrid. To solve this, we use *channel groups*, which are subsets of visibilities with neighboring frequency channels. (See also Section 4.3.)

Section 4.3. A channel group can have a predetermined maximum number of frequency channels, e.g. to match a $\bar{C}$ value that achieves good performance on a given platform.

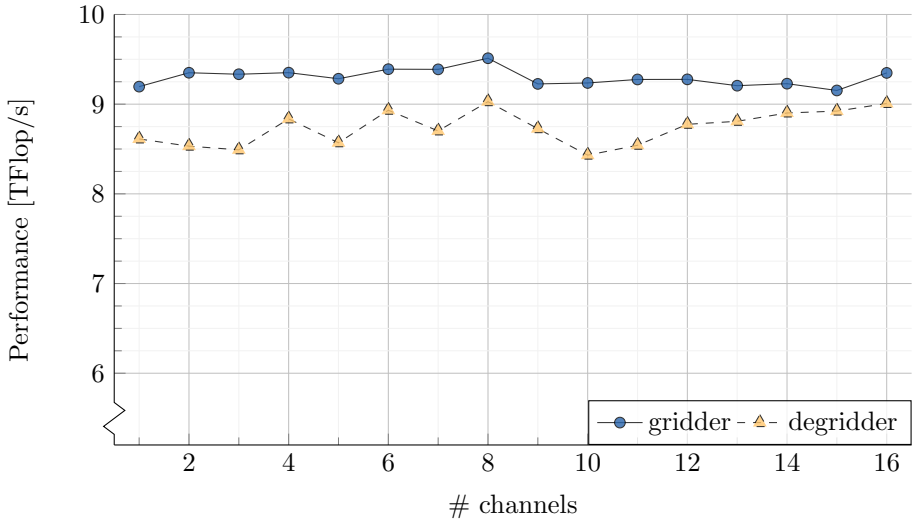



**(a)** Hybrid runtime distribution

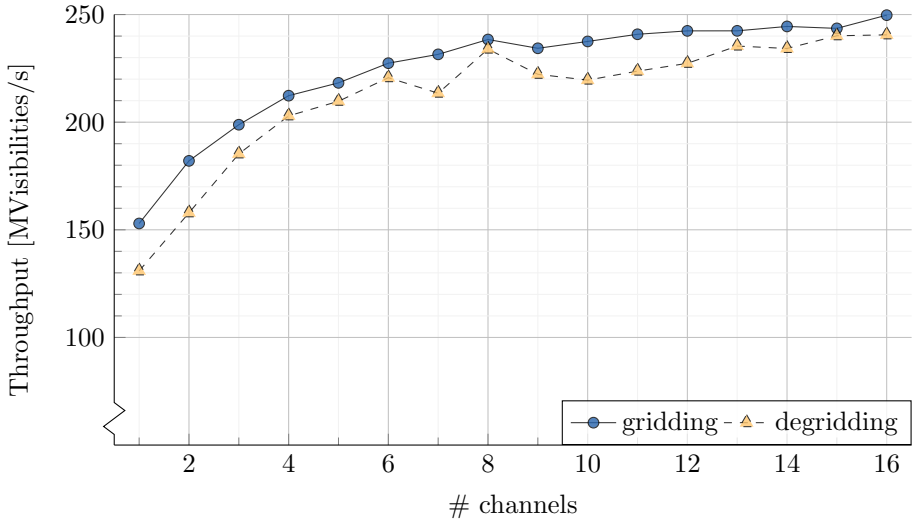



**(b)** Unified runtime distribution

***Fig. 6.13:*** *In* Hybrid *gridding (Fig. 6.13a), 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. 6.13b), 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

***Fig. 6.14:*** *The gridder and degridder performance is relatively consistent for all values of $\bar{C}$. For small values of $\bar{C}$, the time spent in the other kernels (e.g. the Fourier transform of the subgrid and the adder kernel for gridding, not shown in this graph) negatively affects throughput.*

By splitting large datasets into subbands, and by processing subbands in channel groups, the gridding process is similar for every subset of visibilities. Thus for large datasets, the same gridding processes is simply repeated many times, on different data. We, therefore, argue that our results (measured on relatively small datasets) can be extrapolated to the larger datasets that are used in practice.

## 6.5 Conclusion

The Image-Domain Gridding algorithm efficiently maps onto GPUs. Like we have shown for CPUs in Chapter 5, having proper support for the evaluation of sine/cosine is key to achieving high performance for GPUs as well.

By leveraging special-function units, we have demonstrated that NVIDIA GPUs achieve close to the theoretical floating-point unit peak performance. On AMD GPUs, the sine/cosine evaluations are performed on the same execution units that also execute floating-point operations such as multiplications and additions, albeit at a lower rate. Thus like on CPUs, the sine/cosine operations on AMD GPUs compete for resources.

We demonstrated that an NVIDIA (Pascal) GPU is faster than an AMD (Vega) GPU, while it is also more energy-efficient. We have also demonstrated that our IDG implementation for NVIDIA GPUs elegantly uses CUDA Unified Memory to create very large sky images. Finally, we have shown that IDG can also handle specific imaging use cases such as spectral-line imaging.

NVIDIA GPUs, while being general-purpose accelerators, almost seem tailored for IDG, as they provide a balanced mix of floating-point units, special-function units, shared memory, and other resources. Still, we had to carefully optimize the GPU kernels to get close to the theoretical peak performance. Moreover, having a fast interconnect (such as NVLink) and CUDA Unified Memory makes an NVIDIA GPU a very suitable IDG accelerator even in cases where the sky image does not fit in GPU memory.

**RQ3b:** *How efficient is IDG on GPUs?*

IDG runs highly efficient on GPUs. Especially, on NVIDIA GPUs, with support for sine/cosine in dedicated hardware, performance is excellent as the most dominant IDG kernels (the gridder and degridder kernel) approach theoretical peak performance of the GPU.

Chapter 7

# IDG on FPGAs

The contents of this chapter are based on the following paper:

**Radio-Astronomical Imaging: FPGAs vs GPUs**
**Veenboer, B.**, Romein, J. W.
In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), pages 509-521, Springer, 2019* (**best paper award)**

In this chapter we answer the following research sub question:

**RQ3c:** *How efficient is IDG on FPGAs?*

## 7.1 Introduction

FPGAs excel in performing simple operations on high-speed streaming data, at high (energy) efficiency. However, so far, their difficult programming model and poor floating-point support have prevented a wide adoption for typical HPC applications. This is changing, due to recent FPGA technology developments: support for the high-level OpenCL programming language, hard Floating-Point Units, and tight integration with CPU cores. Combined, these are game changers: they dramatically reduce development times and allow using FPGAs for applications that were previously deemed too complex.

In this chapter, we demonstrate how we implemented and optimized a complex radio-astronomical imager on an Arria 10 FPGA. We compare architectures, programming models, optimizations, performance, energy efficiency, and programming effort to highly optimized GPU and CPU implementations. We show that we can efficiently optimize for FPGA resource usage, but also that optimizing for a high clock speed is difficult.

The rest of this chapter is organized as follows: Section 7.2 provides background information on FPGA programming using the Intel FPGA SDK for OpenCL. Section 7.3 explains how we implemented and optimized the most critical parts of the Image-Domain Gridding algorithm. In Section 7.4 we analyze performance and show energy efficiency measurements. Section 7.5 describes the lessons that we learned while implementing and optimizing the same application for both FPGAs and GPUs. We discuss related work in Section 7.6.



## 7.2 Background

A Field-Programmable Gate Array (FPGA) is a chip that contains a number of configurable elements, such as registers, memory blocks (similar to L1 cache), logic blocks, and transceivers, which are connected through reconfigurable interconnects. An FPGA design is a particular configuration of the elements and interconnects, such that the FPGA executes some fixed functionality. Floating-point arithmetic traditionally had to be implemented by combining logic blocks into a circuit which is inefficient as it consumes many resources.

Recent FPGAs, such as the Intel (formerly Altera) Arria 10 and Stratix 10, contain variable-precision Digital Signal Processing (DSP) blocks with a floating-point multiplier and a floating-point adder such that floating-point units (FPUs)

can be implemented in hardware. This makes floating-point computations much more efficient, making FPGAs a potentially interesting target for high-performance computing.

OpenCL is a framework for writing accelerated programs [3]. Using the Intel FPGA SDK for OpenCL, FPGA designs can now be implemented using a high-level programming language [73]. An OpenCL FPGA design is compiled into a dataflow pipeline. To this end, the OpenCL compiler allocates a certain amount of each of the available resources on the FPGA, such as memory blocks and DSP blocks. Each allocated resource corresponds to a specific location in the source code and cannot be employed for other operations in the source code. This makes it especially important to optimize resource usage.

OpenCL GPU kernels are typically *NDRange kernels*: kernels for which the work is divided into *work items*, which are grouped into *work groups*. A GPU several compute units with a number of cores in each compute unit organized similar to vector units on a CPU. The GPU distributes the work by issuing many work items in parallel onto the compute units, where groups of work items execute the same operation on different data. By having many independent instructions in flight, memory and/or instruction latencies are hidden, resulting in optimal utilization of the available compute units. On FPGAs we use *single work-item kernels* to explicitly express a dataflow network. Parallelism is achieved by running multiple single-work-item kernels concurrently within a pipeline. In an efficient design, every cycle, every DSP in the pipeline performs an operation and shifts the results to the next stage in the pipeline. Additionally, parallelism is achieved by replicating parts of a pipeline, and by placing multiple independent kernels onto the FPGA.



Communication between the work items is facilitated by *channels* (an Intel OpenCL extension). Channels are First In, First Out (FIFO) buffers where values of arbitrary *width* can be enqueued at one end and dequeued at the other end. An attribute called the *channel depth* can be set, which indicates the length of the FIFO buffer.

OpenCL GPU kernels are typically compiled at runtime, just before a kernel is used for the first time. For FPGAs, compilation takes much longer: hours instead of seconds. The result of such a compilation is a particular configuration of the elements and interconnects, along with a *clock frequency* at which this design runs. The maximum clock frequency (the $F_{max}$) depends on the complexity of (parts of) the design and the lengths of the paths between elements on the FPGA. Placement

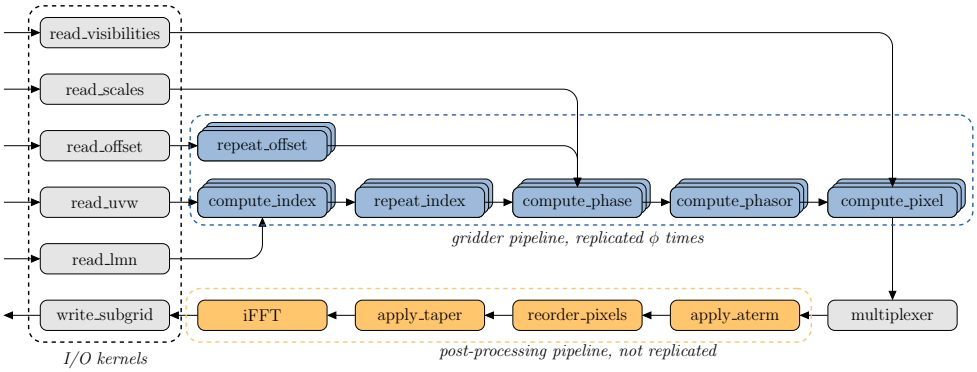



***Fig. 7.1:*** *All kernels in this design are single-work-item kernels. The majority of the computation takes place in the gridder pipeline, which is replicated $\phi$ times to compute multiple subgrids in parallel. These subgrids are multiplexed and passed to the post-processing pipeline, which applies A-term correction, tapering, and a 2D FFT.*

and routing of the design is a stochastic process. One can compile using different *random seeds* and pick the compilation which has the highest clock.

On an FPGA, the performance of an application is bound by the achieved clock frequency of a design and the number of DSP blocks used. When using OpenCL, the relation between the design of the program and the clock frequency is mostly opaque and cannot be influenced directly. It is only possible to use all DSP resources when the design is not constrained by another resource, such as memory blocks or logic.



## 7.3 Implementation

FPGA applications are typically implemented as a data-flow pipeline. We show the data-flow pipeline that we created for the Image-Domain Gridding algorithm (Algorithm 1) in Figure 7.1. The floating-point operations in this algorithm are implemented in hardware using DSP blocks. Our design is scalable and optimizes both the number of DSPs used and the occupancy of these DSPs such that every cycle, (nearly) every DSP performs a useful computation. Although the computations in gridding and degridding are similar, the degridding data-flow network is different and not shown in Figure 7.1.

To implement gridding on the FPGA, we applied the following changes to Algorithm 1: (1) we create a *gridder pipeline* that executes Line 3 through Line 12 to

compute a single subgrid; (2) we move the computation of the *index* value (Line 5) and the computation of *offset* (Line 3) into separate kernels to avoid underutilization of the DSPs used to implement these computations; (3) we unroll the loop over pixels (Line 2) to increase reuse of input data; (4) we replicate the gridder pipeline by a factor $\phi$ to compute multiple subgrids in parallel; (5) input data (such as the visibilities, Line 11) is read from DRAM in bursts in separate kernels and forwarded to the gridder pipelines in a round-robin fashion.

The remaining steps are implemented in the form of a *post-processing pipeline* using as few resources as possible while still meeting throughput requirements imposed by the gridder pipelines. A-term correction (Line 17) is implemented as a series of two complex $2 \times 2$ matrix multiplications (one correction matrix per receiver). Tapering (Line 18) is implemented as a scalar multiplication to every pixel in the subgrid. The 2D FFT (Line 19) is based on the 1D Cooley-Tukey FFT algorithm, which is applied to the rows and columns of the subgrid to perform a 2D FFT.

### 7.3.1 Sine/cosine computations

The OpenCL compiler recognizes the sine and cosine pair and uses 8 memory blocks and 8 DSPs to implement it by creating an *IP block* ($cis_{ip}$). In comparison, only a single DSP is used to compute the *phase* term on Line 8, and 16 DSPs are used to implement the computation on Line 12. Hence, 8 out of every 25 DSPs (32%) are used for sine/cosine computations. To reduce resource usage for $cis(x)$, we investigated how lookup tables can be used as an alternative to the compiler-generated version. In the case of $cis(x)$ the input $x$ is an angle and the output is given as a coordinate on the unit circle, which opens opportunities to exploit symmetry. Our lookup table implementation ($cis_{lu}$) contains precomputed values for $sin(x)$ in the range of $[0 : \frac{1}{2}\pi]$. We use one DSP to convert the input $x$ to an integer index and then derive indices for $sin(x)$ and $cos(x)$ using *logic elements*. We analytically determined that a 1024-entry table provides sufficient accuracy for this application. The lookup table is stored in local memory blocks, which are automatically replicated by the compiler to provide enough internal bandwidth for the many simultaneous sine/cosine lookups.

### 7.3.2 Frequency optimization

The OpenCL FPGA compiler gives feedback on resource usage by generating HTML reports, which is highly useful when optimizing resource usage. Optimizing for high clock frequencies is difficult though: apart from a few general guidelines, there is

little guidance, such as feedback on which part of a (large) program is the clock frequency limiter. There are low-level Quartus timing reports, but these are difficult to comprehend by OpenCL application programmers. Also, even though the FPGA has multiple clock domains, these are not exposed to the programmer. The whole OpenCL program runs at a single clock frequency. Hence, a single problematic statement, possibly not even in the critical path, can slow down the whole FPGA design.

We developed the following method to find clock-limiting constructs: we split the OpenCL program into many small fragments, added dummy data generators and sink routines (so that the compiler does not optimize everything away), and compiled each of these fragments, to determine their maximum clocks. This way, we found for example that a single inadvertently placed modulo 13 operation slowed down the whole application, something which was difficult to pinpoint but easy to fix.

### 7.3.3 Resource optimization

OpenCL GPU kernels are typically compiled the first time they are used, at runtime. The compilation of OpenCL FPGA kernels is a much more lengthy process that, depending on the complexity design and the target FPGA, usually takes multiple hours. OpenCL kernels for an FPGA are therefore compiled offline and loaded onto the FPGA at runtime. The compiler allows the user to run only the first phase of the compilation, where the compiler produces (amongst other things) a report with an estimate of the required FPGA resources. This step takes minutes, not hours, and allows the user to iteratively improve the FPGA design before running the full (lengthy) compilation.



## 7.4 Results

We first show the resource usage when our designs are compiled for an Arria 10 FPGA in Section 7.4.2. Next, we compare throughput and energy efficiency with a comparable CPU and GPU in Section 7.4.3. We analyze the achieved performance in more detail in Section 7.4.4.

### 7.4.1 Experimental setup

We compare our gridding and degridding design on an Arria 10 FPGA to the CPU and GPU implementations presented in Chapter 5 and 6, respectively. We use

***Table 7.1:*** *The Intel Haswell-EP CPU (Xeon E5-2697v3), Intel Arria 10 FPGA (Nallatech 385A) and NVIDIA Maxwell GPU (GTX 750 Ti) used in our experiments.*

| Name | # FPUs | Peak | Bandwidth | TDP | Procedure |
|---|---|---|---|---|---|
| HASWELL | 224 | 1.39 TFlop/s | 68 GB/s | 145W | 28nm (TSMC) |
| ARRIA | 1518 | 1.37 TFlop/s | 34 GB/s | 75W | 20nm (TSMC) |
| MAXWELL | 640 | 1.39 TFlop/s | 88 GB/s | 60W | 28nm (TSMC) |

contemporary devices with a similar theoretical peak performance and produced using a similar lithographical process. We, therefore, use only one socket of the HASWELL system that we used earlier and an NVIDIA GPU based on the Maxwell architecture. See Table 7.1 for details. Note that these devices are different from the devices used previously: we use only one CPU of a dual-socket system and an older and less powerful GPU compared to the GPUs in Chapter 6. This is a deliberate choice, as it allows for a fair comparison. Ideally, these three devices should perform similarly. In practice, as we will show later, this is not the case. Therefore, this methodology allows us to uncover the architectural differences relevant in achieving high performance and high energy efficiency for IDG.

The imaging parameters are set as follows: $\bar{N} = 32$, $T = 128$, and $C = 16$. The FPGA designs are scaled up by increasing $\phi$ until the maximum number of DSPs is reached. The GPU (MAXWELL) uses the 396.26 GPU driver and CUDA version 9.2.88.

The Arria 10 GX 1150 FPGA (ARRIA) comes in the form of an PCIe accelerator card and has two banks of 4 GB DDR3 memory. The FPGA runs a Board-Support Package (BSP) that is required to use the FPGA using Intel's OpenCL platform. We use the *min* BSP, which exposes all 1518 DSPs present on the FPGA to the application and uses only one DDR3 memory bank. We tested various combinations of the Intel FPGA SDK for OpenCL (versions 17.1, 18.0 and 18.1), recompiled each application with dozens of seeds, and report the results for the version that achieves the best clock frequency.

### 7.4.2 Resource usage

We refer to designs that use $cis_{ip}$ (sine/cosine using IP blocks) as GRIDDING-IP and DEGRIDDING-IP, while the GRIDDING-LU and DEGRIDDING-LU designs use our alternative implementation with lookup tables ($cis_{lu}$). We report resource usage and

***Table 7.2:*** *Resource usage of our gridding and degridding designs on* ARRIA. *Logic (*ALUT*s or* FF*s) is counted in terms of thousand elements. The $\phi$ parameter is used to scale up the design, see Fig. 7.1. $F_{max}$ is the achieved clock frequency of the design in MHz.*

| | ALUTs | FFs | RAMs | DSPs | MLABs | $\phi$ | $F_{max}$ |
|---|---|---|---|---|---|---|---|
| GRIDDING-IP | 43% | 31% | 64% | 95% | 71% | 14 | 258 |
| DEGRIDDING-IP | 47% | 35% | 72% | 95% | 78% | 14 | 254 |
| GRIDDING-LU | 27% | 32% | 61% | 99% | 57% | 20 | 256 |
| DEGRIDDING-LU | 33% | 38% | 73% | 99% | 69% | 20 | 253 |

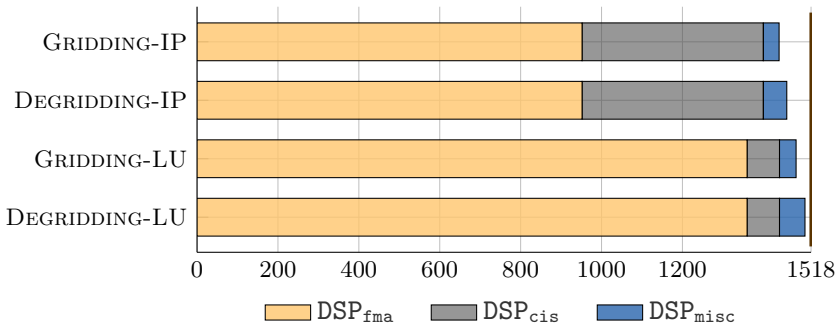



***Fig. 7.2:*** *Breakdown of DSP resource usage. The vertical line at* 1518 *corresponds with the number of DSPs available on* ARRIA.



the highest achieved clock frequency ($F_{max}$) of all designs in Table 7.2. In all four designs, the number of DSPs used is very close to the 1518 DSPs available and we run out of DSPs before we run out of any other resource (which is good; if we would have ran out of another resource before we ran out of DSPs, we would not be able to get close to the peak performance). We provide a breakdown of DSP resource usage in Figure 7.2 where we distinguish between the DSPs used to implement various subparts of the algorithm. For instance for gridding (Algorithm 1), we distinguish between the DSPs used to implement the complex multiplication and addition in Line 12 ($\texttt{DSP}_{\texttt{fma}}$), the sine/cosine evaluation in Line 9 ($\texttt{DSP}_{\texttt{cis}}$) and miscellaneous computations ($\texttt{DSP}_{\texttt{misc}}$), e.g. in the post-processing steps. The implementation of computations outside of the critical path consumes few resources ($\texttt{DSP}_{\texttt{misc}}$). Since $cis_{lu}$ uses fewer resources compared to $cis_{ip}$ to implement the sine/cosine evaluation, we can scale up GRIDDING-LU and DEGRIDDING-LU further (by increasing $\phi$ from 14 to 20) than is possible with GRIDDING-IP and DEGRIDDING-IP.

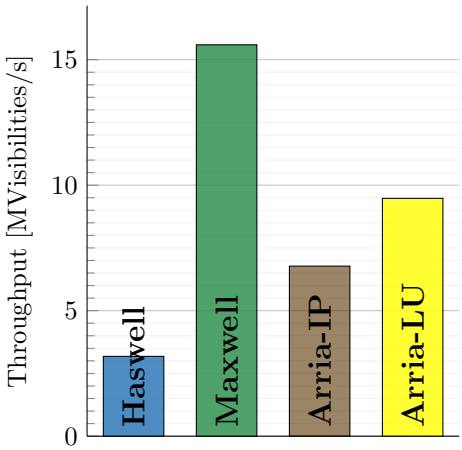



***Fig. 7.3:*** *Imaging throughput*

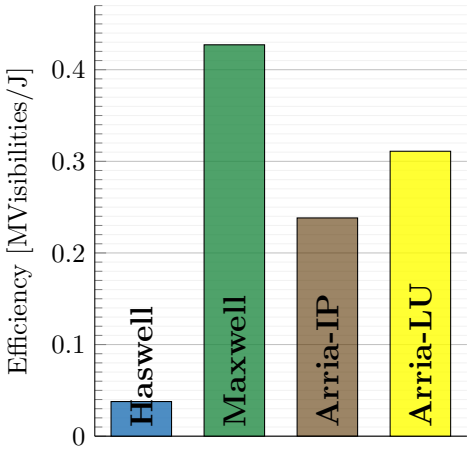



***Fig. 7.4:*** *Energy efficiency*

### 7.4.3 Throughput and energy efficiency

We compare throughput in Figure 7.3. The designs that use a lookup table to implement the sine/cosine evaluation ($cis_{lu}$) achieve a higher throughput due to a larger number of replicated gridder or degridder pipelines running in parallel. Both Arria and Maxwell accelerate gridding and degridding compared to Haswell by achieving more than double the throughput.

On both the FPGA and GPU the visibilities (and other data) are copied to and from the device using PCIe transfers. On Maxwell, we can fully overlap PCIe transfers with computations, such that throughput is not affected by these transfers. On Arria, we found that PCIe transfers overlap only partially: the FPGA idles 9% of the total runtime waiting on PCIe transfers. This is probably a limitation in the OpenCL runtime or Board Support Package. We see no fundamental reason why PCIe transfers could not fully overlap on the FPGA.

We use PowerSensor [48] (see also Chapter 3) to measure the energy consumption of the PCIe accelerator devices (Maxwell and Arria). On Haswell, we use LIKWID [47] to measure the energy consumption of the CPU cores and the associated DRAM memory. Our measurements in Figure 7.4 indicate that both accelerators are much more energy-efficient than Haswell by processing over an order of magnitude more visibilities for every Joule consumed.

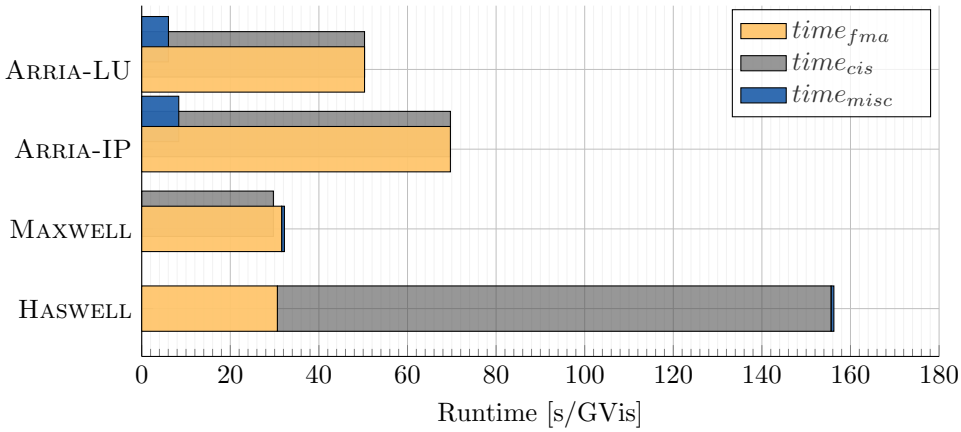



***Fig. 7.5:*** *Runtime breakdown*

### 7.4.4 Performance analysis

Despite their almost identical theoretical peak performance, there is quite a large disparity between the achieved throughput on the various devices. As we illustrate in Figure 7.5, these differences are mainly caused by how sine/cosine ($cis(x)$) is implemented. On Haswell we use MKL to evaluate $cis(x)$ in *software*: MKL issues (vector) instructions that are executed by the FPUs in the CPU cores. On Maxwell, Special Function Units (SFUs) evaluate $cis(x)$ in hardware in a separate processing pipeline, such that FMAs and sine/cosine evaluations can be overlapped. Similarly, the distinct operations ($fma$, $cis$, and $misc$) also overlap on Arria, since these are all implemented using dedicated DSPs. However, unlike Maxwell, these operations compete for resources. On Haswell and Maxwell, the miscellaneous operations contribute negligibly to the overall runtime. On Arria, the *misc* operations are implemented using as few DSPs as possible (and shared by multiple gridder pipelines) to minimize underutilization.

We analyze the achieved floating-point performance by applying the roofline model, see Figure 7.6. In this analysis, we only include all $+$, $-$ and $\times$ floating-point operations in the operation count (e.g. $Flops_{fma} + Flops_{misc}$), while we exclude all $cis(x)$ operations (e.g. $Ops_{cis}$). According to the operational intensity, the performance of gridding and degridding is *compute-bound* on all devices. As we illustrated in Figure 7.5, on Haswell the *Flops* and *Ops* are both executed on the FPUs and the performance is therefore bound by the instructions executed to

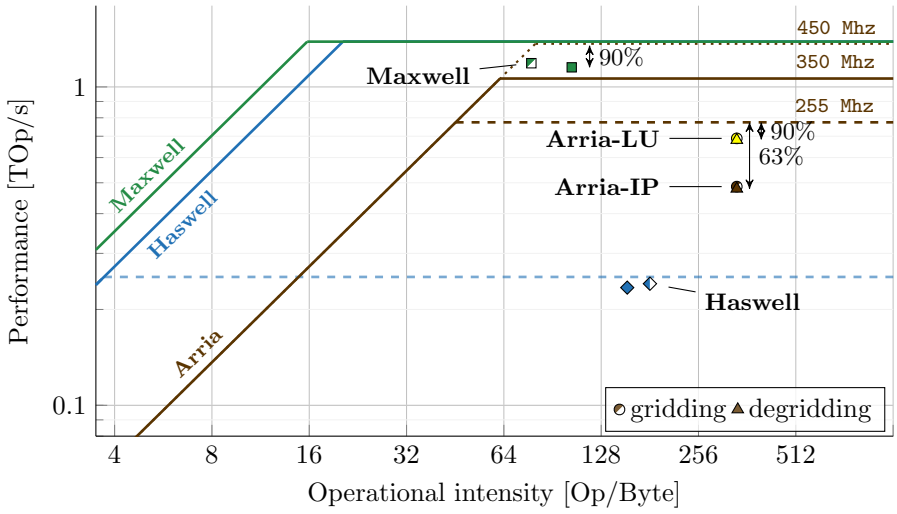



***Fig. 7.6:*** *Roofline analysis*

evaluate sine/cosine (indicated with the dashed line). Due to the SFUs, this problem does not occur on MAXWELL, and the achieved performance is about 90% of the theoretical peak.

The dotted line on the roofline for ARRIA illustrates the theoretical peak, at the advertised frequency of 450 Mhz. In practice, even with only a single DSP used, the maximum clock frequency that the compiler achieves is 350 Mhz resulting in a lower practical peak indicated by the solid line. The achieved clock frequency for our gridding and degridding design is about 255 Mhz on average and is indicated with the dashed line. The percentage of DSPs used to implement *Flops* (63% for GRIDDING-IP and DEGRIDDING-IP, 90% for GRIDDING-LU and DEGRIDDING-LU, see Figure 7.2) provides upper bounds on attainable performance. The achieved performance is within 99% of these bounds, indicating that the designs are nearly stall-free and perform a useful operation nearly every cycle. Note that stalls due to PCIe transfers are not taken into account for this analysis (see Section 7.4.3).

## 7.5 Lessons learned

As we implemented and optimized Image-Domain Gridding for both FPGAs and GPUs, we found differences and similarities with respect to architecture, programming model, implementation effort, optimizations, performance, and energy efficiency.

The source code for the FPGA imager is highly different from the GPU code. This is mostly due to the different programming models: with FPGAs, one builds a dataflow pipeline, while a GPU executes instructions. The FPGA code consists of many (possibly replicated) kernels that each occupy some FPGA resources, and these kernels are connected by *channels* (FIFOs). The programmer has to think about how to divide the FPGA resources (DSPs, memory blocks, logic, etc.) over the pipeline components, so that every cycle all DSPs perform a useful computation, avoiding bottlenecks and underutilization. Non-performance-critical operations, such as initialization routines, can consume many resources, while on GPUs, performance-insensitive operations are not an issue. On FPGAs, it is also much more important to think about timing (e.g., to avoid pipeline stalls), but being forced to think about it leads to high efficiency: in our gridding application, no less than 96% of all DSPs perform a useful operation 99% of the time.

FPGAs have typically less memory bandwidth than GPUs, but we found that with the FPGA dataflow model, where all kernels are concurrently active, it is less tempting to store intermediate results off-chip than with GPUs, where kernels are executed one after another. Our FPGA designs use memory only for input and output data; we would not even have used FPGA device memory at all if the OpenCL Board-Support Package would have implemented the PCIe I/O channel extension. In contrast, the cuFFT GPU library even requires data to be in off-chip memory.



Both FPGAs and GPUs obtain parallelism through kernel replication and vectorization; FPGAs also by pipelining and loop unrolling. This is another reason why FPGA and GPU programs look different. Surprisingly, many optimizations for FPGAs and GPUs are similar, at least at a high level. Maximizing FPU utilization, data reuse through caching, memory coalescing, memory latency hiding, and FPU latency hiding is necessary optimizations on both architectures. For example, an optimization that we implemented to reduce local memory bandwidth usage on the FPGA also turned out to improve performance on the GPU, but somehow, we did not think about this GPU optimization before we implemented the FPGA variant. [1] However, optimizations like latency hiding are much more explicit in FPGA code than in GPU code, as the GPU model implicitly hides latencies by having many

[1] The performance of our initial gridder and degridder kernel for NVIDIA GPUs (presented in [11]) was bound by the memory of shared memory. After coming up with this optimization for FPGAs, we tested a similar optimization for our GPU kernels, which indeed also reduced local memory bandwidth usage, and therefore increased performance. We take this optimization into account in Chapter 6 where we present our GPU implementation and performance analysis.

simultaneous instructions in flight. On top of that, architecture-specific optimizations are possible (e.g., the sin/cos lookup table; see Section 7.3.1).

Overall, we found it more difficult to implement and optimize for an FPGA than for a GPU, mostly because it is difficult to efficiently distribute the FPGA resources over the kernels in a complex dataflow pipeline. Yet, we consider the availability of a high-level programming language and hard FPUs on FPGAs an enormous step forward. The OpenCL FPGA tools have considerably improved during the past few years, but have not yet reached the maturity level of the GPU tools, which is quite natural, as the GPU tools have had much more time to mature.

## 7.6 Related work

Several studies compare energy efficiency between OpenCL applications for FPGAs and GPUs [74–79]. In most cases, they compare FPGAs and GPUs manufactured using a similar lithographical process and report higher energy efficiency for FPGAs compared to GPUs. We compared contemporary and comparable devices (in terms of lithographical process and peak performance) and showed that our implementations perform close to optimal both on the FPGA and on the GPU. On Arria 10 we show that the performance of our designs is bound by clock frequency, something we can not improve with the current OpenCL compiler for FPGAs. On GPUs, this issue is non-existent. Furthermore, we explained that GPUs have an advantage over FPGAs, by computing sine/cosine using dedicated hardware. Therefore, and in contrast to what the related work suggests, our results indicate that FPGAs are not necessarily more energy-efficient than GPUs.



## 7.7 Conclusion

The high-level OpenCL programming environment enabled us to implement IDG on an Intel Arria 10 FPGA, and to optimize resource usage and resource utilization: we were able to use nearly all the available DSPs to perform useful computations, almost every cycle. However, we were unable to achieve peak clock frequencies, and the FPGA tools provide insufficient means to improve this.

Being able to implement a complex application for a FPGA illustrates that having support for a high-level programming language is a major leap forwards in programmability, as we would not have been able to implement this application using a hardware description language.

FPGAs are traditionally used for low-latency, fixed-point, and streaming computations. With the addition of hardware support for floating point computations and the OpenCL programming model, the FPGA has also entered the domain where GPUs are used: high-performance floating-point applications.

Having implemented IDG on both GPUs and on an FPGA, we conclude that for this application GPUs are superior accelerators over the Arria 10 FPGA for a number of reasons: (1) NVIDIA GPUs support the evaluation of sine/cosine in dedicated hardware, which put the GPUs at a significant advantage; (2) clock frequency bottlenecks as we encountered on Arria 10 are no concern on GPUs; (3) GPUs are still easier to program than FPGAs; (4) the GPU programming environment (e.g., the compiler and profiler) are more mature than the Intel FPGA SDK for OpenCL. Intel has announced a new generation FPGAs (Agilex), which provide better theoretical peak performance and energy efficiency compared to Arria 10. Together with ongoing improvements to the programming tools, we expect that FPGAs will become an increasingly popular accelerator platform for general HPC applications.

So far we used the FPGA as an accelerator, and like on a GPU, we used it to offload the computation (and associated data) from a host system to an accelerator device. We discussed that for the fastest GPUs available today, a high-bandwidth interconnect (such as NVLink) is needed to keep the GPU busy. While we did not encounter an interconnect I/O bottleneck for the Arria 10, this will likely change for future (faster) FPGAs. Some FPGAs boards provide (multiple) high-bandwidth (100 GBit/s) interfaces, which open up possibilities for a streaming implementation, where data is transferred directly to the accelerator, rather than via a host system. This is a distinguishing feature that sets FPGAs apart from other accelerators and might prove to be a major advantage for specific workloads.



We demonstrated that our IDG implementation for the Intel Arria 10 FPGA achieves almost three times the throughput of an optimized CPU implementation while the FPGA consumes about eight times less energy than the CPU. This illustrates that FPGAs have become a feasible accelerator platform for high-performance floating-point applications. Programming and optimizations of FPGAs remain challenging, even using OpenCL.

**RQ3c:** *How efficient is IDG on FPGAs?*

FPGAs are efficient accelerators for IDG, as our designs are able to utilize (almost) all the DSP resources to perform useful computations at every cycle.

Chapter 8

# CPUs versus GPUs versus FPGAs

This chapter does not present any new research. Instead, we summarize the results of Chapter 5 through 7 to answer the following research question:

**RQ3:** *What is the most efficient class of hardware architectures for IDG?*

## 8.1 Introduction

In the previous chapters, we explained how we implemented Image-Domain Gridding (IDG) for CPUs, GPUs, and FPGAs and we showed the performance and energy efficiency achieved on these classes of hardware architectures. In this chapter, we compare the results to find the most suitable architecture for IDG.

## 8.2 Performance bounds

The roofline model proved to be a highly useful method to visualize performance and to find performance bounds and optimization opportunities. Every device has a theoretical peak performance ($peak_{MAX}$), but we found that for no device this level of performance is attainable in practice. In all cases there is another limit, $peak_{IDG}$, that limits the performance of our IDG implementation. We optimized the IDG gridder and degridder kernels such that they perform as close to this limit as possible. Table 8.1 provides an overview of the performance bounds and the percentage of the peak performance achieved.

***Table 8.1:*** *Performance bounds for the architectures that we evaluated. $peak_{MAX}$ is the theoretical floating-point peak performance that a device can achieve when it executes single-precision floating-point operations such as $+$, $-$ and $*$. $peak_{IDG}$ denotes the architecture-specific performance bound for IDG, which is the peak performance for the instruction mix in the gridder and degridder kernel, consisting of 17 real-valued fused multiply-add operations and one evaluation of sine/cosine.*

| Name | Type | % of $peak_{MAX}$ | % of $peak_{IDG}$ | Bound by |
|---|---|---|---|---|
| Haswell | CPU | 20% | 95% | sine/cosine in software |
| Knl | CPU | 15% | 70% | sine/cosine in software |
| Maxwell | GPU | 90% | 95% | latency |
| Pascal | GPU | 90% | 95% | latency |
| Vega | GPU | 60% | 90% | sine/cosine in hardware |
| Arria | FPGA | 70% | 99% | clock frequency |

Both Haswell and Knl are bound by the evaluation of sine/cosine in software. On Haswell the percentage of the peak (both for $peak_{MAX}$ and for $peak_{IDG}$ is higher than for Knl. This is mainly due to load imbalance and (partially) masked vector instructions. Since IDG is compute-bound, our implementation for Knl does not benefit from architectural features as MCDRAM or four-way hyperthreading to

gain a performance advantage over Haswell. For IDG, Knl is therefore little more than a many-core CPU running at lower clock frequency which requires code to be vectorized with longer vector length compared to Haswell.

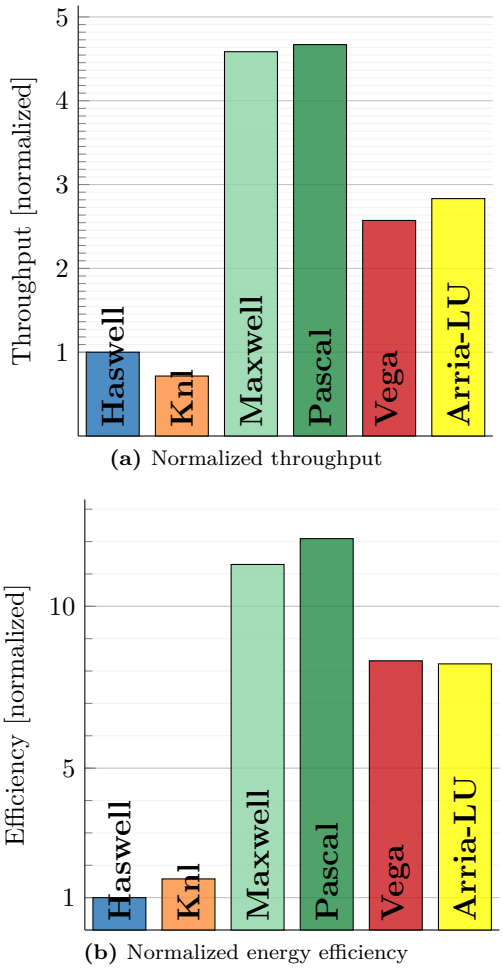
The GPUs support the evaluation of sine/cosine in hardware, but the implementation is different: Vega uses the floating-point units to evaluate them at a quarter of the rate of normal floating-point additions or multiplications. NVIDIA GPUs on the other hand has dedicated special-function units to evaluate sine/cosine simultaneous with other computations. This gives NVIDIA GPUs a very significant advantage over Vega which is also reflected in the much higher percentage of peak performance that is achieved.

The FPGA has a completely different architecture compared to the CPUs and GPUs discussed above. It is programmed as a dataflow engine where an application-specific circuit is created. We were able to use close to all floating-point units (DSPs) and these perform useful computations in almost all cycles. However, the performance is limited by the clock frequency.

## 8.3 Throughput and energy efficiency

We have measured the throughput on devices with different theoretical peak performances. To compare the underlying architecture, and not the specific device, we normalize the results and show the results in Fig. 8.1a. These results indicate that in relative terms, Knl is not faster than Haswell. The Arria FPGA performs similarly to the Vega GPU. While the two NVIDIA GPUs (Maxwell and Pascal) are based on a different architecture, IDG runs very efficiently on either one of them. In this comparison, the results look similarly as they both achieve over 4× the throughput compared to Haswell.

We compare the architectures in terms of energy efficiency in Fig. 8.1b. Knl is slightly more energy-efficient than Haswell, but this is in stark contrast to Arria and Vega, which achieve 8-fold better energy efficiency. Thanks to improvements to the micro-architecture and by using a different processing technology (16nm versus 28nm), NVIDIA GPUs have become more energy-efficient over the years. This is also reflected in these results, which indicate that Pascal is more energy-efficient than Maxwell. Pascal demonstrates superior energy efficiency and is more than an order of magnitude more energy-efficient compared to Haswell.




**(a)** Normalized throughput

**(b)** Normalized energy efficiency

***Fig. 8.1:*** *We normalized throughput and energy efficiency with* HASWELL *as a baseline and we corrected for the theoretical peak floating-point performance of the various devices that we analyzed. This allows us to compare the architectures in relative terms. Thus if we would run IDG on a Haswell CPU and a Pascal GPU with identical peak floating-point performance, the GPU would achieve a* 4× *higher throughput while consuming less than* 10% *of the energy compared to the CPU.*

## 8.4 Conclusion

The performance of IDG is highly dependent on the support for sine/cosine evaluations. Haswell and Knl only support these evaluations in software, which makes these architectures less suitable for IDG than architectures that offer native support. On Arria, we created a custom data-flow circuit tailored for IDG. However, Pascal supports the evaluation of sine/cosine natively in a dedicated processing pipeline and therefore performs best. Also in terms of efficiency, Pascal is the best architecture. Furthermore, as we also discussed in Chapter 7, GPUs are significantly easier to program than FPGAs, despite using a similar programming model.

**RQ3:** *What is the most efficient class of hardware architectures for IDG?*

The most efficient class of hardware architecture for IDG is GPUs. Especially NVIDIA GPUs with support for sine/cosine in hardware perform excellently and achieve superior energy efficiency.

## 8.5 Outlook

The Pascal GPU architecture supports 16-bit (half-precision) data types and arithmetic operations. For the GA102 GPU as found in the GTX 1080, the performance is only 1/64 of the 32-bit (single-precision) floating-point performance. For the GA100 GPU (as found in the Tesla P100) however, the half-precision performance is twice that of single-precision. Moreover, newer generations of GPUs (Volta and newer) support Tensor Cores, which provide 8 times the single-precision performance for special matrix math at half-precision. For IDG, it is not straightforward to take advantage of these advancements in GPU architecture, for two main reasons: 1) we don't yet know whether radio-astronomical imaging in general and IDG in specific can tolerate low-precision data types and arithmetic. (See also Section. 4.4); 2) In its current form, the IDG gridder and degridder kernel have an instruction mix well balanced for GPUs like Pascal, with sine/cosine and FMA operations in roughly the same ratio as the functional units on the GPU. Increasing the FMA performance two-fold (e.g., by performing them in half-precision), would require double the number of SFUs to keep up. If not for performance, reduced precision can still help to reduce the data sizes (which is good to alleviate I/O overhead) and might also prove to be more energy-efficient.

Part III

# *Image-Domain Gridding in Context*

Chapter 9

# IDG versus AWPG

The contents of this chapter are based on the following paper:

**Radio-Astronomical Imaging on Graphics Processors**
**Veenboer, B.** Romein, J. W.
In *Astronomy & Computing, Elsevier, Volume 32, July 2020*

In this chapter we answer the following research question:

> **RQ4:** *How does IDG compare to traditional imaging techniques in terms of performance and energy efficiency?*

## 9.1 Introduction

In this chapter we compare IDG with the W-projection implementation introduced in [13] – the fastest W-projection implementation publicly available. This comparison aims to estimate how much better IDG performs in comparison with traditional W-projection. We refer to the optimized W-projection implementation as WPG (W-projection gridding) 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. As no efficient AW-projection implementation was available, we extended WPG to AWPG to include A-term correction to make a complete comparison. Both are available at [80].

## 9.2 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.) [19, 10]. Furthermore, the size of the W-kernel depends on the baseline length. For LOFAR, the W-kernel can be as large as $500 \times 500$ pixels for the longest baseline. In practice, *W-stacking* is used to limit $N_W$ to small values in all situations (e.g., $N_W \leq 16$) [30, 31].

W-planes are distinct copies of the grid, where slices of the visibilities (based on $w$-coordinate) are gridded onto. The W-planes in the W-stack may be processed one by one to save memory, at the cost of having to sort visibilities. Allowing large W-kernels, however, reduces the need to have additional W-planes. This presents a trade-off between the number of computations to be performed, versus the amount of memory required for the W-stack.



## 9.3 AW-projection gridding implementation

We implemented AWPG by extending WPG with A-term corrections. We assume that for AWPG, the W-term and A-term correction terms are combined into a single convolution kernel, which we will refer to as the *AW-kernel*. Furthermore, we assume that the AWPG imager has the following properties: (1) The AW-kernel is different for every baseline; (2) the AW-kernel changes after a (fixed) number of time steps; (3) the A-term, like in IDG, is provided in the image domain; (4) a Fourier transformation is performed to get the AW-kernel into the Fourier domain.

As the A-term correction is baseline dependent and varies over time (properties 1 and 2), the AW-kernel has to be recomputed frequently. The amount of (device) memory required for the AW-kernels 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 properties 3 and 4, a Fourier transformation of the AW-kernel is performed before the gridder kernel is executed. Pseudocode for AWPG is shown in Algorithm 10. 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.

By caching parts of the grid that are currently being updated in registers, WPG reduces the number of (atomic) grid accesses, which significantly increases performance [13]. This optimization is also used in AWPG.

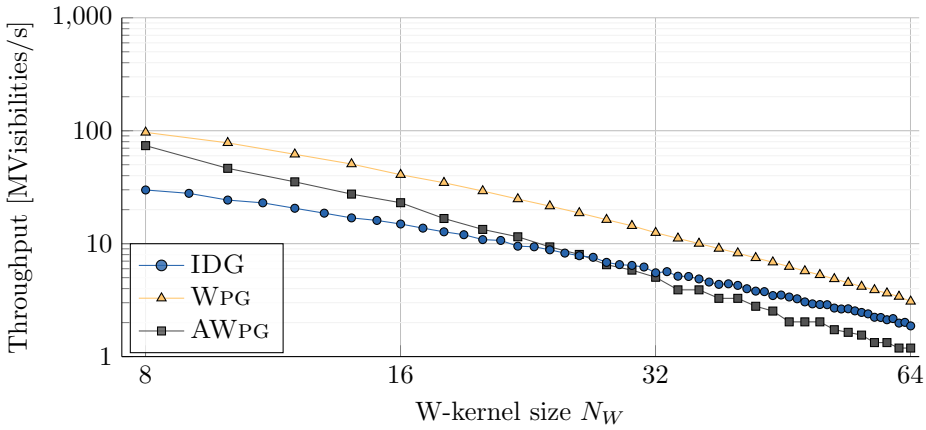
## 9.4 Performance comparison

Fig. 9.1a and 9.1b show the performance of IDG, WPG, and AWPG 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). Both curves show a decline in throughput as the size of the kernel (and hence the number 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 spent 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 than strictly necessary when this increases overall throughput.

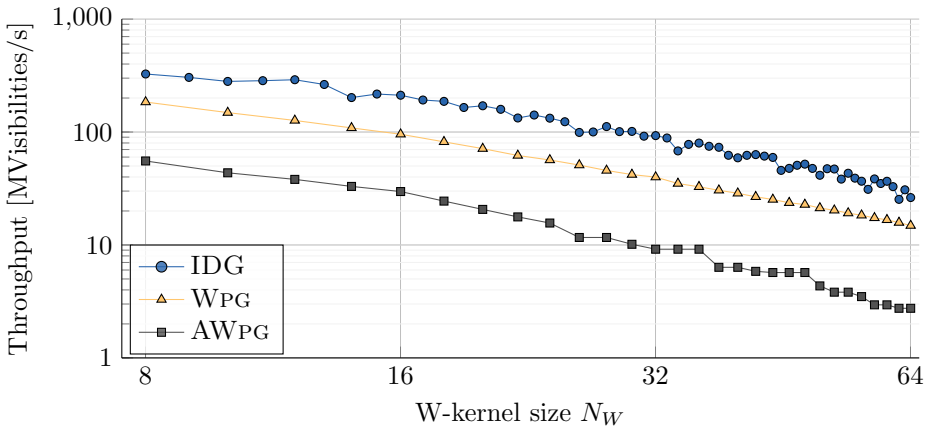
We observe that on HASWELL, IDG is faster than AWPG for larger values of $N_W$. On PASCAL, the differences between WPG, AWPG and IDG is larger than on HASWELL. This is mainly due to WPG and AWPG being mostly I/O bound, while IDG is compute-bound. Due to fewer data transfers between host and device and the W-kernel computation being cheaper than the AW-kernel computation, WPG on average performs about 4× better than AWPG. The highest overall throughput is achieved on PASCAL using IDG, outperforming 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 [64, 81]. Even if WPG and AWPG on PASCAL would be twice as fast, they would still be outperformed by IDG.

```
1  for bl = 1..B_obs do
2      for ts = 1..T_obs..T̄_stepsize do
3          int N_W = compute_kernel_size(bl, ts);
4          complex<float> wkernel = compute_wkernel(N_W);
5          complex<float> [N_W × oversampling][N_W × oversampling] awkernel =
             compute_awkernel(bl, ts, N_W, wkernel);
6          awkernel = apply_2d_fft(awkernel);
7          for t = 1..T̄_stepsize do
8              for c = 1..C_obs do
9                  for y = 1..N_W do
10                     for x = 1..N_W do
11                         for p = 1..P do
12                             complex<float> weight = awkernel(y, x, p);
13                             int y_idx = compute_y_index(bl, t, c, y);
14                             int x_idx = compute_x_index(bl, t, c, x);
15                             complex<float> visibility = visibilities[t][c][p];
16                             grid[p][y_idx][x_idx] += weight * visibility; ;
17                         end
18                     end
19                 end
20             end
21         end
22     end
23 end
```

***Algorithm 10:*** *This pseudocode for* AWPG *illustrates two main differences 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 [13], we use an oversampling rate (*$oversampling = 8$*). The pixel update in Line 16 comprises a complex multiplication and addition, followed by an atomic update of the grid.*

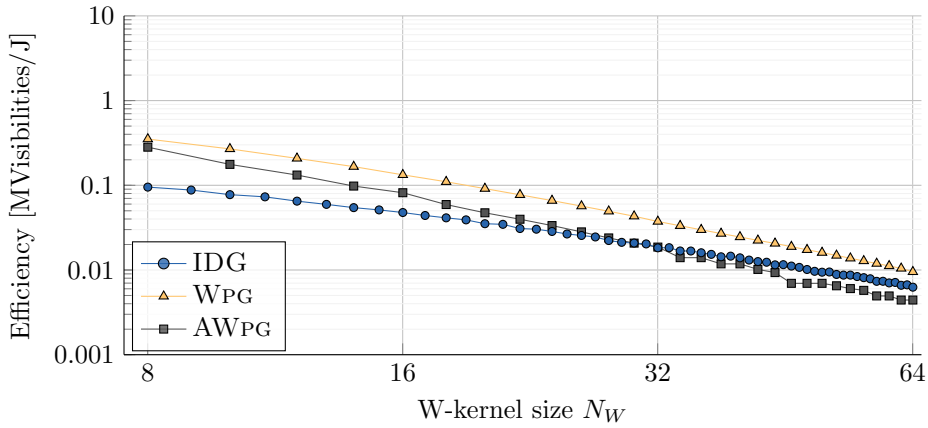



**(a)** CPU throughput

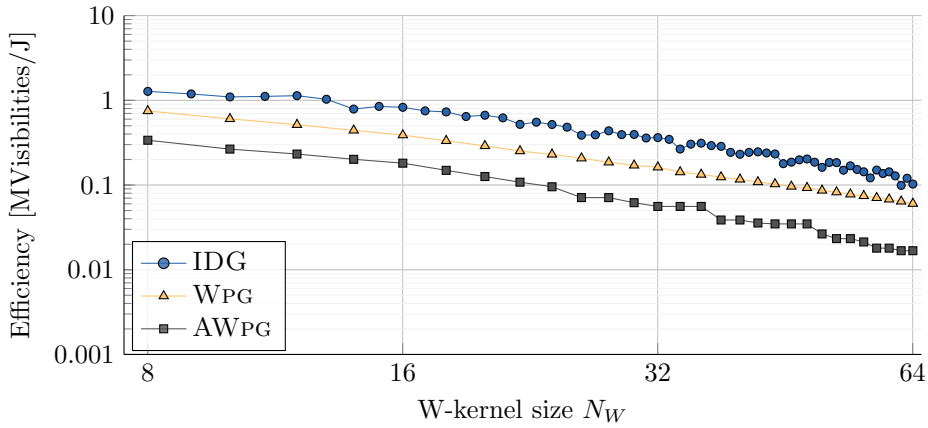



**(b)** GPU throughput

***Fig. 9.1:*** *On the CPU,* WPG *is the fastest gridder (but it does not correct for DDEs).* IDG *performs similarly to* AWPG *for large kernel sizes. On the GPU,* IDG *outperforms both* AWPG *and* WPG*, for all kernel sizes.*




**(a)** CPU efficiency




**(b)** GPU efficiency

***Fig. 9.2:*** *In relative terms, these energy efficiency results match the throughput results in Fig 9.1. 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.

## 9.5 Energy efficiency comparison

We measured the energy consumption for WPG, AWPG, and IDG on HASWELL and PASCAL and found some notable differences, see Fig. 9.2. 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.

## 9.6 Conclusion

We have compared the performance and energy efficiency of W-Projection, AW-Projection and IDG, and can now answer the research question:

> **RQ4:** *How does IDG compare to traditional imaging techniques in terms of performance and energy efficiency?*

On CPUs, IDG is not faster or more energy-efficient compared to W-Projection and AW-Projection. On GPUs, IDG exceeds the performance of the simpler W-projection gridding, while providing the functionality of the more challenging AW-projection gridding.

Chapter 10

# IDG for the Square-Kilometre Array

The contents of this chapter are based on the following paper:

**Radio-Astronomical Imaging on Graphics Processors**
**Veenboer, B.** Romein, J. W.
In *Astronomy & Computing, Elsevier, Volume 32, July 2020*

In this chapter we answer the following research question:

> **RQ5:** *Does IDG meet the performance and energy efficiency requirements for the Square Kilometre Array?*

## 10.1 Introduction

In Section 2.2 we explained that imaging for the Square Kilometre Array (SKA) is a challenging task. We now come back to this challenge by analyzing whether the Image-Domain Gridding (IDG) on GPUs is a viable solution for SKA real-time imaging. To this end, in Section 10.2, we first establish the performance and energy efficiency requirements that a candidate solution should meet. In Section 10.3 we provide an estimate of the performance and energy consumption of a *S*cience Data Processor (SDP) system based on GPUs. In Section 10.4 we extrapolate our results towards this system and in Section 10.5 we conclude.

## 10.2 Required data rates

The SKA community uses a parametric model [36] to analyze processing requirements for the Science Data Processor (SDP) compute platform. This model is available online at [36]. 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 [37] for why this is needed) and take into account that SKA1 Low might only be doing imaging observations half the time [38], 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 6.4.4 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 the case of BLDA will be offset by the lower overall visibility rate.



## 10.3 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 [82]. The computing 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 are estimated to contribute about 13% to the total SDP computation [83]. This implies that $0.13 \times 50$ PFlop/s $\approx 6.5$ 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 built, 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.

## 10.4 IDG for SKA

The average throughput for Unified on Pascal for large images ($40{,}000 \times 40{,}000$ pixels) is about 0.20 GVisibilities/s (see Fig. 6.12). 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. 6.14a), 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 9.4. On Pascal, IDG gridding on average is about an order of magnitude faster and almost $7\times$ 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. $40{,}000 \times 40{,}000$ 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.

The dish-based SKA-1 Mid telescope has a considerably more stable beam shape compared to the dipoles in SKA-1 Low. Therefore, there are also some SKA imaging use-cases where it may not be necessary to apply full DDE correction with A-terms that change continuously throughout the observation. In this setting, the imaging throughput will be somewhere between WPG and AWPG, such that fewer GPUs are needed than computed above. In this case, AWPG with a limited form of DDE correction may also meet the SKA power budget.

## 10.5 Conclusion

IDG runs much more efficiently than 10% of the peak performance generally considered for SDP processing [36]. 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.

> **RQ5:** *Does IDG meet the performance and energy efficiency requirements for the Square Kilometre Array?*

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

Chapter 11

# IDG use cases

The contents of this chapter are based on the following paper:

> **Radio-Astronomical Imaging with WSClean and Image-Domain Gridding**
> **Veenboer, B.**, van der Tol, S., Offringa, A. R., Romein, J. W., Dijkema, T. J., In *General Assembly and Scientific Symposium (GASS) of the International Union of Radio Science (URSI GASS), 2020*

There is no research question that we try to answer in this chapter. Instead, we provide an overview of several use cases of Image-Domain Gridding, illustrating that the research presented in this thesis has a substantial impact on the radio-astronomical scientific community.

## 11.1 Introduction

Image-Domain Gridding started as proof-of-concept developed by Bas van der Tol consisting of a few CUDA kernels along with host code to run these kernels. This code has evolved into a much more extensive and feature-rich astronomical package publicly available at [14]. This package comes with examples that run IDG on realistic albeit generated input data, modeled after the LOFAR [6] or proposed SKA [5] telescope layout. We used this code throughout the thesis to obtain performance and energy efficiency measurements on both kernel level (gridder, degridder) as well as on routine (gridding, degridding) level (i.e. gridding comprises the gridder, subgrid-fft, and adder kernel).

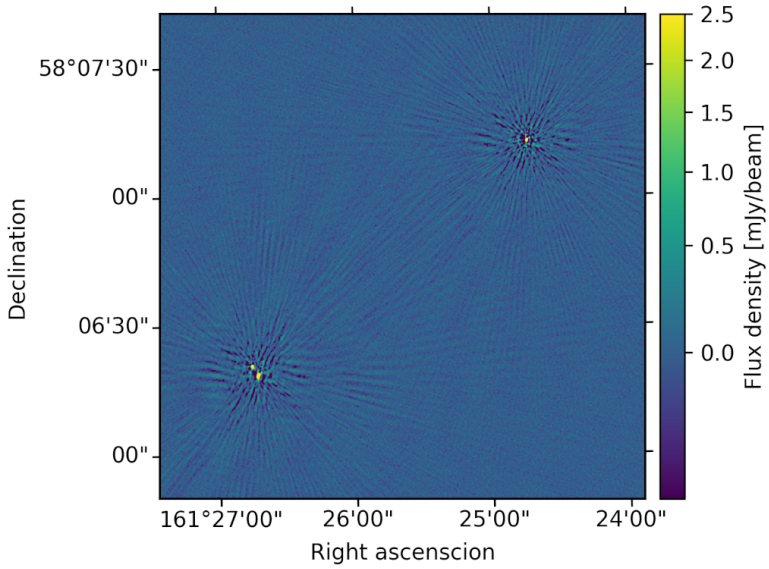
We collaborated with André Offringa to integrate IDG in WSClean [31], such that it could use the high-performance (GPU-accelerated) gridding and degridding routines. This has led to the implementation of many new or improved features to the IDG library, such as the 'Execution Plan', time-varying A-terms *per subgrid*, and 'channel-groups'. Overall, these efforts make IDG a much more robust and widely useable library.

We have published a paper about the use cases of IDG [84] and provide an overview in the following sections.

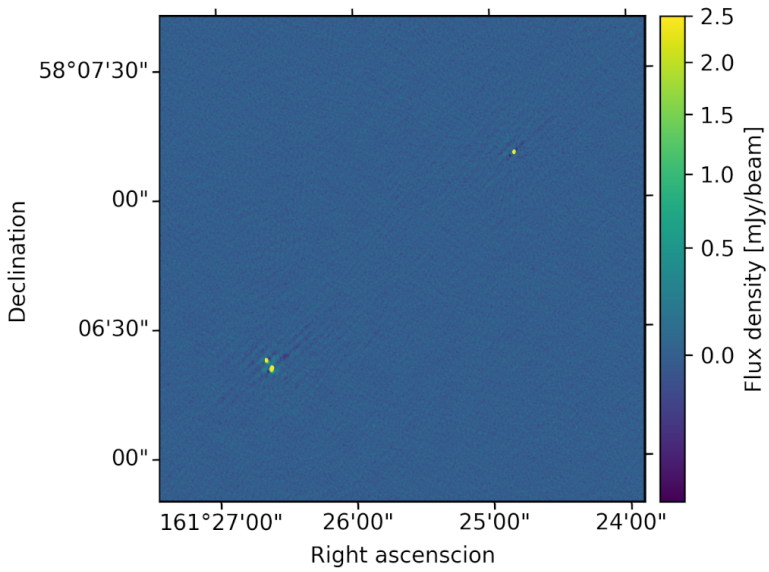
## 11.2 IDG in WSClean

WSClean is a widely-used imager [31] that uses the W-stacking algorithm to implement inversion (gridding) and prediction (degridding) and provides several novel deconvolution algorithms relevant for the SKA. This makes WSClean more robust to calibration errors and an order of magnitude faster than alternative imagers.

We integrated IDG into WSClean such that features as data handling, deconvolution, etc. are maintained, while the existing inversion (gridding) and predict (degridding) functionalities are provided by IDG. Together, WSClean and IDG provide a unique mix of state-of-the-art imaging and deconvolution algorithms. This combination is now used in production by a variety of different radio observatories, such as LOFAR [6] and the MWA [7]. Figure 11.1 illustrates that the direction-dependent ionospheric and gain corrections provided by WSClean+IDG result in superior image quality compared to an image where these corrections are not applied. We are currently working towards creating an imaging pipeline (called 'Rapthor') for LOFAR data processing capable of direction-dependent calibration and imaging.




**(a)** WSClean with corrections for direction-independent effects (DIEs).




**(b)** WSClean with corrections for both direction-independent effects as well as direction-dependent effects (DDEs).

***Fig. 11.1:*** *These images by [85] are an extract of the 'Lockman Hole' field (at 46 Mhz, 0.4″ resolution) and illustrate the benefits of using an imager capable of correcting for direction-dependent effects (DDEs, such as total electron content (TEC) and antenna gain effects) in addition to correction for direction-independent effects (DIEs, such as the shape of the primary beam). By applying a correction for DDEs, the artifacts around the sources are significantly reduced. IDG is the first GPU-accelerated imager capable of efficiently correcting for DDEs during imaging.*

## 11.3 IDG for EoR

Experiments that try to observe the 21-cm redshifted signals from the Epoch of Reionisation (EoR) using interferometric low-frequency instruments have stringent requirements on the processing accuracy. In 21-cm EoR power spectrum experiments, foregrounds are distinguished from the 21-cm signals by their spectral smoothness. Therefore, spectral accuracy is a particularly important aspect of a gridder. Offringa et al. [86] demonstrate that traditional algorithms, with standard settings, are not accurate enough for 21-cm signal extraction. Of the various methods, IDG shows the highest accuracy with the lowest imaging time. Therefore, Image-Domain Gridding is overall the most suitable algorithm for 21-cm EoR power spectrum experiments, including for future analysis of data from the Square Kilometre Array (SKA).

## 11.4 IDG for direction-dependent calibration

Many direction-dependent calibration algorithms split the sky in facets and solve for a gain in each facet, e.g. [87]. Van der Tol et. al. [88] propose a new method in which a smooth screen is directly fitted to the data. This is advantageous in combination with gridders that apply smooth screens, such as IDG as this provides better accuracy and a higher signal to noise. Van der Tol demonstrated IDG-Cal, which is an implementation of such a solver based on the IDG degridder.

## 11.5 Conclusion

We created an imager (WSClean+IDG) that combines the state of the art in deconvolution techniques with an efficient GPU-accelerated imager that corrects for polarized direction-dependent effects during imaging. This imager offers unprecedented imaging capabilities for current and future radio telescopes. This imager is publicly available and is being used in production by various astronomers around the world. Image-Domain Gridding is highly accurate, which makes it very suitable for research into the Epoch of Reionisation. With IDG-Cal we can more accurately calibrate radio-astronomical datasets, which will result in high-quality sky images. Altogether, Image-Domain Gridding contributes towards making discoveries in radio astronomy happen.

Part IV

# *Closing Words*

Chapter 12

# Conclusions and Outlook

In this chapter we first provide an overview of the contributions of this thesis. Next, we summarize our main findings which allow us to answer the main research question:

> **RQ:** *What is the best, fastest, and most energy-efficient imaging algorithm for future radio telescopes?*

This work has lead to new research questions, which we discuss in the final section of this thesis.

## 12.1 Thesis contributions

In this thesis, we set out to implement, optimize, and analyze the Image-Domain Gridding (IDG) algorithm for a variety of accelerator hardware. What started as a proof-of-concept of a new algorithm (by S. van der Tol.), has now become a high-performance radio-astronomical imager that is being used in production by a variety of different radio observatories.

The main contributions of this thesis are as follows:

- The first usable implementation of IDG.
- Highly optimized IDG kernels for CPUs, GPUs, and FPGAs.
- Thorough analysis of the performance and energy efficiency of IDG on the above devices.
- A comparison of the performance and energy efficiency of IDG with traditional imaging algorithms.
- An analysis on whether IDG meets the stringent requirements on performance and energy for the upcoming Square Kilometre Array.
- A software library that is used in production for various radio observatories around the world.

## 12.2 Conclusion

Radio-astronomical imaging is considered to be one of the most challenging stages in the processing pipeline of modern radio telescopes. In this thesis, we used hardware/software co-design to find a method (a new algorithm) and a suitable computing platform (accelerator) to solve this challenge.

We implemented IDG on distinct architectures: an Intel Xeon (Haswell) CPU, an Intel Xeon Phi (Knights Landing), GPUs from NVIDIA (Maxwell and Pascal), an AMD Vega GPU, and an Intel Arria 10 FPGA, performed a thorough performance analysis and we compared performance and energy efficiency.

We demonstrated that especially for the GPUs, the IDG algorithm maps elegantly onto the underlying hardware. Due to hardware support for sine/cosine evaluations, on NVIDIA GPUs, our code achieves close to the floating-point peak performance

and energy efficiency that is more than an order of magnitude higher than the optimized CPU implementation. We also showed that our GPU implementation achieves excellent performance for different imaging use cases, such as continuum imaging and spectral-line imaging, as well as the creation of very large images.

While FPGAs cannot beat GPUs for this application, we showed that having support for a high-level programming language is a major leap forward in programmability. By supporting floating-point operation in hardware, FPGAs have now entered the domain where GPUs are traditionally used.

The comparison with a traditional imaging algorithm illustrates that IDG on GPUs exceeds the performance of the simpler W-projection gridding while providing the functionality of the more challenging AW-projection gridding. Being able to efficiently apply a correction for direction-dependent effects (DDEs), is essential to achieve the dynamic ranges, high sensitivities, and large fields of view of new generations of radio telescopes.

We made IDG available as an open-source library and integrated it with a widely-used imaging application, WSClean. IDG is now being used in production by a variety of different radio observatories. It is not only faster than its competitors, but it also produces better images.

Finally, we answer the main research question of this thesis:

> **RQ:** *What is the best, fastest, and most energy-efficient imaging algorithm for future radio telescopes?*

The IDG algorithm is capable of correcting both DIEs and DDEs during gridding, allowing for high-quality sky images. We efficiently mapping this algorithm onto GPUs, we 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.

## 12.3 Outlook

Now that the gridding and degridding have become so fast, other processing steps have become bottlenecks. After having integrated IDG into WSClean, we measured very significant speedups in overall imaging times. However, the throughput that we achieve in benchmarks (such as the ones shown in the previous chapters) is not achieved in practice. This is a typical case of Amdahl's law, which states that the

theoretical speedup is limited by the part of the execution that is not optimized [89]. Parts of the imaging pipeline (such as deconvolution) used to be negligible in terms of runtime, but have now become noticeable. We would need to conduct end-to-end runtime profiling to identify all performance bottlenecks in the whole processing pipeline and eliminate these bottlenecks one by one.

Moreover, by performing gridding and degridding at these unprecedented speeds, I/O has become a very important bottleneck as well. We generally assume that the input data (e.g., the visibilities) and the output data (the sky image) are read from and stored onto local disk or network storage. Recall that for the fastest GPUs available today, the bandwidth provided by the PCIe bus (which is about 13 GB/s in practice) is hardly sufficient to keep the GPU busy at all times. This implies that data storage should be able to keep up. Even the fastest state-of-the-art NVME SSDs currently available do not provide sufficient bandwidth to make this feasible. Ideally, the processing pipeline should become a real-time streaming pipeline.

In the early days of processor technology performance increases used to be 'for free', CPUs became faster due to an increase in clock speed. At a given moment in time, this trend had reached a plateau and new solutions were needed to keep increasing performance. Similarly, now that we can fully utilize the compute capabilities of GPUs for imaging we have started to run into I/O bottlenecks and need to come up with solutions to improve imaging throughput.

An avenue for future work is to apply Image-Domain gridding to other radio-astronomical applications, most notably for calibration (See also Section 11.4. An initial proof-of-concept has shown that this is feasible, as IDG allows direct fitting for the A-terms (the calibration solutions) in the calibration step. By tightly integrating the processing steps (e.g., calibration and imaging), we might be able to overcome the I/O limitations outlined above. This could provide significant speedups in a full astronomical data-processing pipeline.

High-resolution imaging is challenging because it requires large grids (e.g. 30,000× 30,000 pixels) during gridding and degridding. In Section 6.3.7 we demonstrated that IDG is capable of handling these large grids by using CUDA Unified memory. Furthermore, the long baselines used for high-resolution imaging lead to large W-terms, which make gridding and degridding particularly costly. As we also explained in Section 2.1.1, this is typically remedied by using W-stacking. W-stacking reduces the maximum W-term size by using multiple grids (W-layers) onto which visibilities are gridded based on their $w$-coordinate. Two downsides of this technique are the potentially prohibitive memory consumption (if multiple W-layers are kept in memory)

and additional computations needed to combine the W-layers. We are currently working on an IDG extension called W-tiling, which addresses both these issues. With W-tiling, only a single grid is needed and W-stacking takes place on much smaller *W-tiles*. The initial results with a GPU-accelerated W-tiling implementation show that we can now easily make those $30{,}000 \times 30{,}000$ pixel images.

# Acknowledgements

Now that I am writing this final section of the thesis and my PhD journey has almost finished I would like to take the opportunity to thank all the people that helped me along the way and contributed to making this thesis a reality.

First of all, I would like to thank Wan and Henri, for sparking my interest in high-performance- and parallel computing during my time at the Vrije Universiteit. As my professor, Henri enabled me to pursue a PhD degree to learn much more about these subjects, while also diving deep into the fascinating world of radio astronomy.

Many thanks to John, for being my daily PhD supervisor at ASTRON. He not only taught me all about accelerator programming and optimization but also how to conduct scientific research and write papers.

The first stage of my PhD was conducted during the DOME time, where ASTRON worked together with IBM on research for future-generation computing systems. This was a great time with many nice colleagues. Rik, Erik, Leandro, Przemek, Sanaz, Giovanni, and Li-Ying: you all contributed to creating an inspiring place to conduct research and I really enjoyed all our discussions on work-related and other topics.

Bas deserves special attention, if it wasn't for him, this thesis would likely have turned out very differently. I am very grateful that in the early stage of my PhD, he trusted me with a rough prototype of what we now call Image-Domain Gridding.

Initially, the research focused mainly on getting the algorithm to work correctly and efficiently on CPUs. Matthias, thank you for the nice collaboration at this stage. I could always turn to you with questions about the math behind signal processing, and together we built the infrastructure of the IDG library. After a while, Matthias, Bas, André and I started developing IDG as a high-performance component for the WSClean imager. I am proud of what we have achieved together, as IDG is now used for data-processing of various radio-telescopes around the world.

Many ASTRON colleagues have contributed in one way or another, and I would like to name a few in particular: Tammo Jan, thank you for helping me get to grips with complicated radio-astronomical software packages. Yan and Chris, also after

the DOME colleagues had left, you two always seemed to be around for a coffee or a nice chat. On many occasions, this has either led to new ideas on how to tackle the problems at hand or provided the necessary distraction to keep things fun.

Next, I would like to thank the members of my PhD committee, Ana Lucia, Henk, Kumar, Oleg, and Wan for reading my thesis and providing constructive feedback. The same goes for all the anonymous reviewers that provided valuable feedback to my papers.

During my PhD I had the opportunity to visit several interesting places. The yearly meetings at IBM Zürich are memorable, but especially the visits to the GPU Technology Conference in San Jose were amazing: I learned a lot and had a great time. I would like to thank IBM, ASTRON, and several funding agencies that made this possible.

Especially towards the end, it was sometimes difficult to combine my research and the work on projects. Fortunately, Walter helped me to stay focused and made sure that I had sufficient time to finish writing this thesis.

There are colleagues, collaborators, and family members that I haven't mentioned explicitly yet, but I am grateful for all the help and/or support that you provided during my PhD.

## List of publications

**Image-Domain Gridding on Graphics Processors**
**Veenboer, B.**, Petschow, M., Romein, J. W.
in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), pages 545–554, IEEE, 2017*

**Image Domain Gridding**
van der Tol, S., **Veenboer, B.**, Offringa, A. R.,
In *Astronomy & Astrophysics (A&A), article A28, EDP Sciences, 2018*

**PowerSensor 2: A Fast Power Measurement Tool**
Romein, J. W., **Veenboer, B.**
In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 111-113, IEEE, 2018*

**Radio-Astronomical Imaging: FPGAs vs GPUs**
**Veenboer, B.**, Romein, J. W.
In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), pages 509-521, Springer, 2019* (**best paper award)**

**Precision requirements for interferometric gridding in the analysis of a 21 cm power spectrum**
Offringa, André R., Mertens F., van der Tol, S., **Veenboer, B.**, Gehlot, B. K., Koopmans L. V. E., Mevius M.,
In *Astronomy & Astrophysics (A&A), article A12, EDP Sciences, 2019*

**Estimating continuous direction-dependent gain screens from radio interferometric visibilities and a large skymodel**
van der Tol, S., **Veenboer B.**, Offringa, A. R., Rafferty, D.,Mevius M., Dijkema T. J.,
In *Astronomical Data Analysis Software and Systems (ADASS), 2019*

**Radio-Astronomical Imaging on Graphics Processors**
**Veenboer, B.** Romein, J. W.
In *Astronomy & Computing, Elsevier, Volume 32, July 2020*

**Radio-Astronomical Imaging with WSClean and Image-Domain Gridding**
**Veenboer, B.**, van der Tol, S., Offringa, A. R., Romein, J. W., Dijkema, T. J.,
In *General Assembly and Scientific Symposium (GASS) of the International Union of Radio Science (URSI GASS), 2020*

**Near Memory Acceleration on High Resolution Radio Astronomy Imaging**
Corda, S., **Veenboer, B.**, Awan, A. J., Kumar, A., Jordans, R., Corporaal, H.,
In *Mediterranean Conference on Embedded Computing (MECO), IEEE, 2020*

## References


[1] Sebastiaan van der Tol, Bram Veenboer, and André R. Offringa. Image-Domain Gridding: a fast method for convolutional resampling of visibilities. *Astronomy & Astrophysics*, 616:A27, August 2018.

[2] David Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging*, pages 836–838, May 2008.

[3] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Comput. Sci. Eng.*, 12(1-3):66–73, 2010.

[4] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009.

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

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

[7] 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, January 2013.

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

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

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

[11] Bram Veenboer, Matthias Petschow, and John W. Romein. Image-Domain Gridding on Graphics Processors. In *2017 IEEE International Parallel and Distributed Processing Symposium*, pages 545–554. IEEE, May 2017.

[12] B Veenboer and J.W. Romein. Radio-astronomical imaging on graphics processors. *Astronomy and Computing*, 32:100386, jul 2020.

[13] John W. Romein. An efficient work-distribution strategy for gridding radio-telescope data on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 321–330, June 2012.

[14] ASTRON Netherlands Institute for Radio Astronomy. Image-Domain Gridding. https://gitlab.com/astron-idg/idg, 2019.

[15] Bram Veenboer and John W Romein. Radio-Astronomical Imaging: FPGAs vs GPUs. In *Euro-Par 2019: Parallel Processing*, pages 509–521. Springer International Publishing, 2019.

[16] Wilner J. David. Imaging and deconvolution, May 2014.

[17] The SKA Organisation. SKA Engineering Change Proposal. https://skaoffice.atlassian.net/wiki/display/EP/ECP+Register, 2017.

[18] Oleg M. Smirnov. Revisiting the radio interferometer measurement equation. *Astronomy & Astrophysics*, 531:A159, July 2011.

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

[20] T. J. Cornwell and R. a. Perley. Radio-interferometric imaging of very large fields - The problem of non-coplanar arrays. *Astronomy and Astrophysics*, 261:353–364, 1992.

[21] T. J. Cornwell, Kumar Golap, and Sanjay Bhatnagar. The Non-coplanar Baselines Effect in Radio Interferometry: The W-projection Algorithm. *IEEE J. Sel. Topics Signal Process.*, 2(5):647–657, October 2008.

[22] U. Rau et al. Advances in Calibration and Imaging Techniques in Radio Interferometry. *IEEE Proceedings*, 97:1472–1481, August 2009.

[23] U. Rau and T. J. Cornwell. A multi-scale multi-frequency deconvolution algorithm for synthesis imaging in radio interferometry. *Astronomy & Astrophysics*, 532:A71, aug 2011.

[24] F. Li, T. J. Cornwell, and F. de Hoog. The application of compressive sampling to radio astronomy. *Astronomy & Astrophysics*, 528:A31, apr 2011.

[25] J.N. Girard, H. Garsden, J.L. Starck, S. Corbel, A. Woiselle, C. Tasse, J.P. McKean, and J. Bobin. Sparse representations and convex optimization as tools for LOFAR radio interferometric imaging. *Journal of Instrumentation*, 10(08):C08013–C08013, aug 2015.

[26] A. Dabbech, C. Ferrari, D. Mary, E. Slezak, O. Smirnov, and J. S. Kenyon. MORESANE: MOdel REconstruction by Synthesis-ANalysis Estimators. *Astronomy & Astrophysics*, 576:A7, apr 2015.

[27] H. Junklewitz, M. R. Bell, M. Selig, and T. A. Enßlin. RESOLVE: A new algorithm for aperture synthesis imaging of extended emission in radio astronomy. *Astronomy & Astrophysics*, 586:A76, feb 2016.

[28] A. R. Offringa and O. Smirnov. An optimized algorithm for multiscale wideband deconvolution of radio astronomical images. *Monthly Notices of the Royal Astronomical Society*, 471(1):301–316, oct 2017.

[29] A. Scaife. SDP Memo: The SDP imaging pipeline. Technical report, SKA Science Data Processor Consortium, 2016.

[30] T. J. Cornwell, M. A. Voronkov, and B. Humphreys. Wide field imaging for the Square Kilometre Array. *Proc. SPIE*, 8500, August 2012.

[31] 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):606–619, August 2014.

[32] C. Tasse et al. Faceting for direction-dependent spectral deconvolution. *Astronomy & Astrophysics*, 611:A87, March 2018.

[33] S. Jaeger. The Common Astronomy Software Application (CASA). In R. W. Argyle, P. S. Bunclark, and J. R. Lewis, editors, *Astronomical Data Analysis Software and Systems XVII*, volume 394 of *ASP Conference Series*, pages 623–627, August 2008.

[34] R. Nijboer et al. Parametric models of SDP compute requirements. Technical report, ASTRON Netherlands Institute for Radio Astronomy, 2015. SKA SDP PDR deliverable.

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

[36] The SKA Organisation. SDP parametric model. https://gitlab.com/ska-telescope/sdp-par-model, 2019.

[37] R. Braun et al. SKA1 science priority outcomes. Technical report, ASTRON Netherlands Institute for Radio Astronomy, 2014. SKA design document.

[38] R. Braun et al. SKA1 Level 0 Science Requirements. Technical report, The SKA Organisation, 2015.

[39] Carole Jackson. SKA Science: A Parameter Space Analysis. Technical report, SKA Consortium, 2003.

[40] Gordon E. Moore. Cramming more components onto integrated circuits. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, sep 2006.

[41] Chris A. Mack. Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, may 2011.

[42] Daniel A. Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, jun 2015.

[43] Balaji Subramaniam, Winston Saunders, Tom Scogland, and Wu Chun Feng. Trends in energy-efficient computing: A perspective from the Green500. In *2013 International Green Computing Conference Proceedings, IGCC 2013*, pages 1–8. IEEE, jun 2013.

[44] Sushant Sharma, Chung-Hsing Hsu, and Wu-chun Feng. Making a case for a Green500 list. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, volume 2006, page 8 pp. IEEE, 2006.

[45] Sandra Wienke and Sridutt Bhalachandra. *Accelerator Programming Using Directives*, volume 12017 of *Lecture Notes in Computer Science*. Springer International Publishing, 2020.

[46] Justin W. Richardson, Alan D. George, and Herman Lam. Performance Analysis of GPU Accelerators with Realizable Utilization of Computational Density. In *2012 Symposium on Application Accelerators in High Performance Computing*, pages 137–140. IEEE, jul 2012.

[47] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. *Proceedings of the International Conference on Parallel Processing*, pages 207–216, 2010.

[48] John W. Romein and Bram Veenboer. PowerSensor 2: a Fast Power Measurement Tool. *2018 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 111–113, 2018.

[49] Przemyslaw Lenkiewicz, P. Chris Broekema, and Bernard Metzler. Energy-efficient data transfers in radio astronomy with software UDP RDMA. *Future Generation Computer Systems*, March 2017.

[50] John W. Romein. A Comparison of Accelerator Architectures for Radio-Astronomical Signal-Processing Algorithms. In *International Conference on Parallel Processing 2016*, pages 484–489, August 2016.

[51] ASTRON Netherlands Institute for Radio Astronomy. PowerSensor. https://gitlab.com/astron-misc/PowerSensor, 2016.

[52] J. H. Laros III, P. Pokorny, and D. DeBonis. PowerInsight – A Commodity Power Measurement Capability. In *Int. Workshop on Power Measurement and Profiling*, Arlington, VA, June 2013.

[53] Daniel Bedard, Min Yeol Lim, Robert Fowler, and Allan Poterfield. PowerMon: Fine-grained and Integrated Power Monitoring for Commodity Computer Systems. In *Proc. of the IEEE SoutheastCon*, pages 479–484, March 2010.

[54] Thomas Ilsche, Daniel Hackenberg, Stefan Graul, Joseph Schuchart, and Robert Schöne. Power Measurements for Compute Nodes: Improving Sampling Rates, Granularity and Accuracy. In *2015 Sixth International Green and Sustainable Computing Conference*, pages 1–8, December 2015.

[55] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink. How a Single Chip Causes Massive Power Bills GPUSimPow: A GPGPU Power Simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 97–106, April 2013.

[56] Francisco D. Igual et al. A Power Measurement Environment for PCIe Accelerators: Application to the Intel Xeon Phi. *Computer Science - Research and Development*, 30(2):115–124, May 2015.

[57] Ron Bracewell. *The Fourier Transform and Its Applications*. McGraw Hill, 1965.

[58] Anthony Griffin and Andrew Ensor. End-to-end Modelling of the Imaging Pipeline in Radio Astronomy. In *2018 IEEE 10th Sensor Array and Multichannel Signal Processing Workshop (SAM)*, volume 8, pages 480–484. IEEE, jul 2018.

[59] Anthony Griffin and Andrew Ensor. SDP Memo: Numerical Precision. Technical report, SKA Science Data Processor Consortium, 2018.

[60] Stefano Salvini. SDP Memo: On the Precision Required in SDP Pipelines. Technical report, SKA Science Data Processor Consortium, 2018.

[61] Christoph 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*, pages 407–411. IEEE, November 2016.

[62] Henri Bal et al. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *IEEE Computer*, 49(5):54–63, May 2016.

[63] Y.N. Srikant and P. Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2018.

[64] Bruce Merry. Faster GPU-based convolutional gridding via thread coarsening. *Astron. Comput.*, 16:140–145, July 2016.

[65] Ben van Werkhoven. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems*, 90:347–358, January 2019.

[66] S.F. Oberman and M.Y. Siu. A High-Performance Area-Efficient Multifunction Interpolator. In *17th IEEE Symposium on Computer Arithmetic*, pages 272–279, June 2005.

[67] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2018.

[68] Michal Drobot. Low Level Optimizations for GCN, May 2014.

[69] S'ebastien Lagarde. Inverse trigonometric functions GPU optimization for AMD GCN architecture. https://seblagarde.wordpress.com/2014/12/01/inverse-trigonometric-functions-gpu-optimization-for-amd-gcn-architecture, December 2014.

[70] NVIDIA Corporation. NVIDIA GeForce GTX 1080 Whitepaper. Technical report, NVIDIA Corporation, 2016.

[71] Stephen Jones. CUDA New Features And Beyond, 2019.

[72] Jülich Supercomputing Centre. JURON (IBM-NVIDIA pilot). https://hbp-hpc-platform.fz-juelich.de/?page_id=1073, 2016.

[73] Byron Sinclair, Andrew C. Ling, and Genady Paikin. Harnessing the Power of FPGAs with the Intel FPGA SDK for OpenCL™. In *IWOCL 2017 Proceedings of the 5th International Workshop on OpenCL*, pages 1–1, 05 2017.

[74] Hamid Reza Zohouri et al. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420, nov 2016.

[75] Hamid Reza Zohouri. *High Performance Computing with FPGAs and OpenCL*. PhD thesis, Tokyo Institute of Technology, 2018.

[76] Jason Cong et al. Understanding Performance Differences of FPGAs and GPUs. In *2018 IEEE 26th International Symposium on Field-Programmable Custom Computing Machines*, pages 93–96, apr 2018.

[77] Zheming Jin and Hal Finkel. Power and Performance Tradeoff of a Floating-Point Intensive Kernel on OpenCL FPGA Platform. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 716–720, may 2018.

[78] Fahad Bin Muslim et al. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. *IEEE Access*, 5:2747–2762, 2017.

[79] Umar Ibrahim Minhas et al. Exploring Functional Acceleration of OpenCL on FPGAs and GPUs Through Platform-Independent Optimizations. In *14th International Symposium, ARC 2018*, pages 551–563, 2018.

[80] ASTRON Netherlands Institute for Radio Astronomy. AW-Projection Gridding. https://gitlab.com/astron-misc/wprojection, 2019.

[81] Bruce Merry. Spectral-line imager for MeerKAT. https://github.com/ska-sa/katsdpimager, 2020.

[82] The SKA Organisation. Designing the Square Kilometre Array. https://cdr.skatelescope.org, 2018.

[83] Alexander P. et al. SDP System Module Decomposition and Dependency View. Technical report, SKA Science Data Processor Consortium, 2015.

[84] B. Veenboer, S. van der Tol, A. R. Offringa, J. W. Romein, and T. J. Dijkema. Radio-Astronomical Imaging with WSClean and Image-Domain Gridding. In *URSI GASS*, pages 3–6, sep 2020.

[85] N. J. Jackson, S. Badole, J. Morgan, R. Chhetri, K. Prusis, A. Nikolajevs, L. Morabito, and Et Al. Sub-arcsecond imaging with the International LOFAR Telescope. II. Completion of the LOFAR Long-Baseline Calibrator Survey. *Astronomy & Astrophysics*, pages 1–14, may 2021.

[86] A. R. Offringa, F. Mertens, S. van der Tol, B. Veenboer, B. K. Gehlot, L. V. E. Koopmans, and M. Mevius. Precision requirements for interferometric gridding in the analysis of a 21 cm power spectrum. *Astronomy & Astrophysics*, 631:A12, nov 2019.

[87] S. Kazemi, S. Yatawatta, S. Zaroubi, P. Lampropoulos, A. G. de Bruyn, L. V. E. Koopmans, and J. Noordam. Radio interferometric calibration using the SAGE algorithm. *Monthly Notices of the Royal Astronomical Society*, 414(2):1656–1666, jun 2011.

[88] S. van der Tol, Veenboer B., A. R. Offringa, D. Rafferty, Mevius M., and Dijkema T. J. Estimating continuous direction-dependent gain screens from radio interferometric visibilities and a large skymodel. In *Astronomical Data Analysis Software and Systems (ADASS)*, 2019.

[89] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, july 2008.