



# Pulsar search acceleration using FPGAs and OpenCL templates

Julian Oppermann<sup>1</sup> · Mitchell B. Mickaliger<sup>2</sup> · Oliver Sinnen<sup>3</sup>

Received: 17 August 2021 / Accepted: 13 December 2022  
© The Author(s) 2023

## Abstract

The Square Kilometre Array (SKA) is the world's largest radio telescope currently under construction, and will employ elaborate signal processing to detect new pulsars, i.e. highly magnetised rotating neutron stars. This paper addresses the acceleration of demanding computations for this pulsar search on Field-Programmable Gate Arrays (FPGAs) using a new high-level design process based on OpenCL templates that is transferable to other scientific problems. The successful FPGA acceleration of large-scale scientific workloads requires custom architectures that fully exploit the parallel computing capabilities of modern reconfigurable hardware and are amenable to substantial design space exploration. OpenCL-based high-level synthesis toolchains, with their ability to express interconnected multi-kernel pipelines in a single source language, excel in this domain. However, the achievable performance strongly depends on how well the compiler can infer desirable hardware structures from the code. One key aspect to excellent performance is commonly the uninterrupted, high-bandwidth streaming of data into and through the design. This is difficult to achieve in complex designs when data order needs to be re-arranged, e.g. transposed. It is equally hard to pre-fetch and burst-load from DDR memory when reading occurs in non-trivial patterns. In this paper, we propose new approaches to these two problems that use OpenCL-based code templates.

We demonstrate the practical benefits of these approaches with the acceleration of a key component in the SKA's pulsar search pipeline: the Fourier Domain Acceleration Search (FDAS) module. Using our proposed methodology, we are able to develop a more scalable FDAS accelerator architecture than previously possible. We explore its design space to eventually achieve a 10x throughput improvement over a prior, thoroughly optimised implementation in plain OpenCL.

---

Research was conducted while J. Oppermann was employed at the University of Auckland, New Zealand

---

✉ Julian Oppermann  
[oppermann@esa.tu-darmstadt.de](mailto:oppermann@esa.tu-darmstadt.de)

Extended author information available on the last page of the article.

**Keywords** Radio astronomy · Pulsar search · FPGA · OpenCL · High-level synthesis · Template code generation

## 1 Introduction and related work

The design of FPGA-based accelerators for real-world scientific problems using OpenCL has received an increasing amount of academic attention over the last years. Examples include cancer treatment simulations [20], convolutional neural networks [12], accelerating matrix multiplications [4], solving Maxwell's equations [9], and general stencil computations [21], to mention only the most recent publications – refer to [7, 15] for a broader overview. This is no surprise, as OpenCL-based high-level synthesis toolchains, with their ability to express interconnected multi-kernel pipelines in a single source language, excel in this domain.

This trend coincides with general maturing of the Intel toolchain, but also with the introduction of FPGA-specific extensions to the base OpenCL language. It is well recognised today that OpenCL is functionally portable between different target architectures, but usually not performance portable. In order to obtain very good performance on FPGAs, OpenCL designs need to be developed with a hardware architecture in mind, in other words FPGA-centric.

One of the most profound differences to CPU and GPU programming is the concept of the *single work-item* (SWI) kernel, which practically abandons the standard's data-parallel execution model, defined in terms of work-items and work-groups, for the automatic formation of computational pipelines from loops, a classic high-level synthesis technique. Using SWI kernels is a recommendation in Intel's Best Practices Guide [6], one of the first steps in the optimisation framework proposed by Sanaullah et al. [16], and the default choice in the analysis tool presented by Jiang et al. [7]. Other FPGA-specific idioms are the inference of shift registers [21], and influencing the banking of local memory buffers with the help of attributes [6], or better by statically separating arrays in the source code [16].

Relying on low-latency *channels* (or pipes), instead of communicating via global memory, is the second outstanding concept in FPGA-centric OpenCL programming. In principle, this enables the single-source specification of systolic arrays and other multi-stage pipelines, and therefore plays well into the FPGA's key strength, laying out computations spatially.

Large scientific applications, as targeted in our case study in this paper, require custom architectures that fully exploit the parallel computing capabilities of modern FPGAs and are amenable to substantial design-space exploration, especially if adaptability across multiple FPGA generations is desired. To that end, we employ an interconnected multi-kernel pipeline architecture, expressed exclusively as SWI kernels in OpenCL.

However, the language support to actually form such pipelines from multiple kernels is lacking in Intel's toolchain. The parameterisation of an architecture can lead to code that is either inefficient to synthesise, because optimisation opportunities might be obscured for the compiler, or cumbersome to write and maintain, due to the necessary manual code duplication and specialisation. SWI kernels may be duplicated with

a set of vendor-specific attributes [21], but cannot take arguments or access the global memory then. Some authors have resorted to workarounds, such as unrolling arrays of processing elements inside a single kernel, which can be challenging to get right [20], or by using preprocessor macros to wrap individual kernel declarations around the actual algorithm [4]. We propose to prepend a lightweight code-generation step to the OpenCL synthesis flow: The design is specified using OpenCL *templates*, i.e. kernel code augmented with snippets of Python code. We use a template engine to produce standard OpenCL code, specialised to a concrete set of parameter values, which is then processed by the Intel toolchain. Our approach is more capable than the “knobs” used to generate variants of the benchmark kernels in the suite presented by Gautier et al. [3], but more lightweight than the full-fledged code generation from an application-class-specific intermediate representation as proposed by Mu et al. [12].

With the large computing resources of modern devices, one key aspect to excellent performance is commonly the uninterrupted, high-bandwidth streaming of data into and through the design. This is difficult to achieve in complex, scientific designs, when data order needs to be re-arranged, e.g. transposed or reversed. It is equally hard to pre-fetch and burst-load from DDR memory when reading occurs in non-trivial patterns. In this paper, we propose OpenCL approaches to these two problems, leveraging the template-based approach to generalise them and to make them parametrisable for design-space exploration.

We demonstrate the use of a combination of the proposed approaches in the design of a hardware architecture for a large scientific high-performance computing problem, namely for a pulsar search pipeline.

Pulsars are extremely dense, highly magnetised, and fast-rotating neutron stars that were formed from the remnants of supernovae. Their physical properties, and in consequence, the emission of electromagnetic radiation, are very stable over time, making them useful reference objects for astronomers. Finding new pulsars is a key objective for the Square Kilometre Array (SKA) [1], which is currently being constructed, and will become the world’s largest radio telescope when finished.

As pulsar signals usually arrive very weak on Earth, multiple radio telescopes, distributed over the surface of our planet, have to work together to form a highly sensitive receiver. Still, elaborate signal processing is required for detection. The SKA’s pulsar search subelement contains modules to filter out terrestrial radio interference and cosmic noise, and to compensate for the unknown distance and movement of a pulsar candidate.

The focus of this work is the Fourier domain acceleration search (FDAS) module, which aims to reverse smearing of the received signal due to the acceleration of a pulsar orbiting another object, e.g., a second pulsar.

To the best of our knowledge, Wang et al. [17–19] conducted the only thorough investigation into the problem, and developed and evaluated an FPGA-based accelerator for FDAS. They established that FPGAs offer better energy efficiency than GPU-based accelerators for this application, and showcased the benefits of describing hardware accelerators, to be deployed in a global science project, in OpenCL. Our work builds upon their research.

The acceleration of the FDAS algorithm is, admittedly, a niche problem because it concerns a very specialised scientific community. Nevertheless, it is an important

problem to research, as due to the scale of the SKA telescope, thousands of FDAS accelerators will need to run continuously for months, so any improvements in throughput and energy-efficiency will yield significant practical savings.

More importantly, it is somewhat representative of large scientific high-performance computing problems: *i)* it consists of several high-level stages; *ii)* it is a large scale problem, with strong requirements; *iii)* data order needs to be adjusted on the fly; *iv)* it needs to use large amounts of external (DRAM) memory; *v)* it loads from memory with non-trivial patterns.

The main contributions of this work are

- the study of OpenCL constructs for stream data reordering,
- the proposal of an OpenCL kernel structure for pre-fetched burst loads at different speeds,
- the use of a template engine for parametrisable and maintainable code that can be effectively synthesised,
- the proposal of a new architecture for accelerating FDAS inside the pulsar search subelement, employing the proposed techniques, and
- the experimental evaluation of the new FDAS architecture, which achieves a 10x speedup over a previous design on the same hardware.

The presented FDAS implementation is parameterisable, and thus enables automatic design-space exploration for our current, Arria 10-based target card, as well as for more capable FPGA devices. This is a key requirement, because the SKA project has not yet finalised its decision for a particular accelerator device. The key to our very substantial 10x speedup was: *i)* a holistic approach that optimises connected major pipeline stages at the same time; *ii)* the focus of the new design on data read performance through the proposed techniques and a new systolic array architecture for parts of FDAS; *iii)* finding the right configurations and replications of the various kernels with an extensive design-space exploration.

The remainder of this paper is organised as follows. Section 2 studies and proposes the advanced OpenCL aspects and techniques we are going to employ in our new FDAS architecture design. Section 3 explains the radio-astronomical context of the FDAS algorithm, and the computer-engineering challenges its implementation poses. In Section 4, we summarise key design choices from the state-of-the-art work, analyse its limitations and propose a novel accelerator design. Section 5 presents the new proposed architecture, detailing the major aspects that lead to the increase in performance. We present design-space exploration results in Section 6, and conclude in Section 7.

## 2 Advanced OpenCL aspects of multi-kernel architectures

The case study in the second part of this paper demonstrates that parametrisable multi-kernel architectures are a powerful methodology to guide an high-level synthesis compiler into laying out an algorithm spatially in a way that efficiently utilises the available hardware resources. Before we dive into the details of the concrete signal-processing algorithm, we highlight three aspects of “programming” such an

accelerator in OpenCL, which we expect to be applicable to a broader class of FPGA-based designs.

## 2.1 Kernel replication

The ability to replicate the functionality of a kernel multiple times is a crucial building block for any scalable architecture. In the following, we revisit the three options available in the Intel OpenCL compiler, and propose the use of an external template engine to achieve a more compact and concise coding style.

**Using compiler-specific attributes** The vendor-specific attribute `num_compute_units(N)` instructs the compiler to instantiate the annotated kernel  $N$  times<sup>1</sup> internally, as illustrated in Listing 1.

```

1  __attribute__((num_compute_units(N)))
2  __attribute__((autorun))           // mandatory
3  __attribute__((max_global_work_dim(0))) // mandatory
4  kernel void K(/* no arguments allowed */) {
5      uint id = get_compute_id(0);
6      /* code */
7  }
```

**Listing 1** Replication via compiler-specific attributes

This replication is not visible at the language level, hence it is mandatory to annotate such kernels with the `autorun` attribute, as the individual instances cannot be interacted with from the host code. In consequence, no kernel arguments are allowed, which makes it impossible to access the global memory, and impractical to reuse an instance at different locations within the accelerator pipeline.

Each kernel instance can be customised based on the identifier returned by the `get_compute_id` intrinsic. While the identifier's concrete value will be constant-propagated in each instance, language-wise it is prohibited to be used when a literal is expected, e.g. as a dimension in an array declaration as shown in Listing 2.

```

1  float buffer[get_compute_id(0)+1][1024]; // error!
```

**Listing 2** Forbidden use of intrinsic function in buffer declaration

**Using preprocessor directives** OpenCL C supports the usual preprocessor directives to define macros and conditionally select parts of the source code. These can be combined to facilitate kernel replication.

<sup>1</sup> Similar to the OpenCL workgroup concept, up to three dimensions may be specified. For brevity of the discussion, we only show the one-dimensional case here.

In the example in Listing 3, the functionality intended to be instantiated multiple times is contained in a helper function **k\_func**. The actual kernel function is wrapped in the macro **KERNEL(name)**. The compiler will inline **k\_func** into its body. Now, as the preprocessor does not offer a looping construct, one needs to manually invoke the macro, with different kernel names and wrapped in nested **#ifs**, “enough” times, i.e. up to an expected maximum number of replicas. Optionally, a macro representing the instance number, may be defined and used inside **k\_func**.

```

1 inline void k_func(/* arguments */) { /* code */ }
2
3 #define KERNEL(name) \
4     kernel void name (/* arguments */) { k_func(/* arguments */); }
5
6 #if N > 0
7     #define ID 0
8     KERNEL(K_0)
9     #undef ID
10    #if N > 1
11        #define ID 1
12        KERNEL(K_1)
13        #undef ID
14        ... // repeat until an upper bound for N is reached
15    #endif
16    #endif

```

**Listing 3** Replication with the help of the preprocessor

The resulting instances behave like normal kernels, e.g. they can have arguments and are visible to the host. However, as evident in the example above, the manual effort required makes this method error-prone and verbose.

**Using code transformations** The last built-in method, as depicted in Listing 4, does not actually instantiate multiple kernels.

```

1 inline void k_func(uint id, /* arguments */) { /* code */ }
2
3 kernel void K(/* arguments */) {
4     #pragma unroll
5     for (uint k = 0; k < N; ++k)
6         k_func(k, /* arguments */);
7 }

```

**Listing 4** Replication through code transformations

Using a combination of the compiler’s loop unrolling and inlining transformations, the desired functionality, which in this example is contained in the helper function **k\_func**, is replicated inside a single kernel. This method is only suited for simple, regular and non-stalling computations, as the replicas are all part of the same pipeline in the surrounding kernel. If **k\_func** contains loops, the underlying C semantics may force the compiler to execute their replicas sequentially.

**Using a template engine** For maximum flexibility, we propose to introduce a lightweight template engine to the compile flow. An example is shown in Listing 5. In this paper, we specifically use Mako<sup>2</sup> to augment OpenCL source files with snippets of Python code, but expect other templating languages to work just as well.

With Mako's `% for` loop—, we can replicate the kernel as concisely as with the `num_compute_units`-attribute, but are not subject to that method's limitations.

```

1 % for k in range(N):
2 kernel void K_${k}(/ arguments */) { /* code */
3 % endfor

```

**Listing 5** Proposed replication using Mako

In fact, the template engine can be thought of as a better preprocessor that enables instance customisation with the full expressiveness of Python. Consider the example in Listing 6, which is representative of code used in the preload-kernels discussed in Section 5.2. Here, we again use a Mako loop to instantiate a parameter-dependent number of buffers. The concrete number is a non-linear function of a global parameter and the instance identifier, and is computed in the Python snippets.

```

1 <% import math
2 def num_buffers(u, v):
3     return ...
4 %>
5
6 % for k in range(N):
7 kernel void K(/ arguments */) {
8     <% n_buf = num_buffers(someParam, k+1) %>
9     % for p in range(n_buf):
10 float buffer_${p}[1024];
11 % endfor
12 for (uint i = 0; i < 1024; ++i) {
13     % for p in range(n_buf)
14     buffer_${p}[i] = ...
15     % endfor
16 }
17 ...
18 }
19 % endfor

```

**Listing 6** Proposed instantiation of a parametrisable number of buffers using Mako

Additionally, the template mechanism allows us to use subscripted names to communicate to the compiler that these buffers shall be independently accessible. While the same effect can often be achieved in plain OpenCL with unrolled loops and the automatic (or user-assisted) banking of multi-dimensional arrays, the programmer must be aware of certain pitfalls, e.g. that all dimensions but the first must be powers of two. In situations in which the programmer cannot or does not want to cater

<sup>2</sup><https://www.makotemplates.org>

to the constraints of the automatic banking, the illustrated approach is a valuable alternative.

In our case study, we use the template-based replication to distribute work to a parametrisable number of fast Fourier transformation (FFT) engines, and the instance customisation to compute the exact minimum number of load units and buffers in kernels that feed data into a systolic array structure.

## 2.2 Variable-speed burst loading

Getting data from the global memory into the compute pipeline as fast and efficient as possible is one of the key problems in FPGA-based accelerator designs. With the *prefetching load unit* (PFLU), the Intel OpenCL toolchain already offers a component to efficiently stream chunks of data.

Suppose we want to gather data from different parts of a large global memory buffer. These accesses should be independent in the sense that a stall in one should not thwart progress in the others. To achieve such a decoupling, it is desirable to wrap PFLUs in simple kernels that feed the data to the compute parts of the pipeline via channels. However, instantiating a parametrisable number of such load kernels is impractical because of the limitations of the built-in replication methods discussed in the previous section. In contrast, the template method makes this easy to express, as illustrated in Listing 7.

```

1 channel float4 data_out[${N}];
2 % for k in range(N):
3 kernel void loader_${k}(global float4 * ptr, const uint n_bundles) {
4     for (uint b = 0; b < n_bundles; ++b) {
5         float4 data = ptr[b]; // simple access pattern, results in PFLU
6         write_channel_intel(data_out[${k}], data);
7     }
8 }
9 % endfor

```

**Listing 7** Instantiation of a parametrisable number of prefetching load units wrapped in simple kernels

An additional benefit of this concept is that the consumer can control the pace of the data, e.g. request data every three cycles, without introducing complex control flow in the load kernel. Keeping the kernel structure and the memory access pattern simple is key to guarantee that the compiler will actually infer<sup>3</sup> a PFLU.

In context of our case study, we employ this technique for feeding the input data provided by the host into the accelerator pipeline. Furthermore, in the second phase of the FDAS algorithm, being able to combine data streamed from multiple locations and at different speeds turned out to be the missing piece for achieving a bandwidth-efficient access pattern.

<sup>3</sup>Intel recently introduced the `__prefetching_load` intrinsic to let the programmer manually request this particular type of load unit.



## 2.3 In-stream reordering

FPGAs excel at pipeline processing, but not every algorithm can be mapped to a perfect pipeline. Instead, an intermediate stage for reordering the stream at the granularity of a *tile* of data is often required. *Double buffering* allows a back-to-back processing of tiles without impairing the pipeline's throughput.

We present a generic recipe to implement such a kernel in OpenCL. To make the discussion more illustrative, we show how to perform a simple transpose operation on a stream of *bundles* (here, comprised of four floating-point values) in Fig. 1.

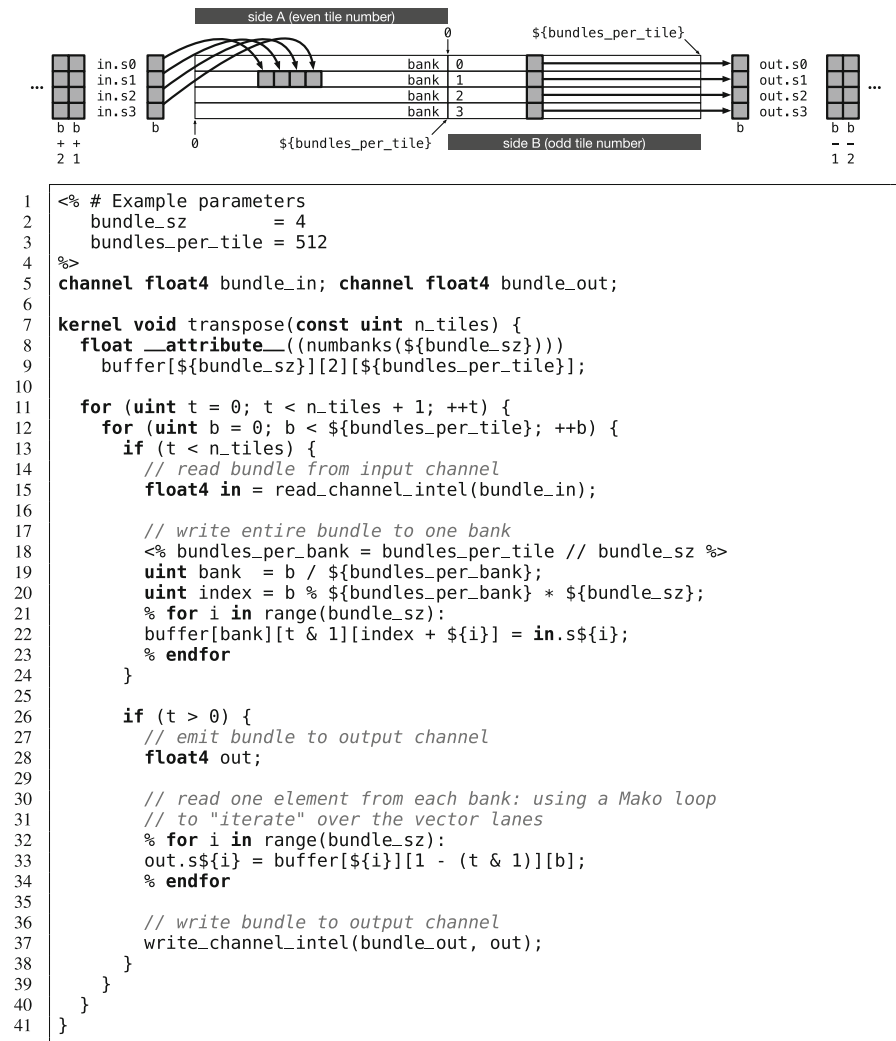


Fig. 1 Transposition as an example of double-buffered, in-stream reordering

In the example, we use the built-in<sup>4</sup> vector type **float4** to represent a bundle of data, and assume that each tile contains 512 such bundles. As the incoming data rate must match the outgoing data rate, we use this type for both the input and the output channel (line 5).

In lines 8–9, the reorder buffer is declared. Together with the **numbanks** attribute, this particular order of dimensions results in a layout and banking as shown at the top of Fig. 1, which allows one bundle to be stored and one bundle to be read in each cycle, stall-free, which is essential to sustain a continuous operation of the reorder kernel.

The kernel processes **n\_tiles** tiles per launch. The main loop (lines 11–40) runs for **n\_tiles + 1** iterations, and implements the desired double buffering via the middle dimension: New tiles are written into the buffer at **[t & 1]** in iterations 0...**n\_tiles - 1** (lines 13–24), whereas transposed tiles are emitted from the buffer at **[1 - (t & 1)]** in iterations 1...**n\_tiles** (lines 26–38). The compiler can infer from this idiomatic coding style that the read and write accesses are mutually-exclusive in all iterations.

Up until this point, the presented structure is generic. The operation-specific parts are in lines 17–23, i.e. the handling of the input bundle, and in line 30–34, i.e. the preparation of the output bundle. These parts can be replaced with arbitrary access patterns, e.g. bit-reversal of the bank number or element index, as long as *no more than one stall-free access is issued per bank*. The example satisfies this constraint: In the input part, the entire bundle is written to one of the banks using a single, wide store operation, whereas in the output part, one element is read from each bank.

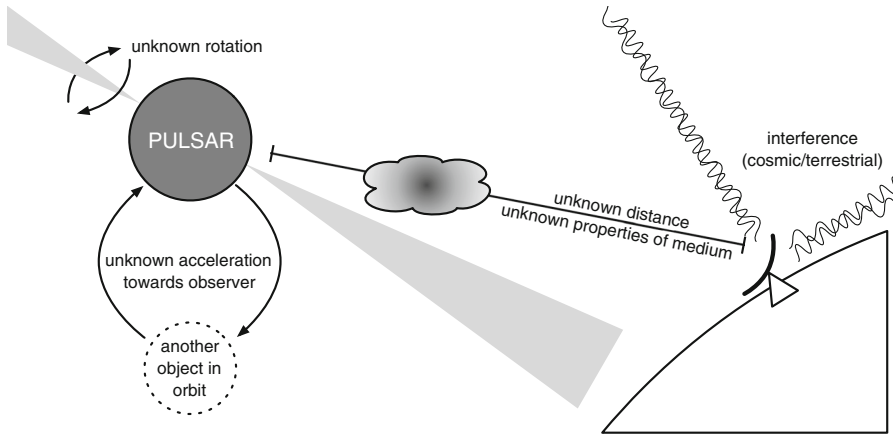
In the case study, we apply this recipe for a frictionless integration of FFT engines into the accelerator pipeline. The particular engines used in this work expect a transposed stream of bundles with bit-reversed elements as input, and emit the output tile's bundles in a bit-reversed order.

### 3 Pulsar Search with the SKA

Pulsars emit electromagnetic radiation at a characteristic spin frequency, coupled to their very slowly decreasing rotation speed. Once a pulsar's location and rotational parameters are known, it can serve as a cosmic reference point for maps and probes into the interstellar medium. However, detecting new pulsars is a challenge due to a number of unknowns, as summarised in Fig. 2. Under ideal conditions, a pulsar would appear as an easily identifiable peak in a frequency spectrum received by a radio telescope. In reality, the signal is overlain by interference from other terrestrial or cosmic sources, distorted from travelling an unknown distance through an unknown interstellar medium (e.g. gas clouds), and smeared across a range of frequencies due to the potential movement of the object towards or away from the observer.

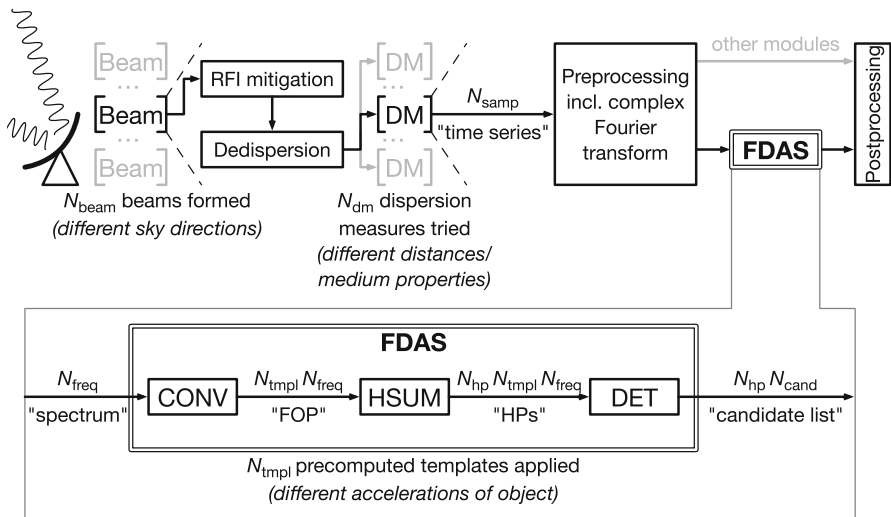
The SKA's PSS pipeline [10], outlined in Fig. 3 as part of the central signal processor, attempts to mitigate these effects. The pipeline operates according to the parameters that we introduce in the following paragraphs, and summarise in Table 1.

<sup>4</sup>User-defined data types would work as well, and could even be subject to a template parameter.



**Fig. 2** Challenges of pulsar search

A single observation spans  $T_{\text{obs}} = 536.86$  seconds and produces  $N_{\text{samp}} = 2^{23}$  samples.  $N_{\text{beam}} \in 1000 \dots 2000$  different sky locations are recorded at the same time, and RF interference mitigation techniques are applied to the resulting beams. The dedispersion module tries  $N_{\text{dm}} = 6000$  dispersion measures (DMs) to reverse the effects of different distances and medium properties. Each dedispersed data chunk then goes through further preprocessing steps (detailed in [10]), including a complex Fourier transformation, which yields a frequency spectrum comprising  $N_{\text{freq}} = 2^{22}$  points. A cleaned version of this spectrum is the input for the FDAS module, which is the focus of this paper and will be introduced in the next paragraph. Due to the high data rate coming from the telescope's receivers, long-term storage of the raw data is



**Fig. 3** Overview of the pulsar search subelement inside SKA's central signal processor

**Table 1** Summary of pulsar search subelement parameters

Parameter	Description	Value
$T_{\text{obs}}$	Observation time	536.86 s
$N_{\text{beam}}$	Number of beams	1000-2000
$N_{\text{dm}}$	Number of dispersion measures	6000
$N_{\text{samp}}$	Number of samples per observation	$2^{23}$
$N_{\text{freq}}$	Number of frequency bins	$2^{22}$
$N_{\text{tmpl}}$	Number of templates	85
$N_{\text{tpas}}$	Number of templates per acceleration sign	$\lfloor N_{\text{tmpl}}/2 \rfloor$
$N_{\text{mtcf}}$	Maximum number of template coefficients	421
$N_{\text{hp}}$	Number of harmonic planes	8
$N_{\text{cand}}$	Number of candidates per harmonic plane	64

source: project-internal documents [8, 11]

infeasible, and thus the pipeline must be able to process data within the observation time. To this end, all preprocessing steps, as well as the FDAS module, should operate with an initiation interval (II) of at most 90 ms ( $= T_{\text{obs}}/N_{\text{dm}}$ ), assuming that we have separate compute resources per beam, and handle all dispersion measure trials in a pipelined fashion.

The SKA's FDAS module [11] is based on work by Ransom et al. [14]. Its objective is to detect binary pulsars, which are pulsars in a close orbit with other massive objects, and thus are of particular interest to astronomers as they allow the study of gravitational waves. In contrast to individual pulsars, the signal received from a binary pulsar will be smeared depending on its movement relative to the observing telescope.

As shown in Fig. 3, FDAS conceptually comprises three phases: convolution (labelled “CONV”), harmonic summing (“harmonic summing”), and detection (“DET”). In the convolution phase, the input spectrum is convolved with a set of  $N_{\text{tmpl}}$  precomputed *templates*<sup>5</sup>, which compensate the smearing for a specific acceleration. In context of the SKA,  $N_{\text{tmpl}} = 85$  such templates, modelling accelerations from  $-350\text{m/s}^2$  to  $350\text{m/s}^2$ , and comprising up to  $N_{\text{coef}} = 421$  coefficients, are used. There are  $N_{\text{tpas}} = \lfloor N_{\text{tmpl}}/2 \rfloor$  templates per acceleration sign, which we number as follows: Templates  $-N_{\text{tpas}} \dots -1$  correct for the negative accelerations, template 0 models zero acceleration, i.e. it passes through the input unmodified, and templates  $1 \dots N_{\text{tpas}}$  compensate the positive accelerations. The result of this phase is called

<sup>5</sup>Please note the name clash between the *acceleration* templates introduced here, and the OpenCL *code* templates referenced elsewhere in this paper.

the filter-output plane (FOP)<sup>6</sup>, a two-dimensional data structure with  $N_{\text{tmpl}}N_{\text{freq}}$  elements.

The harmonic summing phase aims to isolate potential pulsar signals further from the noise floor: If a peak at a certain frequency, in the spectrum corrected for a particular acceleration, actually originates from a pulsar, we expect to detect smaller peaks at the harmonics, i.e. integer multiples, of that frequency and acceleration. To that end,  $N_{\text{hp}} = 8$  harmonic plane  $\text{HP}_1, \dots, \text{HP}_{N_{\text{hp}}}$  are computed from the FOP, according to Equation 1. The recursive definition relies on a stretched view (by an integer factor  $k$ ) of the FOP. Analogously to the FOP, the harmonic planes are indexed by a template number  $t$  in the range  $[-N_{\text{tpas}}, N_{\text{tpas}}]$  and a frequency bin number  $f$  in the range  $[0, N_{\text{freq}}]$ .

$$\text{HP}_k(t, f) = \begin{cases} \text{FOP}(t, f) & k = 1 \\ \text{HP}_{k-1}(t, f) + \text{FOP}\left(\left\lfloor \frac{t}{k} \right\rfloor, \left\lfloor \frac{f}{k} \right\rfloor\right) & k = 2, \dots, N_{\text{hp}} \end{cases} \quad (1)$$

In the detection phase, we compare the signal power, obtained by squaring the magnitude of each complex point in each of the harmonic plane, with a threshold value: If it is greater, then we record this detection as a candidate  $(k, t, f, |\text{HP}_k(t, f)|^2)$ , represented by the point's harmonic, template and frequency bin numbers, as well as the corresponding peak power. Afterwards, a candidate list containing  $N_{\text{cand}}$  entries (here: 64) per HP is passed to postprocessing modules that validate and refine the detections, before passing the results to the science data processor for further analysis.

## 4 Improved baseline architecture

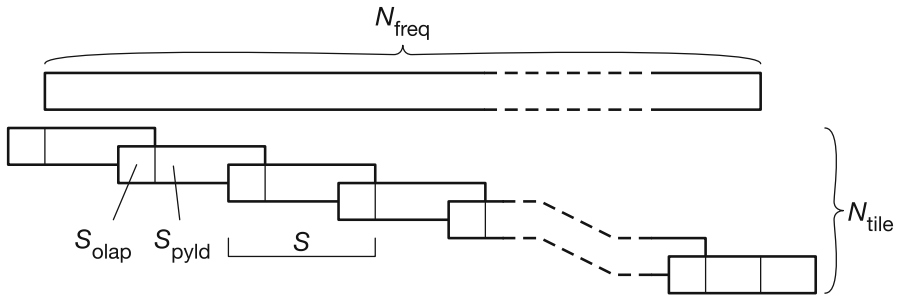
In this section, we establish an improved baseline architecture, incorporating the insights from prior work and current best-practices for OpenCL-based accelerator design.

### 4.1 Algorithmic tweaks

We include the following design decisions and algorithmic tweaks proposed by Wang et al. [17–19] in our baseline architecture.

**FT convolution and choice of FFT parameters** Wang et al. found that implementing the convolution phase with the help of the convolution theorem (“FT convolution”) is more resource-efficient than the direct approach for supporting up to  $N_{\text{coef}}$  coefficients, but a suitable FFT engine for  $N_{\text{freq}}$  points could not be fit on the device. To that end, the authors use the overlap-save (OLS) algorithm [13], which splits the input into smaller tiles of  $S$  elements, and thus reduces the required FFT size to that value. For semantic correctness, the neighbouring tiles must overlap by at least

<sup>6</sup>The use of the term ‘filter’ here refers to an engineer’s viewpoint, as the convolution of an input with a set of coefficients is equivalent to applying an FIR filter to a signal.



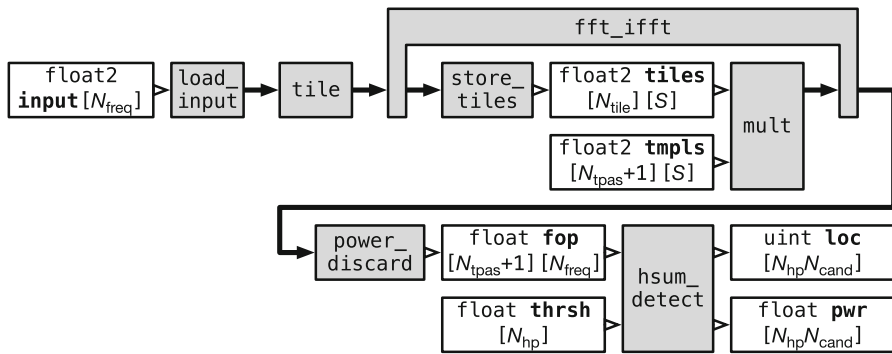
**Fig. 4** Tiling scheme, according to the overlap-save algorithm, to split a single  $N_{\text{freq}}$ -point Fourier transformation into  $N_{\text{tile}}$  smaller ones.  $S$  is the *tile size*,  $S_{\text{olap}}$  is the required *overlap* (or zero-padding) between neighbouring tiles, and  $S_{\text{pyld}}$  is the resulting *payload*, i.e. the usable chunk of input data in each tile

$S_{\text{olap}} = N_{\text{coef}} - 1$  points. The first and the last tile are simply padded with zeros. It follows that each tile carries a payload of  $S_{\text{pyld}} = S - S_{\text{olap}}$  points, and the process yields  $N_{\text{tile}} = \lceil N_{\text{freq}} / S_{\text{pyld}} \rceil$  tiles in total. This scheme is illustrated in Fig. 4. Each tile is Fourier-transformed, then multiplied element-wise with the Fourier transformation of one template's coefficients. The result undergoes an inverse Fourier transformation. The overlapping parts of the tiles are discarded, and the concatenation of the payloads yields the convolution result. Wang et al. investigated various tile sizes  $S$  and FFT implementations handling  $P$  points in parallel per cycle, and recommend  $S = 2048$  and  $P = 4$ .

**Early computation of spectral power** After the convolution phase, the FOP conceptually contains multiple, de-smearred versions of the input spectrum, i.e. it is comprised of complex amplitudes per frequency bin, but the detection phase only evaluates the signal power, a real quantity. Therefore, the power computation is hoisted towards the end of the convolution phase, before writing the FOP to global memory, which halves the amount of data that needs to be transferred and stored.

**On-the-fly harmonic summing and detection** Instead of explicitly computing and storing  $N_{\text{hp}}$  planes as input for the detection phase, precious bandwidth can be saved by interleaving it with the harmonic summing phase: Each coordinate  $(t, f)$  is “visited” only once, during which all  $\text{HP}_k(t, f)$  are computed and compared to the respective thresholds together.

**Parallel trials for positive and negative accelerations** Each DM trial can be split trivially across two accelerator cards: During harmonic summing, the FOP-halves, representing positive respectively negative candidate accelerations, are independent except for template 0. An individual FDAS accelerator therefore needs to handle merely  $N_{\text{tpas}} + 1 = 43$  templates.



**Fig. 5** Baseline architecture. Host and kernels (grey) communicate either via global memory buffers (white) or channels (arrows)

## 4.2 Baseline architecture

Figure 5 shows our baseline architecture, comprised of seven SWI kernels, arranged into a pipeline. The computation is implemented using single-precision floating-point arithmetic.

**Convolution** The pipeline’s first six kernels implement the FT convolution. The main component, the **fft\_ifft** kernel, is used twice, to perform an FFT as well as to compute its inverse. The kernel wraps a 2048-point, 4-parallel radix-2<sup>2</sup> feed-forward FFT engine based on the work of Garrido et al. [2], and adapted from an Altera reference implementation [5]. The engine is fully pipelined and processes four points per cycle. This characteristic defines the design of the FT convolution pipeline: In order to match the data rate of the engine, the surrounding kernels also process four points per cycle, and are connected by 256-bit wide OpenCL channels. While the engine’s raw output is in the FFT-typical bit-reversed order, we employ double-buffering as outlined in Section 2.3 to linearise it before emission to the wrapper kernel’s output channels.

The pipeline starts with the **input** buffer, which is filled by the host. The kernel **load\_input**, which wraps a prefetching load unit to ensure bandwidth-efficient burst-reads from global memory (cf. Section 2.2), linearly reads packs of four complex values from the buffer and emits these to the output channels. Next, the **tile** kernel combines two functions: First, it uses a shift register to generate the partly overlapping tiles, as mandated by the OLS algorithm, from the stream of input points. Secondly, it uses the recipe from Section 2.3 to reorder the tile elements to accommodate the FFT engine’s internal design, which requires that elements (0, 1024, 512, 1536) arrive in step 0, elements (1, 1025, 513, 1537) arrive in step 1, and so forth. The **store\_tiles** kernels linearly stores the packs it receives from **fft\_ifft** to the **tiles** buffer.

During initialisation, the host uploads the template coefficients, which are already Fourier-transformed and arranged to match the order in the **tiles** buffer, to the global memory buffer **tmps**. The **mult** kernel first burst-loads one template to an internal

buffer. We then continuously load packs of tile elements and template coefficients, perform the element-wise multiplication, and feed the result to the **fft.ifft** kernel. Lastly, the **power\_discard** kernel computes the magnitude of the complex points it receives, and discards the overlapping parts of the tiles while writing the convolution result linearly to the **fop** buffer. Note that **mult**, **fft.ifft** in inverse mode and **power\_discard** are launched for each of the  $N_{\text{tpas}} + 1$  templates.

**Harmonic summing and detection** The **hsum\_detect** kernel loads the detection thresholds from the global memory buffer **thrsh**. We iterate over all template numbers  $t$  and frequency bins  $f$ , compute all  $\text{HP}_k(t, f)$  with the unrolled form of Equation 1, and compare them to the threshold values in parallel. Detections are stored stall-free to two intermediate ring buffers, which are copied to the global memory buffers **loc** ( $k$ ,  $t$  and  $f$  packed into an **uint** value) and **pwr** (peak power) at the end of execution.

### 4.3 Comparison to prior work

Using the same FPGA card as in this work (see Section 6.1), Wang et al. [17] report an II of 570 ms for their best-performing architecture (“AOLS-2048+NaiveMultipleHP”), which is algorithmically very similar to our baseline architecture. The main differences between their and our work is that our pipeline exclusively contains SWI kernels, requires no explicit FOP preparation phase, and is compiled using the newest board support package (BSP) and OpenCL toolchain available for the target board. Our baseline architecture operates at 267 MHz and achieves a steady-state II of 772 ms (corresponding to the latency of the **hsum\_detect** kernel), but also processes twice the number of frequency bins ( $N_{\text{freq}} = 2^{22}$ , vs.  $2^{21}$  in [17]), and thus already outperforms the state-of-the-art FDAS accelerator design. The roughly 1.5x increase in throughput is welcome, but still insufficient to fulfil the SKA requirements. On the other hand, the resource utilisation is quite low (20 % logic, 13 % DSP blocks, 25 % RAM blocks). To that end, we investigate further parallelisation and optimisation opportunities in the next section.

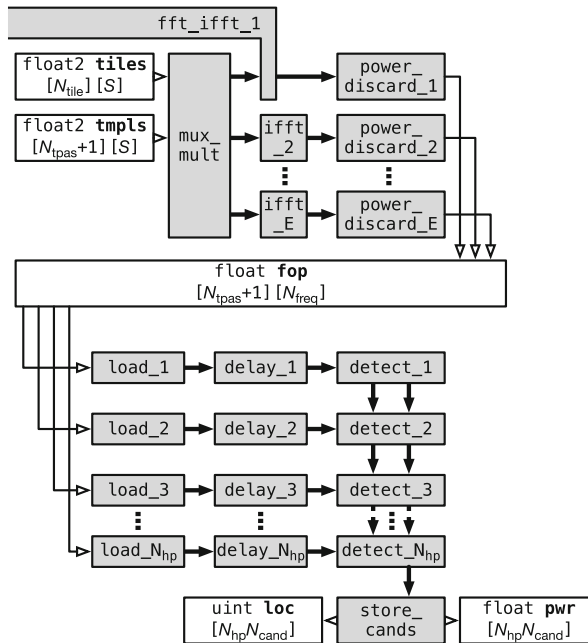
## 5 Proposed architecture

Figure 6 shows our proposed, novel FDAS accelerator architecture. We will discuss its features and underlying design decisions in the following sections.

### 5.1 Parallelisation of FT convolution

The obvious way to reduce the latency of the first phase is to add additional FFT engines to the pipeline, in order to perform the inverse FFT for multiple FT convolutions in parallel. To that end, we instantiate  $E$  kernels wrapping one engine each, using the template-based replication introduced in Section 2.1. This method allows



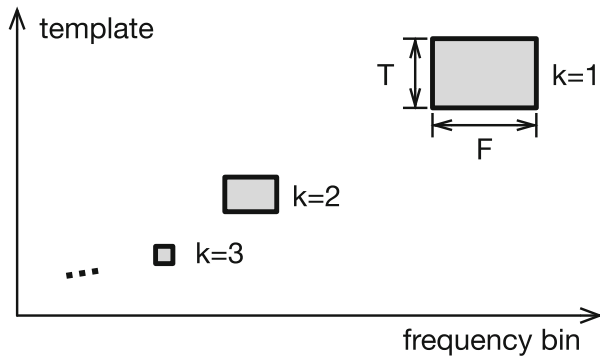


**Fig. 6** Proposed architecture. The input tiling (omitted here) is the same as in the baseline architecture (Fig. 5). Details of the load-delay-detect systolic array are given in Fig. 9

one of these kernels to be switched between normal and inverse operation via a kernel argument, and hence to be used twice in the pipeline. The remaining replicas are specialised to the inverse transformation at synthesis time. The **mux\_mult** replaces the **mult** kernel from the baseline architecture. Internally, it multiplexes each pack of data read from the **tiles** buffer to  $E$  complex multipliers, and forwards the results to the inverse FFT kernels, which in turn are connected to a replica of the **power\_discard** kernel. As a consequence,  $E$  rows of the FOP are written concurrently.

## 5.2 Optimisation of harmonic summing and detection

For the harmonic summing part of the pipeline, we propose a different optimisation approach. Clearly, the amount of data read from global memory, and the non-linear access pattern to it, resulting from implementing Equation 1 naively, is the main challenge for an FPGA's memory controller, whereas the actual computation comprises only simple arithmetic operations, e.g. additions and comparisons, and as such is trivial to map spatially to reconfigurable hardware. Therefore, a simple replication of the baseline architecture's **hsum\_detect** kernel will not suffice to meaningfully improve the performance. Rather, there are two paths to optimise this memory-bound task: *i)* reduce the total amount of data that is read, by increasing the data reuse within



**Fig. 7** FOP elements that need to be loaded from global memory to compute a  $T \times F$ -sized window of the  $HP_1(t, f)$ ,  $HP_2(t, f)$ ,  $HP_3(t, f)$ , ... values together. Increasing harmonic numbers mean more data reuse, and hence fewer elements to load

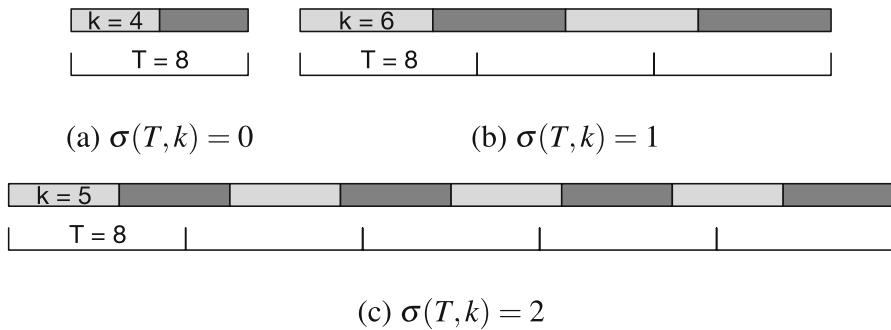
the algorithm, and *ii*) improve the bandwidth efficiency, by presenting a more linear access pattern to the memory system. Fortunately, we are able to address both aspects.

**Reducing total amount of data** Let us revisit Equation 1. In order to compute the  $N_{hp}$  values  $HP_k(t, f)$  for a given coordinate, we need to access the FOP at indices  $(t, f)$ ,  $(\lfloor t/2 \rfloor, \lfloor f/2 \rfloor)$ ,  $(\lfloor t/3 \rfloor, \lfloor f/3 \rfloor)$ , and so forth. This means that neighbouring coordinates can share some of the memory accesses. More generally, consider a *window* covering  $T$  templates and  $F$  frequency bins, as illustrated in Fig. 7. Then, Equation 2 gives a tight upper bound on the number of FOP locations we need to access to perform harmonic summing and detection.

$$\sum_{k=1}^{N_{hp}} \left( \left\lfloor \frac{T}{k} \right\rfloor + \sigma(T, k) \right) \cdot \left( \left\lfloor \frac{F}{k} \right\rfloor + \sigma(F, k) \right) \quad (2)$$

$$\text{with } \sigma(m, n) = \begin{cases} 0 & \text{if } m \bmod n = 0 \\ 1 & \text{if } m \bmod n = \gcd(m, n) \\ 2 & \text{otherwise} \end{cases}$$

The intuition behind the formula is covering a  $T \times F$  rectangle with  $k \times k$  squares ( $1 \leq k \leq N_{hp}$ ). The following discussion refers to the  $T$  dimension only, for brevity, but holds analogously for the  $F$  dimension. In the interior,  $\lfloor T/k \rfloor$  squares always fit. If  $k \leq T$  but  $k$  does not divide  $T$ , additional squares, modelled by the function  $\sigma$ , are required to cover the remaining  $r = T \bmod k$  points. In general, two extra squares, i.e. before *and* after the interior ones, are necessary to complete the cover. Note that the first interior square always has a coordinate that is an integer multiple of  $\gcd(T, k)$ . Therefore, if  $r = \gcd(T, k)$ , only one extra square (before *or* after) will overlay the boundary. Figure 8 illustrates the situation for  $T = 8$  and  $k = 4, 5, 6$ . If  $k > T$ , the bound decays to represent one of two possible situations: In case  $k$  is a multiple of  $T$ , then one access is sufficient per rectangle, otherwise two accesses are required.



**Fig. 8** A graphical interpretation of the  $\sigma$  function defined in Equation 2. The grey rectangles represent  $k$  consecutive points; the brackets below are a moving window of size  $T$ . (a) The “interior”  $\lfloor T/k \rfloor$  points suffice to cover the window. (b) One extra point (left *or* right of the interior) is required to cover the window. (c) Up to two extra points (left *and* right of the interior) are required to cover the window

With the help of the bound, we can predict a significant reduction of the total amount of data read from the global memory. For example, with  $T = 4$  and  $F = 2$ , we need at most 26 loads, compared to  $8 \cdot 4 \cdot 2 = 64$  without sharing data across neighbouring coordinates.

Even so, the windowing approach has major drawbacks. Processing each window still yields many narrow and non-consecutive global memory reads, and we have a varying number of redundant loads at the window borders each time one of its dimensions is not a multiple of the current harmonic. On top of that, we suspect that orchestrating the actual sharing inside the window will result in complex control flow, which might impair the accelerator’s operating frequency.

**Improving bandwidth efficiency** In order to make better use of the available global memory bandwidth, we need to shift our viewpoint. While the memory locations corresponding to a coordinate (or windows thereof) are not consecutive, the accesses required per harmonic plane  $k$  certainly are, assuming we iterate in the direction of increasing frequency bins, and the `fop`[ $N_{\text{tpas}} + 1$ ][ $N_{\text{freq}}$ ] buffer follows the usual layout of C arrays.

We designed the systolic array shown in the lower part of Fig. 6 around this insight. The array has  $N_{\text{hp}}$  rows and three columns. The kernels inside the array, as well as the OpenCL channels connecting them, are parameterised to handle windows of  $T \times F$  coordinates, as introduced in the previous section. Figure 9 gives a concrete example for the  $k = 3$  row in the array, parameterised to  $T = 4$  and  $F = 2$ .

The `load`  $k$  kernels implement the data reuse across different FOP rows, i.e. different templates, which are passed as launch arguments by the host. The kernels contain  $\lfloor T/k \rfloor \cdot \sigma(T, k)$  prefetching load units, each  $F$  floats wide, which are multiplexed to  $T$  output channels. Note that each output channel is fed by at most two load units. If  $k$  divides  $T$ , the mapping from templates to output channels is completely static. Otherwise, the host provides the information required to set up the channel mapping at launch time. In the example, we assume a window starting at a divisible-by-3 template  $\tau$ , hence the green PFLU drives the first three output channels. However, if

$\tau \bmod 3$  were 2, then the blue PFLU would drive all except the first channel. Rows may also be conditionally deactivated if the template number would exceed  $N_{\text{tpas}}$ , or a particular row is not needed to feed the output channels, which periodically happens if  $\sigma(T, k) = 2$ .

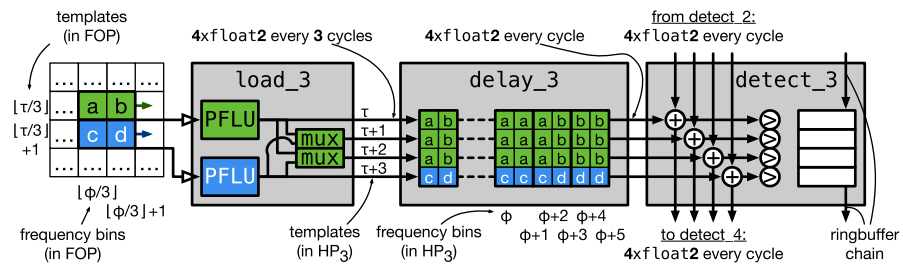
The **delay**. $k$  kernels stretch each incoming data bundle to  $k$  outgoing ones, and thus represent the data reuse across different frequency bins. In the example, the **delay**.3 kernel produces three bundles, containing six consecutive elements  $\phi, \dots, \phi + 5$ , from every input bundle it receives from the **load**. $k$  kernel. The four channels are processed in lockstep. As we continuously stream and delay each FOP row, our approach eliminates the load redundancy across neighbouring windows in the frequency bin direction.

The **load**. $k$  and **delay**. $k$  kernels build upon several of the aspects discussed in Section 2. Their instances are customised based on the architectural parameters as well as on the instance number  $k$ : The load kernels wrap a parameter-dependent number of load units, while the hardware-side of multiplexing logic is determined by a Python function. The delay kernels encompass a template-generated state machine to request new data from the corresponding load kernel at the right time. In combination, these two kernel types demonstrate the variable-speed loading concept of Section 2.2.

The **detect**. $k$  kernels handle a window of  $T \times F$  coordinates per cycle. They receive the partial sum  $\text{HP}_{k-1}(t : t + T - 1, f : f + F - 1)$  from **detect**. $k - 1$ , as well as  $\text{FOP}(\lfloor t/k \rfloor : \lfloor t + T - 1/k \rfloor, \lfloor f/k \rfloor : \lfloor f + F - 1/k \rfloor)$  from **delay**. $k$ . From these inputs, the element-wise sum is computed, representing  $\text{HP}_k(t : t + T - 1, f : f + F - 1)$ , which is then compared against the  $k$ th detection threshold (a kernel argument), and forwarded to the next **detect** kernel. Pulsar candidates are stored stall-free to a ring buffer.

After all FOP coordinates have been visited, we use the second chain of channels (cf. Fig. 6) to transmit the detections serially through the array to the **store\_cands** kernels, which burst-writes the candidate lists to the respective global memory buffers. We choose this design to reduce the number of global memory ports for this non-performance-critical write-back phase.

This multi-kernel architecture lets us concisely express the concurrent streaming of data from disjunct parts of the FOP. Besides serving as high-throughput links, the



**Fig. 9** Third row of the systolic array (see Fig. 6) parameterised for  $T = 4$  and  $F = 2$ . We illustrate processing of three consecutive windows covering templates  $\tau, \dots, \tau + 3$  and frequency bins  $\phi, \dots, \phi + 5$ . We assume  $\tau$  is divisible by 3

OpenCL channels provide an elegant way to synchronise the **load** kernels advancing at different paces. In consequence, the control flow inside the individual kernels remains simple enough to achieve high operating frequencies.

Our implementation is available as an open-source project on GitHub<sup>7</sup>.

## 6 Evaluation

### 6.1 Setup

**Target device** We evaluate the proposed accelerator architecture on a Bittware 385A card, featuring an Arria 10 GX 1150 FPGA and two banks of 4GB DDR3 memory. The memory controller operates at 266 MHz, and provides a maximum bandwidth of  $266 \text{ MHz} \times 64 \text{ byte} \approx 17 \text{ GB/s}$  per bank, if the user logic runs at least at that frequency as well. The host communicates with the device via a PCIe Gen3 x8 link. Recall that we opt to process only one half of the templates per accelerator (cf. Section 4.1) as a concession to the problem size. Therefore, at least two independently operating FPGA cards would be used per beam in the data centre.

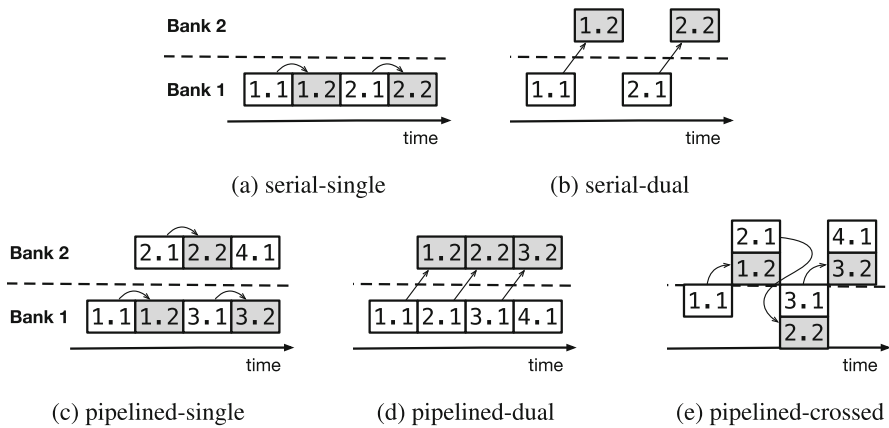
We use the newest available OpenCL BSP from Bittware (“R001.005.0004”) together with the latest supported compiler version (“Intel FPGA SDK for OpenCL 19.1”). This combination then mandates Quartus 17.1.1 for synthesis. We set the HLS target frequency to 300 MHz, to increase the likelihood that the synthesised accelerators match the memory controller’s frequency. Additionally, we disable the automatic interleaving across the DDR3 banks.

**Execution modes and buffer allocations** The FOP buffer naturally decouples the kernels in the proposed architecture into two pipeline stages: *Stage 1* encompasses the transfer of the input spectrum from the host to the device and the OLS-FT convolution. *Stage 2* includes the harmonic summing and detection phase, as well as the transfer of the candidate lists back to the host.

Seeking to design an FDAS accelerator that is capable of sustaining an initiation interval (II) of 90 ms, we investigated both the serial and the pipelined execution of the stages. In the former, the II corresponds to the sum of the stages’ individual latencies, but each stage has undisturbed access to the available memory bandwidth. In the latter, two DM trials overlap in the accelerator in the steady state, necessitating a second set of global memory buffers, which the DDR3 memory can easily accommodate, but reducing the II to the maximum of the stage latencies.

We also conducted experiments with different allocation schemes of buffers to the available memory banks to test out which one suits the memory controller best. Specifically, we associate the **input** and **tiles** buffers with Stage 1, and the **fop**, **loc** and **pwr** buffers with Stage 2. Overall, this results in the five unique combinations of

<sup>7</sup><https://github.com/UOA-PARC-SKA/FDAS>



**Fig. 10** Unique combinations of execution modes and buffer allocations. The boxes represent the execution of a stage, with buffers allocated to a certain bank, and are labeled as  $\langle \text{DM trial} \rangle \cdot \langle \text{stage} \rangle$ . In “single” configurations, both stages’ buffers reside in the same bank for an individual DM trial, whereas in the “dual” and “crossed” configurations, the buffers of stage 1 and stage 2 are allocated in different banks

execution modes and buffer allocations shown in Fig. 10. An initial analysis of the experimental data yielded the following insights:

- For the serial execution, allocating the stages’ buffers to different banks (*serial-dual*, Fig. 10b) always improves performance compared to the same-bank allocation. This is unsurprising, as Stage 1 continuously reads data from the **tile** buffer and writes its results to the **fof** buffer. Hence, allocating the buffers to different banks effectively doubles the available bandwidth.
- For the pipelined execution, the allocation of both stages’ buffers to the same bank (*pipelined-single*, Fig. 10c) dominates the other schemes. We attribute this to the interference of the memory accesses<sup>8</sup> caused by the overlapping execution of the stages, which appears to cancel out the gains we have seen in the serial execution.

In the remainder of this section, we therefore focus on the comparison between the two execution modes, while always using the respective best-performing buffer allocation schemes.

**Design space** In order to stake off the design space for our accelerator, we define lower bounds for the number of cycles required to complete the convolution and harmonic summing phases. Table 2 recaps the relevant architectural parameters that were introduced in Sections 4 and 5.

The decisive factor for the FT convolution (Equation 3) is the duration of passing all tiles through one of the FFT engines, which takes  $N_{\text{tile}} \cdot S/P$  cycles. We always

<sup>8</sup>*pipelined-dual*: Stage 1 has its bank’s full bandwidth available for reading, but writes to the same bank that the previous trial’s stage 2 is reading from. *pipelined-crossed*: Stages 1 and 2 share one bank’s read bandwidth, but Stage 1 can freely write to the other bank.

**Table 2** Summary of parameters of the proposed architecture

Parameter	Description	Value(s)	Remark
$E$	Number of FFT engines	3, 4, 5	
$T$	Number of templates (FOP <i>rows</i> ) handled per pass	1, 2, 3, 4, 6, 8, 12	
$F$	Number of frequency bins (FOP <i>columns</i> ) handled per cycle	1, 2, 4, 8	
$S$	Tile size = FFT size	2048	[17]
$P$	Number of FFT inputs per cycle	4	[17]

need one such pass for the forward transformation, but can parallelise the inverse transformations, leading to a total number of  $1 + \lceil N_{\text{tpas}} + 1/E \rceil$  passes.

$$\text{cycles}_{\text{FTC}}(E) = \left(1 + \left\lceil \frac{N_{\text{tpas}} + 1}{E} \right\rceil\right) \cdot N_{\text{tile}} \cdot \frac{S}{P} \quad (3)$$

The harmonic summing architecture is designed to process windows of  $T \times F$  coordinates in a single cycle. Therefore, in Equation 4, we determine the number of such windows required to cover the entire half-FOP.

$$\text{cycles}_{\text{HSUM}}(T, F) = \left\lceil \frac{N_{\text{tpas}} + 1}{T} \right\rceil \cdot \left\lceil \frac{N_{\text{freq}}}{F} \right\rceil \quad (4)$$

Plugging the current SKA parameters (cf. Table 1) into the bounds, and assuming an operating frequency of 266 MHz, to match the memory controller, it follows that we necessarily need to instantiate at least  $E = 3$  FFT engines, and process  $T \cdot F = 8$  coordinates per cycle, to be able to achieve the desired II.

## 6.2 Results

Table 3 presents the results for our experimental evaluation. Our selection of architectures comprises the four harmonic summing architectures with 8 coordinates per cycle, and three with 12 coordinates per cycle. We exclude non-power of two settings for  $F$ , as these result in inefficient loads, and combine each harmonic summing configuration with either 3, 4 or 5 FFT engines.

First, we list several runtime characteristics for the serial and pipelined execution, measured and averaged from 20 DM trials being processed in the steady state of the accelerator's operation. Under "Latency", the execution time is broken down per stage. Next, the "II" column shows the initiation interval, our main performance indicator. These values are computed from the timestamps of the input data transfers, to be as realistic as possible, and thus may be slightly greater than the sum (serial execution) respectively the maximum (pipelined execution) of the stages' individual latencies, due to kernel launch overheads.

The two "Bandwidth" columns display the mean utilisation of the global memory bandwidth, computed as the amount of data read and written in both stages and divided by the II. Note that these amounts vary between architectures, as smaller

**Table 3** Runtime and synthesis results

Architecture	Serial, dual (Figure 10b)							Pipelined, single (Figure 10c)							Synthesis			
	Latency		II	Bandwidth	En.			Latency		II	Bandwidth	En.			Resources			$f_{\max}$
	S1	S2						S1	S2						ALM	DSP	RAM	
$E \times T \times F$	ms	ms	ms	MB/s	%	J		ms	ms	ms	MB/s	%	J		%	%	%	MHz
$3 \times 1 \times 8$	92	129	221	<b>15740</b>	<b>48</b>	13.6		143	130	145	<b>24780</b>	<b>75</b>	9.1		<b>33</b>	<b>38</b>	<b>35</b>	258
$3 \times 2 \times 4$	89	126	215	13593	40	12.7		135	126	136	21959	64	8.6		34	38	37	274
$3 \times 4 \times 2$	88	<b>109</b>	197	13914	41	<b>11.2</b>		136	110	136	20534	60	8.8		36	38	39	283
$3 \times 8 \times 1$	92	116	209	13032	39	12.0		140	117	141	19675	59	9.2		38	38	44	259
$3 \times 3 \times 4$	95	121	217	13193	43	12.4		139	122	140	20866	69	9.1		41	40	44	237
$3 \times 6 \times 2$	95	113	208	13105	43	11.9		140	113	141	19696	64	9.2		42	40	46	240
$3 \times 12 \times 1$	100	111	212	12734	44	12.4		145	114	146	18650	64	8.9		47	40	54	229
$4 \times 1 \times 8$	82	129	212	15642	46	13.7		142	129	144	23660	69	9.6		36	49	40	293
$4 \times 2 \times 4$	85	126	211	13063	44	12.7		126	126	130	21218	71	8.5		37	49	42	233
$4 \times 4 \times 2$	82	<b>109</b>	192	13468	40	11.4		116	<b>109</b>	117	22654	67	8.0		38	49	44	<b>299</b>
$4 \times 8 \times 1$	85	116	201	12684	39	11.9		120	117	121	21077	64	8.1		41	49	49	256
$4 \times 3 \times 4$	84	121	206	13082	43	12.3		123	122	125	21368	71	8.4		44	51	48	236
$4 \times 6 \times 2$	84	111	195	13077	40	11.5		120	113	121	21710	67	8.3		45	51	51	253
$4 \times 12 \times 1$	87	111	198	12749	44	11.7		124	114	125	20553	72	8.5		49	51	58	224
$5 \times 1 \times 8$	<b>78</b>	129	208	15528	46	13.4		141	129	144	23049	68	9.5		39	60	45	281
$5 \times 2 \times 4$	79	126	205	13041	38	13.0		121	127	131	20443	60	9.0		40	60	46	281
$5 \times 4 \times 2$	82	<b>109</b>	<b>191</b>	13041	42	11.3		<b>112</b>	<b>109</b>	<b>113</b>	22062	72	<b>7.7</b>		41	60	48	240
$5 \times 8 \times 1$	83	117	200	12339	40	11.9		115	117	117	20338	66	8.1		44	60	53	239
$5 \times 3 \times 4$	82	121	204	12798	44	12.5		120	123	125	20658	71	8.6		46	62	53	227
$5 \times 6 \times 2$	82	111	193	12782	41	11.7		116	114	117	20965	67	8.1		48	62	55	244
$5 \times 12 \times 1$	85	111	196	12421	45	11.8		117	115	118	20504	<b>75</b>	8.2		52	62	63	214

1 MB =  $10^6$  B. Bandwidth utilisation is relative to  $2 \cdot 64 \cdot \min(266, f_{\max})$  MB/s. Resource utilisation is relative to 427.2k ALMs, 1518 DSPs and 2713 RAM blocks. Best outcomes per column are marked in bold. Grey background indicates HSUM configurations that process 12 coordinates per cycle

values for the parameters  $E$  and  $T$  lead to more redundant memory accesses. Overall, this metric, which we present here in absolute numbers as well as relative to the operating frequency-dependent available bandwidth, is an indicator how well the memory controller copes with the accelerator's access patterns.

The last runtime metric is the energy consumption, for which we measured the workstation's peak power usage in the steady state with a smart meter, subtracted the systems idle power without the Bittware card installed, and multiplied with the mean II. The remaining columns cover the results of the logic synthesis, extracted from the Quartus report. We present the utilisation of the logic resources ("ALM" in Intel's terminology), "DSP" blocks, and on-chip "RAM" blocks. The last column, " $f_{\max}$ ", lists the operating frequency.

The best-performing architecture overall is  $5 \times 4 \times 2$  with an II of 113 ms in pipelined mode. This represents a 10x speed-up over the best result reported by Wang et al. [17], and still a 6.8x speedup over our improved baseline architecture (cf. Section 4), and is close to satisfy the SKA requirements.

The pipelined execution outperforms the serial execution for all considered architectures. When Stage 1 can run undisturbed, four FFT engines are generally sufficient to complete the stage in less than 90 ms (some configurations with  $E = 3$  and high



$f_{\max}$  also cross this threshold). Unfortunately, the accelerator cannot sustain this performance level when overlapping the processing of two DM trials, as apparent in the reported Stage 1 latencies for the pipelined execution. However, as Stage 2's latency, which ranges between 110–130 ms, seems to be unaffected by the execution mode, the pipelined execution is still faster than the serial execution of the stages.

The results suggest that we have already reached a stagnation point with the hardware at hand. Adding the fifth fast Fourier transformation engine does not improve Stage 1's latency any further. Architectures processing at least four templates in the FOP together yield the best performance for Stage 2. By construction, all architectures read the same amount of points in the frequency bin direction, however, the greater  $T$  is chosen, the more data is reused, and the fewer passes over the FOP are required. Increasing the parallelism from 8 to 12 coordinates per cycle in itself does not improve the stage's latency.

The resource utilisation ranges between  $1/3$  and  $2/3$  and is mostly dependent on the number of FFT engines. A single engine uses  $\approx 10\%$  of the available DSP blocks, and has negligible demands regarding the other FPGA resources. The overhead for a generic instance, i.e. with the capability to perform the transformation in both directions, is small compared to a specialised engine ( $< 1000$  ALMs). We conclude that its double use in the pipeline makes sense from a resource austerity standpoint, but given that the available DSP blocks did not prove to be the limiting factor in our evaluation, we leave it to future work to investigate whether two specialised instances yield a better system performance overall.

The operating frequency is consistently well above 200 MHz, and reaches as high as 299 MHz, demonstrating the toolchain's ability to synthesise high-quality hardware from our template-generated OpenCL code. The bandwidth utilisation fluctuates between 40–50 % for the serial execution, and between 60–75 % for the pipelined execution, an affirming result, considering the memory access pattern underlying the FDAS algorithm. Nevertheless, this data points to the card's memory controller and the small number of independent DDR3 banks as the root causes of the observed performance bottlenecks.

Curiously, the best performance is reached neither by the architecture operating at the highest frequency ( $4 \times 4 \times 2$ ), nor the one having the highest bandwidth utilisation ( $3 \times 1 \times 8$  and  $5 \times 12 \times 1$ ), but at a non-obvious trade-off point, demonstrating the need to enable a systematic exploration even for niche applications such as ours.

Installing the FPGA card and running the FDAS accelerator adds 56–65 W (serial execution) respectively 61–70 W (pipelined execution) to the host's power consumption. We observe that the static power draw blurs the differences among the investigated architectures. In consequence, there is a strong correlation between the mean energy consumed per DM trial and the achieved IIs, attesting the architectures with the shortest IIs the best energy-efficiency.

## 7 Conclusion

We presented a novel accelerator architecture for a demanding computation task in radio astronomy, namely the FDAS module of the Pulsar Search Pipeline of the SKA.

This novel accelerator enables a 10x throughput improvement over the current state-of-the-art design, and at the same time pushes against the limits of our target FPGA card, close to achieving the desired performance envelope.

Our architecture plays to the strengths of FPGAs and is built around the idea of a spatial, pipelined form of computation, which was difficult to express in plain OpenCL without manual code duplication or the danger of obscuring important optimisation potential.

To achieve this faster architecture, we proposed OpenCL constructs for the optimisation of data reordering and data loading from DRAM memory, which can be of interest to other computation problems. Putting the prefetching load units into separate kernels connected by channels helped to address the problem of gathering data from multiple locations and at different paces.

Augmenting the kernel code with snippets of Python code, which is executed by a template engine to produce standard-conforming OpenCL code, proved to be a lightweight and effective solution, which we believe could be helpful in developing OpenCL-based accelerators for other scientific big data applications.

This made our implementation easy to parameterise for design-space exploration. The obvious next step is to repeat the design-space exploration on a platform offering more memory bandwidth. We are especially interested in investigating the distribution of buffers to the many memory channels provided by FPGAs equipped with high-bandwidth memory.

**Acknowledgements** This work benefitted from discussions with the SSquare Kilometre Array Time Domain Team (TDT), a collaboration between Manchester and Oxford Universities, and MPIfR Bonn.

**Author Contributions** *Julian Oppermann*: Conceptualization, Methodology, Validation, Formal analysis, Software, Investigation, Data Curation, Writing - Original Draft, Visualization.

*Mitchell B. Mickaliger*: Conceptualization, Validation, Writing - Review & Editing, Resources.

*Oliver Sinnen*: Conceptualization, Methodology, Validation, Formal analysis, Resources, Writing - Review & Editing, Supervision.

**Funding** Open Access funding enabled and organized by Projekt DEAL. The authors did not receive funding from third parties.

**Availability of data and material** The implementation (see below) contains scripts to prepare suitable test data for the performance measurements.

**Code Availability** The implementation is available as an open-source project on GitHub (<https://github.com/UOA-PARC-SKA/FDAS>).

## Declarations

**Conflicts of interest/Competing interests** The authors have no relevant financial or non-financial interests to disclose. The authors have no conflicts of interest to declare that are relevant to the content of this article.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not

permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

1. Dewdney, P., Hall, P., Schilizzi, R., Lazio, T.: The square kilometre array. *Proc. IEEE* **97**(8), 1482–1496 (2009). <https://doi.org/10.1109/JPROC.2009.2021005>
2. Garrido, M., Grajal, J., Sanchez, M.A., Gustafsson, O.: Pipelined Radix-2<sup>k</sup> Feedforward fast Fourier transformation Architectures. *IEEE Trans. Very Large Scale Integration (VLSI) Syst.* **21**(1), 23–32 (2013). <https://doi.org/10.1109/TVLSI.2011.2178275>
3. Gautier, Q., Althoff, A., Meng, P., Kastner, R.: Spector: an OpenCL FPGA benchmark suite. In: 2016 International conference on field-programmable technology, FPT 2016, Xi'an, China, 7–9 Dec 2016, IEEE, pp. 141–148. <https://doi.org/10.1109/FPT.2016.7929519> (2016)
4. Gorlani, P., Kenter, T., Plessl, C.: OpenCL implementation of cannon's matrix multiplication algorithm on intel stratix 10 FPGAs. In: International conference on field-programmable technology, FPT 2019, Tianjin, China, 9–13 Dec 2019, IEEE, pp. 99–107. <https://doi.org/10.1109/ICFPT47387.2019.00020> (2019)
5. Intel: Fast fourier transformation (1D) design example. <https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/fft-1d.html> (2018)
6. Intel: Intel FPGA SDK for OpenCL pro edition: best practices guide. <https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html> (2021)
7. Jiang, J., Wang, Z., Liu, X., Gómez-Luna, J., Guan, N., Deng, Q., Zhang, W., Mutlu, O.: Boyi: a systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs. In: FPGA '20: the 2020 ACM/SIGDA international symposium on field-programmable gate arrays, Seaside, CA, USA, 23–25 Feb 2020, ACM, pp. 299–309. <https://doi.org/10.1145/3373087.3375313> (2020)
8. Karastergiou, A., Stappers, B., Baffa, C., Williams, C., Roy, J., Levin-Preston, L., Pearson, M., Mickaliger, M., Thiagaraj, P., Lyon, R., Armour, W., Barr, E., Giani, E., Sinnen, O., Dimoudi, S., Adamek, K., Wiesner, K.: SSquare kilometre array central signal processor pulsar search sub-element detailed design document (ED-4a). Internal document SSquare Kilometre Array-TEL-central signal processor-0000082 (2018)
9. Kenter, T., Mahale, G., Alhaddad, S., Grynko, Y., Schmitt, C., Afzal, A., Hannig, F., Förstner, J., Plessl, C.: OpenCL-Based FPGA design to accelerate the nodal discontinuous Galerkin method for unstructured meshes. In: 26th IEEE annual international symposium on field-programmable custom computing machines, FCCM 2018, Boulder, CO, USA, 29 - April 1 May 2018, IEEE computer society, pp. 189–196. <https://doi.org/10.1109/FCCM.2018.00037> (2018)
10. Levin, L., Armour, W., Baffa, C., Barr, E., Cooper, S., Eatough, R., Ensor, A., Giani, E., Karastergiou, A., Karuppusamy, R., Keith, M., Kramer, M., Lyon, R., Mackintosh, M., Mickaliger, M., Van Nieuwpoort, R., Pearson, M., Prabu, T., Roy, J., Sinnen, O., Spitler, L., Spreeuw, H., Stappers, B.W., Van Straten, W., Williams, C., Wang, H., Wiesner, K., The SKA TDT team: Pulsar Searches with the SSquare Kilometre Array. *Proc. Int. Astronomical Union* **13**(S337), 171–174 (2017). <https://doi.org/10.1017/S1743921317009528>
11. Mickaliger, M., Armour, W., Keith, M., Stappers, B.: SKA1 CSP pulsar search sub-element signal processing MATLAB model (ED-7). Internal document SSquare Kilometre Array-TEL-central signal processor-0000085 (2017)
12. Mu, J., Zhang, W., Liang, H., Sinha, S.: Optimizing OpenCL-based CNN design on FPGA with comprehensive design space exploration and collaborative performance modeling. *ACM Trans. Reconfigurable Technol. Syst.* **13**(3), 1–28 (2020). <https://doi.org/10.1145/3397514>
13. Pavel, K., Davi, S.: Algorithms for efficient computation of convolution. In: Design and architectures for digital signal processing, InTech. <https://doi.org/10.5772/51942> (2013)
14. Ransom, S.M., Eikenberry, S.S., Middleditch, J.: Fourier techniques for very long astrophysical time-series analysis. *Astronomical J.* **124**(3), 1788–1809 (2002). <https://doi.org/10.1086/342285>

15. Sanaullah, A., Herbordt, M.C.: Unlocking performance-programmability by penetrating the intel FPGA OpenCL Toolflow. In: 2018 IEEE high performance extreme computing conference, HPEC 2018, Waltham, MA, USA, 25–27 Sept 2018, IEEE, pp. 1–8. <https://doi.org/10.1109/HPEC.2018.8547646> (2018)
16. Sanaullah, A., Patel, R., Herbordt, M.C.: An empirically guided optimization framework for FPGA OpenCL. In: International conference on field-programmable technology, FPT 2018, Naha, Okinawa, Japan, 10–14 Dec 2018, IEEE, pp. 46–53. <https://doi.org/10.1109/FPT.2018.00018> (2018)
17. Wang, H., Thiagaraj, P., Sinnen, O.: Combining multiple optimised FPGA-based pulsar search modules using OpenCL. *J. Astronomical Instrument.* <https://doi.org/10.1142/S2251171719500089> (2019a)
18. Wang, H., Thiagaraj, P., Sinnen, O.: FPGA-based acceleration of FT convolution for pulsar search using OpenCL. *TRETS* **11**(4), 24:1–24:25 (2019b). <https://doi.org/10.1145/3268933>
19. Wang, H., Thiagaraj, P., Sinnen, O.: Harmonic-summing module of SKA on FPGA - optimizing the irregular memory accesses. *IEEE Trans. Very Large Scale Integr. Syst.* **27**(3), 624–636 (2019c). <https://doi.org/10.1109/TVLSI.2018.2882238>
20. Young-Schultz, T., Lilje, L., Brown, S., Betz, V.: Using OpenCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator. In: FPGA '20: the 2020 ACM/SIGDA international symposium on field-programmable gate arrays, Seaside, CA, USA, 23–25 Feb 2020, ACM, pp. 86–96. <https://doi.org/10.1145/3373087.3375300> (2020)
21. Zohouri, H.R., Podobas, A., Matsuoka, S.: Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In: Proceedings of the 2018 ACM/SIGDA international symposium on field-programmable gate arrays, FPGA 2018, Monterey, CA, USA, 25–27 Feb 2018, ACM, pp. 153–162. <https://doi.org/10.1145/3174243.3174248> (2018)

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

## Affiliations

Julian Oppermann<sup>1</sup>  · Mitchell B. Mickaliger<sup>2</sup> · Oliver Sinnen<sup>3</sup>

Mitchell B. Mickaliger  
mitchell.mickaliger@manchester.ac.uk

Oliver Sinnen  
o.sinnen@auckland.ac.nz

<sup>1</sup> Technical University of Darmstadt, Darmstadt, Germany

<sup>2</sup> University of Manchester, Manchester, UK

<sup>3</sup> University of Auckland, Auckland, New Zealand