Radio-Astronomical Imaging: FPGAs vs GPUs

Bram Veenboer and John W. Romein

ASTRON (Netherlands Institute for Radio Astronomy)
{veenboer, romein}@astron.nl

Abstract. 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 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 paper, we show how we implemented and optimized a radio-astronomical imaging application 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. All together, we demonstrate that OpenCL support for FPGAs is a leap forward in programmability and it enabled us to use an FPGA as a viable accelerator platform for a complex HPC application.

1 Introduction

Field-Programmable Gate Arrays (FPGAs) have long been favoured as energy-efficient platform for fixed-precision computations. Their floating-point performance used to be sub-par, because floating-point units (FPUs) had to be assembled from logic blocks, which is rather inefficient and consumes many FPGA resources. Recent FPGAs, such as the Intel Arria 10, have hardware support for floating-point operations, making them an interesting platform for high-performance floating-point computing.

FPGAs are traditionally programmed using hardware description languages, such as Verilog and VHDL, which is notoriously difficult, time-consuming, and error-prone. FPGA manufacturers such as Intel (formerly Altera) and Xilinx now support OpenCL as a high-level alternative. In this paper, we describe how we use the Intel FPGA SDK for OpenCL to implement and optimize a complex radio-astronomy imaging application for the Arria 10 FPGA, which would have been a daunting task when using a hardware description language. Radio-astronomical imaging is a computationally challenging problem and poses strict performance and energy-efficiency requirements, especially for future exa-scale instruments such as the Square Kilometre Array (SKA). We previously

demonstrated that imaging works particularly well on GPUs [11], so how does the FPGA perform in comparison?

The main contributions of this paper are: (1) We explain how we use the Intel FPGA SDK for OpenCL to build an efficient data-flow network for a complex radio-astronomy application; (2) We compare our implementation on the Arria 10 FPGA to highly optimized CPU and GPU implementations and evaluate performance and energy efficiency; (3) We discuss the differences and similarities between FPGAs and GPUs in terms of architecture, programming model, and implementation effort.

The rest of this paper is organized as follows: Section 2 provides background information on radio-astronomical imaging. Section 3 explains how we implemented and optimized the most critical parts of an astronomical imaging application. In Section 4 we analyze performance and show energy efficiency measurements. Section 5 describes the lessons that we learned while implementing and optimizing the same application for both FPGAs and GPUs. In Section 6 we discuss related work and we conclude in Section 7.

The source code of the FPGA implementations discussed in this paper is available online [1].

2 Radio-astronomical imaging

A radio telescope detects electromagnetic waves that originate from radio sources in the universe, which are used to construct a map of the sky containing the positions, intensity, and polarization of the sources. Radio telescopes (such as LOFAR and the future SKA-1 Low telescope) comprise many receivers of which the signals are combined using a technique called 'interferometry'. Figure 1 shows a simplified version of a radio-astronomical interferometer, where sky-images are created in three steps: correlation, calibration, and imaging. Every receiver measures two signals, corresponding to two orthogonal polarizations. The signals from a receiver pair (q, r) (called a *baseline*) are multiplied and integrated for a short period of time (correlated) such that the resulting sample $V_{(q,r)}$ (called a *visibility*) contains the 2×2 combinations of the (polarized) signals measured by receiver q and r, hence $V_{(q,r)} \in \mathbb{C}^{2 \times 2}$. Visibilities have associated (u, v, w)-coordinates that depend on the location of the receivers with respect to the observed sky. Due to earth rotation, (u, v, w)-coordinates change over time and every baseline contributes a track of measurements. During an observation, each baseline collects T_{Obs} integration periods, where every sample consists of C_{Obs} measurements in frequency. There exists a Fourier relation between the sampled data and the observed sky. Therefore, in the *imaging* step, visibilities are first placed onto a regular grid by an operation called *gridding*. This operation corresponds to applying a convolution to every visibility. After gridding, the grid is Fourier transformed to obtain a *sky image*. *Degridding* is the reverse operation where visibilities are computed taking a grid as input.

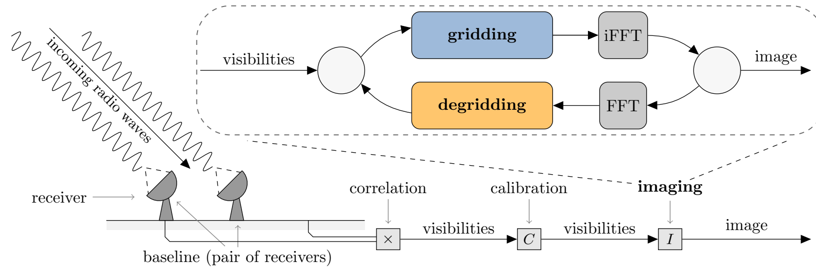


Fig. 1: In a radio-telescope, signals are received by pairs of *receivers*. The *correlator* combines the signal into *visibilities*. After *calibration* of the visibilities, the *imager* produces an image of the sky, using an *imaging pipeline*.

2.1 Image-Domain Gridding

Image-Domain Gridding (IDG [10, 11]) is a novel imaging technique where neighbouring visibilities are first gridded onto so-called *subgrids*, after which the subgrids are Fourier transformed and added to the full grid. Subgrids are $N{\times}N$ pixels in size and are positioned such that they cover T integration periods (each with C frequency channels) and their corresponding convolution kernels. Algorithm 1 shows pseudocode for gridding.

By applying gridding in the *image-domain*, IDG avoids the use of convolution kernels in traditional gridding. Furthermore, the computation of one subgrid (one iteration of the loop on Line 2) is not dependent on the computation of another subgrid, making IDG very suitable for parallelization. We will refer to Line 4 through Line 15 as the *gridder*. After this step, a-term correction, tapering and a 2D FFT are applied. We will refer to these operations as *post-processing*.

Pixels of the subgrid are computed as a direct sum of phase-shifted visibilities [10]. This shift takes both the position of the subgrid (the *phase offset*, Line 5) and the position of the visibility in the subgrid (the *phase index*, Line 7) into account. Furthermore, the phase index is scaled according to frequency (Line 9).

The *phasor* term in Line 11 is a complex number that is computed by an evaluation of $cos(phase)$ and $sin(phase)$ or in more common terms $cis(phase)$ where $cis(x) = cos(x) + isin(x)$. *cmul* denotes a complex multiplication, which comprises four real-valued multiply-add operations. Since $P = 4$, the loop on Line 13 is typically unrolled. Thus for every iteration of the loop over frequency channels in line 8, one sine, one cosine, and 17 multiply-add operations are performed, one in the computation of *phase* in Line 10, and 16 in the complex multiplication of *phasor* with visibilities and addition to the subgrid in Line 15.

The operations outside this critical loop (the *offset* computation on Line 5, the index computation on Line 7, and post-processing steps) are described by van der Tol et al. [10]. The grid can be several tens of GBs in size and is therefore typically stored on a CPU-based system, while the computationally most

```
1  #pragma parallel
2  for s = 1...S :
3  |  complex<float> subgrid[P][N×N];
4  |  for i = 1...N×N :
5  |  |  float offset = compute_offset(s, i);
6  |  |  for t = 1...T :
7  |  |  |  float index = compute_index(s, i, t);
8  |  |  |  for c = 1...C :
9  |  |  |  |  float scale = scales[c];
10 |  |  |  |  float phase = offset - (index × scale);
11 |  |  |  |  complex<float> phasor = {cos(phase), sin(phase)};
12 |  |  |  |  #pragma unroll
13 |  |  |  |  for p = 1...P : // 4 polarizations
14 |  |  |  |  |  complex<float> visibility = visibilities[t][c][p];
15 |  |  |  |  |  subgrid[p][i] += cmul(phasor, visibility);
16 |  apply_aterm(subgrid);
17 |  apply_taper(subgrid);
18 |  apply_ifft(subgrid);
19 |  store(subgrid);
```

Algorithm 1: Gridding pseudocode that is executed for every subgrid s of $N \times N$ pixels in size. $T \times C$ visibilities are associated with a subgrid, where T and C denote time and frequency channel, respectively. Typical values for these parameters are $N = 32$, $T = 128$ and $C = 16$.

challenging gridding step is preferably performed on an accelerator (such as a FPGA or a GPU).

3 Implementation

As we discuss in more detail later, 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 2. 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, 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 2.

To implement gridding on the FPGA, we applied the following changes to Algorithm 1: (1) we create a *gridder pipeline* that executes Line 5 through Line 15 to compute a single subgrid; (2) we move the computation of the *index* value (Line 7) and the computation of *offset* (Line 5) into separate kernels to avoid underutilization of the DSPs used to implement these computations; (3) we unroll the loop over pixels (Line 4) to increase reuse of input data; (4) we replicate the gridder pipeline by a factor ϕ to compute multiple subgrids in parallel; (5) input data (such as the visibilities, Line 14) is read from DRAM in bursts in separate kernels and forwarded to the gridder pipelines in a round-robin fashion.

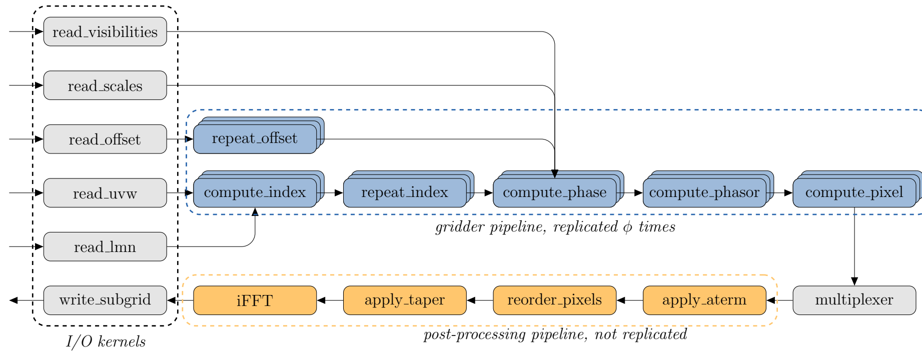


Fig. 2: All kernels in this design are single work-item kernels. The majority of the computation takes place in the gridder pipeline, which is replicated ϕ times to compute multiple subgrids in parallel. These subgrids are multiplexed and passed to the post-processing pipeline, which applies a-term correction,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

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 16) is implemented as a series of two complex 2×2 matrix multiplications (one correction matrix per receiver). Tapering (Line 17) is implemented as a scalar multiplication to every pixel in the subgrid. The 2D FFT (Line 18) 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.

3.1 The sine/cosine computations

The Intel FPGA OpenCL compiler recognizes the sine and cosine pair and uses 8 memory blocks and 8 DSPs to implement it by creating a so-called *IP block* (cis_{ip}). In comparison, only a single DSP is used to compute the *phase* term on Line 10, and 16 DSPs are used to implement the computation on Line 15. 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.

3.2 Optimizing for frequency

The OpenCL FPGA compiler gives feedback on resource usage by generating HTML reports, which is highly useful when optimizing for 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 therefore 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.

4 Results

We compare our gridding and degridding design on an Arria 10 FPGA to a GPU in terms of performance and energy efficiency. We also add an optimized CPU implementation for comparison. We use contemporary devices with a similar theoretical peak performance and produced using a similar lithographical process, see Table 1 for details. The imaging parameters are set as follows: $N = 32$, $T = 128$ and $C = 16$. The FPGA designs are scaled up by increasing ϕ until the maximum number of DSPs is reached.

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 so-called Board-Support Package (BSP) that is required to use the FPGA using the Intel FPGA SDK for OpenCL. 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.

The CPU that we use is part of a dual-socket system, of which we use only a single processor (HASWELL) and the corresponding memory. We use an Intel compiler and the Intel Math Kernel Library (MKL) (both version 2019.0). The GPU (MAXWELL) uses the 396.26 GPU driver and CUDA version 9.2.88.

4.1 Resource usage

We refer to designs that use cis_{ip} 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 the highest achieved clock frequency (F_{max}) of all designs in Table 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). We provide a

Table 1: The Intel Haswell-EP CPU, Intel Arria 10 FPGA and NVIDIA Maxwell GPU used in our experiments. We refer to these devices as Haswell, Arria and Maxwell.

	# FPUs	Peak	Bandwidth	TDP	Process
Intel Xeon E5-2697v3	224[a)]	1.39 TFlop/s	68 GB/s	145W	22nm (TSMC)
Nallatech 385A	1518	1.37 TFlop/s	34 GB/s	75W	20nm (TSMC)
NVidia GTX 750 Ti	640	1.39 TFlop/s	88 GB/s	60W	28nm (TSMC)

[a)] # cores × # vector units × vector length

Table 2: Resource usage of our gridding and degridding designs on Arria. Logic (ALUTs or FFs) is counted in terms of thousand elements. The ϕ parameter is used to scale up the design, see Fig. 2. The theoretical peak F_{max} of Arria is 450 Mhz.

	ALUTs	FFs	RAMs	DSPs	MLABs	ϕ	F_{max}
gridding-ip	334 (43%)	487 (31%)	1514 (64%)	1439 (95%)	5317 (71%)	14	258
degridding-ip	364 (47%)	550 (35%)	1711 (72%)	1441 (95%)	6418 (78%)	14	254
gridding-lu	207 (27%)	490 (32%)	1448 (61%)	1498 (99%)	5921 (57%)	20	256
degridding-lu	252 (33%)	583 (38%)	1723 (73%)	1503 (99%)	7520 (69%)	20	253

breakdown of DSP resource usage in Figure 3 where we distinguish between the DSPs used to implement various subparts of the algorithm. E.g. for gridding (Algorithm 1): $\texttt{DSP}_{\texttt{fma}}$ (Line 15), $\texttt{DSP}_{\texttt{cis}}$ (Line 11) and $\texttt{DSP}_{\texttt{misc}}$ for the post-processing steps and miscellaneous computations, and similarly for degridding. The implementation of computations outside of the critical path consume few resources ($\texttt{DSP}_{\texttt{misc}}$). Since cis_{lu} uses fewer resources compared to cis_{ip} to implement the sine/cosine evaluation, we are able to scale up gridding-lu and degridding-lu further (by increasing ϕ from 14 to 20) than is possible with gridding-ip and degridding-ip.

4.2 Throughput and energy efficiency

We compare throughput, measured as the number of visibilities processed per second, in Figure 4a. 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 parallel gridder or degridder pipelines. 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. In Figure 4a we therefore only include the kernel runtime to determine throughput.

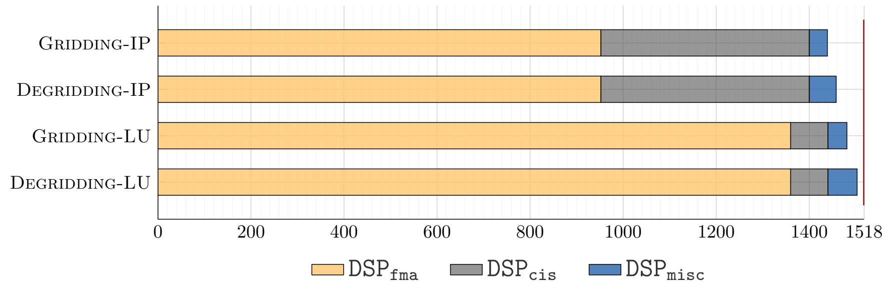
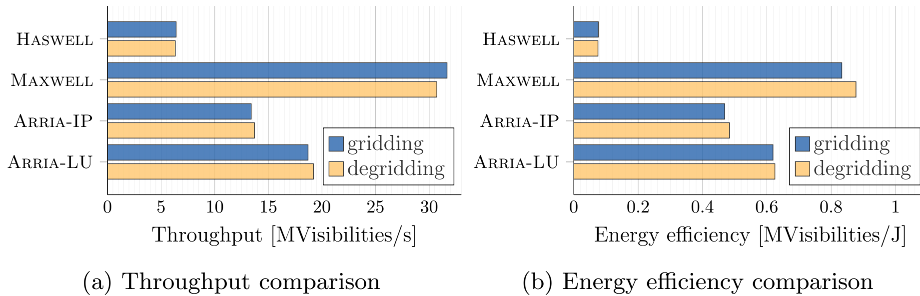


Fig. 3: Breakdown of DSP resource usage



(a) Throughput comparison (b) Energy efficiency comparison

Fig. 4: Throughput (the number of visibilities processed per second, MVis/s) and energy efficiency (the number of visibilities processed per Joule, MVis/J).

To asses energy-efficiency, we use PowerSensor [8] to measure energy consumption of the full PCIe device in case of ARRIA. On MAXWELL we use NVML and on HASWELL we use LIKWID [9]. Our measurements in Figure 4b indicate that both accelerators are much more energy-efficient then HASWELL by processing about an order of magnitude more visibilities for every Joule consumed.

4.3 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 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* by issuing instructions onto the FPUs. In the operations mix found in IDG (17 FMAs and one evaluation of $cis(x)$) 80% of the time is spent in the sine/cosine evaluation [11]. 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

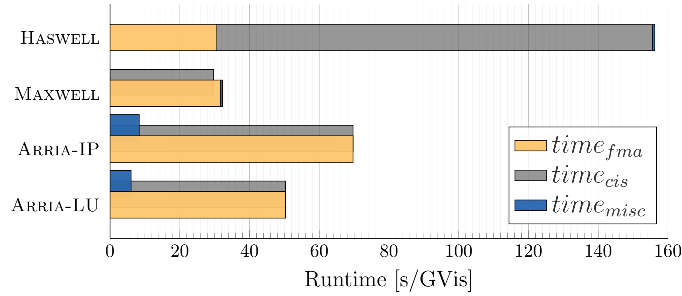


Fig. 5: Breakdown of gridding runtime for FMA operations ($time_{fma}$), sine/cosine evaluations ($time_{cis}$) and all other operations ($time_{misc}$). On Haswell, 80% of the time is spend in sine/cosine evaluations. On Maxwell and on Arria, the sine/cosine evaluations are performed concurrently with the FMA operations.

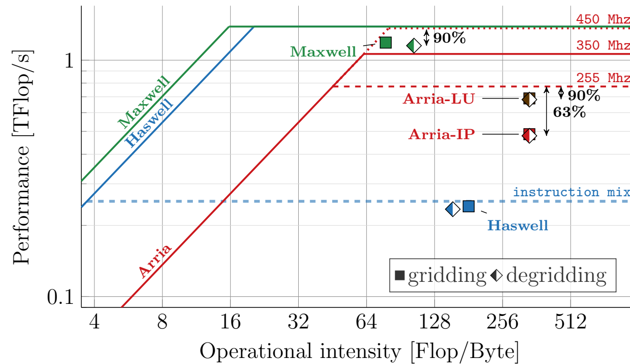


Fig. 6: The implementation of sine/cosine evaluations in software imposes an upper bound on performance on Haswell. Maxwell performs sine/cosine operations concurrently with FMA operations and performs close to the theoretical peak. On Arria, the performance is bound by the clock frequency.

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 [12], see Figure 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 5, on Haswell the *Flops* and *Ops* are both executed on the FPUs and the performance is therefore bound by the performance of the $cis(x)$ implementation, e.g. Intel MKL (for which the bound is indicated with the blue dashed line). A lookup table does not improve performance over using

the Intel MKL library. Due to the SFUs, on Maxwell the achieved performance is over 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. Our gridding and degridding designs on average achieve about 255 Mhz (indicated with the red 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 3) provides upper bounds on attainable performance. The achieved performance is within 99% of these bounds, indicating that the designs are nearly stall-free.

5 FPGAs vs. GPUs: 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, and performance.

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 GPU code is imperative. 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. In fact, 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 differently. 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 are necessary optimizations on both ar-

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

6 Related work, discussion and future work

Licht et al. [4] present an overview of HLS FPGA code transformations such as transposing of the iteration space, replication and streaming dataflow that we also applied. However, they do not describe code transformation for overcoming underutilization of resources. Yang et al. [14] address underutilization of resources by using a consumer-producer model, which they implement using channel arbitrage. We also connect kernels running at different rates using channels, but we use channel depth to facilitate buffering and to avoid stalls.

Several studies compare energy efficiency between OpenCL applications for FPGAs and GPUs [16, 15, 3, 5, 7, 6]. 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 apply the roofline model to illustrate 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 are bound by clock frequency, something we can not improve with the current OpenCL compiler for FPGAs. We also explain that the GPU has an advantage, by computing sine/cosine using dedicated hardware. In contrast to what the related work suggests, our results indicate that FPGAs are not necessarily more energy-efficient than GPUs.

Intel claims that the Stratix 10 FPGA (produced at 14nm) will be about 3.6× as energy-efficient compared to Arria 10 [13] and have a peak performance of up to 9 TFlop/s. In future work, we would like to extend our analysis to compare Stratix 10 and NVIDIA Turing GPUs.

7 Conclusion

In this paper we set out to implement a complex radio-astronomy application on an Arria 10 FPGA using the Intel FPGA SDK for OpenCL. Being able to implement such an application 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. We show optimization techniques that make our implementation very scalable as it uses almost all DSPs available to perform useful floating-point computations while it stalls less than 1% of the time.

We compared optimized implementations of an astronomical imaging application on a GPU, FPGA, and a CPU. While the theoretical peak-performance for these devices is almost identical, the FPGA and GPU perform much better than the CPU and they consume significantly less power. In absolute terms, the GPU is the fastest and most energy-efficient device, mainly due to support for sine/cosine operations using dedicated hardware. On the FPGA, our implementation of a custom lookup-table for these operations is advantageous, but the maximum achieved clock frequency is only about 70% of the theoretical peak. Unfortunately, the Intel FPGA SDK for OpenCL (currently) provides few means to improve the clock frequency. This issue is non-existent on GPUs.

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.

Acknowledgments

This work is funded by the Netherlands eScience Center (NLeSC), under grant no 027.016.G07 (Triple-A 2), the EU Horizon 2020 research and innovation programme under grant no 754304 (DEEP-EST) and by NWO (DAS-5 [2]). The European Commission is not liable for any use that might be made of the information contained in this paper. The authors would like to thank Atze van der Ploeg (NLeSC) and Suleyman S. Demirsoy (Intel) for their support.

References

1. ASTRON Netherlands Institute for Radio Astronomy: Image-Domain Gridding for FPGAs. GitLab: astron-idg/idg-fpga (2019)
2. Bal, H., et al.: A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. IEEE Computer 49(5), 54–63 (2016)
3. Cong, J., et al.: Understanding Performance Differences of FPGAs and GPUs. In: 2018 IEEE 26th International Symposium on Field-Programmable Custom Computing Machines, pp. 93–96 (2018)
4. de Fine Licht, J., et al.: Transformations of High-Level Synthesis Codes for High-Performance Computing. Computing Research Repository (CoRR) (2018)

5. Jin, Z., Finkel, H.: Power and Performance Tradeoff of a Floating-Point Intensive Kernel on OpenCL FPGA Platform. pp. 716–720 (2018)
6. Minhas, U.I., et al.: Exploring Functional Acceleration of OpenCL on FPGAs and GPUs Through Platform-Independent Optimizations. In: 14th International Symposium, ARC 2018, pp. 551–563 (2018)
7. Muslim, F.B., et al.: Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. IEEE Access 5, 2747–2762 (2017)
8. Romein, J.W., Veenboer, B.: PowerSensor 2: a Fast Power Measurement Tool. 2018 IEEE International Symposium on Performance Analysis of Systems and Software pp. 111–113 (2018)
9. Treibig, J., Hager, G., Wellein, G.: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. Proceedings of the International Conference on Parallel Processing pp. 207–216 (2010)
10. van der Tol, S., Veenboer, B., Offringa, A.: Image Domain Gridding. Astronomy & Astrophysics 616 (2018)
11. Veenboer, B., Petschow, M., Romein, J.W.: Image-Domain Gridding on Graphics Processors. Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS pp. 545–554 (2017)
12. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM 52, 65–76 (2009)
13. Won, M.S.: Meeting the Performance and Power Imperative of the Zettabyte Era with Generation 10. Tech. rep., Intel Programmable Solutions Group (2013)
14. Yang, C., et al.: OpenCL for HPC with FPGAs: Case study in molecular electrostatics. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–8 (2017)
15. Zohouri, H.R.: High Performance Computing with FPGAs and OpenCL. Ph.D. thesis, Tokyo Institute of Technology (2018)
16. Zohouri, H.R., 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, pp. 409–420 (2016)