Abstract—High-end FPGAs are widely adopted as hardware accelerators, due to their power efficiency, flexibility, and high-performance computing ability. They are, therefore, extremely useful devices for a project with challenges and constraints such as the Square Kilometre Array (SKA). However, the traditional design methods require expert hardware knowledge and long development times for each of the SKA’s target applications, making it difficult to make the most out of an array of FPGAs as a shared resource.

High-level development approaches are positioned to overcome this issue. In this paper, we investigate the development efficiency and achievable performance of two popular high-level methods, Maxeler’s MaxCompiler and OpenCL. They are evaluated by implementing a lengthy FIR filter with complex single precision floating-point (SPF) numbers, both in time and frequency domains (TDFIR and FDFIR, respectively). Our results show that high performance can be achieved with low development effort using such high-level methods, where OpenCL outperforms the MaxCompiler for TDFIR.

OpenCL is flexible enough to develop and compare different filter approaches quickly, and as expected FDFIR based implementations clearly outperform TDFIR based ones. To demonstrate OpenCL portability and to compare performance with GPUs, the filters are also evaluated on a GPU platform. The evaluation shows that while the GPU performs better in TDFIR, it is outperformed by the FPGA in FDFIR.

Keywords: FIR Filters, FPGA, OpenCL, Maxeler, SKA

1. Introduction

Field programmable gate array (FPGA) devices have been widely used in digital signal processing applications as acceleration hardware, such as wireless communication, industrial IoT, and massive scale scientific projects [1], [2]. The Square Kilometre Array (SKA) project, as the largest radio telescope array, plans to take an enormous amount of FPGA devices in its Central Signal Processor (CSP) system. The high-end FPGAs have many advantages over CPUs and GPUs in dealing with signal processing applications in SKA, such as power efficiency, flexibility, and high-performance computing capabilities. Although FPGAs are an ideal hardware for high-performance computing, it is necessary for developers to have professional knowledge regarding digital circuit design and FPGA structure. As the design becomes more complex, the development time and cost increased broadly. For traditional FPGA development, when the details of the plan are settled, low-level hardware description languages (HDLs), such as Verilog HDL and VHDL are applied. After system-level and register-transfer level (RTL) verifications, the compiler synthesizes and implements the HDL designs, and generates the bitstream. However, the development and debugging period of a low-level HDL design takes a long time and is prone to errors.

To make it easier for designers, a number of high-level FPGA development approaches emerged, based on languages such as C/C++ and Java. We investigate two high-level FPGA development methods in this paper, OpenCL and Maxeler’s MaxCompiler, and choose the digital finite impulse response (FIR) filters for the FDAS module [3] of SKA with extreme tight timing schedule as the target application. The FIR filter, as a basic building block, is employed in a wide range of digital signal processing applications. As the filter size grows, it becomes increasingly difficult to implement it while maintaining a high throughput and a low latency, due to resource constraints. For the specific case of the SPF FIR filters, this is due to a limited number of digital signal processors (DSPs) in the FPGA devices.

This paper presents high-level synthesis approaches for developing lengthy FIR filters, and is organized as follows. In Section 2, current high-level development techniques of FPGAs are discussed. We introduce time-domain and frequency-domain algorithms for convolution in Section 3. Two high-level development techniques are presented, and three FPGA-based FIR filter structures are proposed in Section 4. In Section 5, two high-level development approaches are compared, the performance of various FIR filters are evaluated, and part of the result are compared with a GPU implementation. The conclusions are drawn in Section 6.
2. Background

To simplify the design flow and reduce the time-to-market and the non-recurring engineering cost, Altera and Xilinx, the two primary FPGA vendors, have released high-level development tools.

The Open Computing Language (OpenCL [4]) standard is published by Khronos Group, and its framework supports heterogeneous platforms developments, such as GPUs and CPUs. Altera makes OpenCL available for developing its FPGAs and provides a development tool called Altera SDK for OpenCL (AOCL) [5]. The AOCL and OpenCL are about the same, while the development time for OpenCL is significantly shorter.

The Vivado HLS is widely adopted for high-level Xilinx FPGA development, targeting C, C++ and System C. AutoESL’s AutoPilot HLS tool coupled with domain-specific system-level implementation platforms is used in [8]. A sphere decoder experiment is done in [8] by applying both HLS and hand-coded HDL. The outcome shows that the HLS solution costs less resource than that of the hand-coded design. Vivado HLS is also applied to computationally demanding control algorithms in [9].

Besides these works, a number of high-level synthesis related research works have been released. For example, LegUp [10] is an open source high-level synthesis tool that accepts a standard C description at the input. In [11], a framework called Nimble that can automatically compile system-level applications specified in C to executables on the target platform. Maxeler has its own high-level FPGA development approaches, which supports both Java and C/C++.

These research works and tools all use assisted high-level language development, including C, C++, System C, Java, and OpenCL, and then transfer the high-level code to low-level HDLs. For our research, we focus on both OpenCL and Maxeler based development on FPGA platforms.

3. Convolution

Our target is a lengthy finite impulse response (FIR) filter with complex single precision floating-point (SPF) coefficients. Assuming the length of target FIR filter is $K$ and the length of input signal is $N$, so the length of output array is $K + N - 1$. We selected a FIR filter for our study as it is a relatively simple problem, yet it is non-trivial to obtain good performance for large filters and input sizes that exceed the resources of an FPGA. Two main FPGA-based approaches are discussed: time-domain FIR filter (TDFIR) and frequency-domain FIR filter (FDFIR) [12].

Algorithm 1 Overlap-Add Algorithm

For a $K$-tap FIR filter $m$, the output $y_m$ is the convolution of filter coefficients $h_m$ with the input signal $x_m$ (Equation (1)).

$$y_m[i] = \sum_{k=0}^{K-1} x_m[i - k]h_m[k], \text{ for } i = 0, 1, \ldots, N - 1, \tag{1}$$

where $x_m[\cdot], h_m[\cdot], \text{ and } y_m[\cdot]$ are all complex numbers. To take advantage of multiple resources of an FPGA, the multiplications in Equation (1) are parallelized when implementing an FIR filter in time-domain. In an high-end FPGA, SPF multipliers are instantiated by using DSP blocks. The amount of DSP blocks decides the number of SPF multiplications that can be executed in parallel. If the FIR filter length $K$ is larger than the parallelization capability of an FPGA, it costs more clock cycles to generate one output and might affect the pipeline structure.

To maintain the pipeline structure and low latency, the coefficient array can be split into a group of sub-arrays to make each sub-array fit into the FPGA. The overlap-add (OLA) algorithm is applied by processing the same input signal a number of times with different coefficient sub-arrays and then sum the overlapped part, as shown in Figure 1. The details are shown in Algorithm 1.

3.1. Time-domain Approach

Based on the convolution theorem in Equation (2), the output can be obtained by the following steps: Fourier transform of input array and coefficient array, element-wise multiplication of these two arrays, and inverse Fourier transform the output array.

$$x * h = \mathcal{F}^{-1}\{\mathcal{F}\{x\} \cdot \mathcal{F}\{h\}\}, \tag{2}$$
where $\mathcal{F}\{\cdot\}$ is the Fourier transform of an array. For the FDFIR, the Fourier transformed input array, coefficient array, and output array should be of the same length. Before Fourier transforming, the input array and coefficient array have to be enlarged to $N + K − 1$ with zero padding, for example, at the end of each array. For short length input signals, such as $2^{10}$ or $2^{12}$, it is possible to direct Fourier transform it by FPGA. However, with the increase of input length $N$, an FPGA cannot execute large FFTs directly and the efficiency of off-chip memory accesses might influence the overall performance. Moreover, additional memory has to be arranged for $N + K − 1$ complex coefficients instead of $K$ complex numbers.

To reduce the FFT length and memory cost, we applied the overlap-save (OLS) algorithm to split the input array into a group of sub-arrays. As it can be seen from the basic process of the OLS algorithm in Figure 2, each input sub-array $I_i$ has overlap section with its neighbors, with length $K − 1$, since each output value depends on $K$ input numbers. For the first input sub-array $O_1$, the first $K − 1$ numbers are padded with zeros, in terms of coefficient array, it is padded with zeros to make its size the same as that of each input sub-array. After the element-wise multiplication in the frequency-domain, we can obtain the output sub-array by discarding the front $K − 1$ overrun elements. When all the output sub-arrays have been generated, we can achieve the final output array by concatenating each output sub-array in order. The details of the OLS algorithm in splitting input signal into $S$ sub-arrays is sketched in Algorithm 2.

4. High-level Approaches and Implementation

In this section, the two selected high-level FPGA description and design approaches, OpenCL and Maxeler’s MaxCompiler, are introduced. The structures of the Naïve and the Overlap-Add TDFIR kernels and the Overlap-Save FDFIR kernel are discussed separately. These structures are applied to process an input array, composed of $N$ complex numbers, by using a $K$-tap FIR filter, where $K$ can be represented as the product of two positive integer $R$ and $K'$, which means $K = R \times K'$.

4.1. High-level Approaches

4.1.1. MaxCompiler. MaxCompiler’s software is developed in Java and C/C++, both higher-level languages. With tailor-made libraries and a sophisticated compiler [13], software development for Maxeler’s devices becomes considerably faster, enabling a higher number of projects. A MaxCompiler project can be split into three parts: the device kernel(s), the device manager configuration, and the host code.

The device kernel, written in Java, is the code that will actually be executed in the FPGA. Given the difference between von Neumann and FPGA’s architectures, programming in Java for a device feels inherently different. The programmer must keep in mind that every kernel instruction will be executed during each clock cycle and, as a result, this resembles a hardware description language, where the notion of time is explicitly included.

The device manager configuration, also written in Java, allows the programmer to define the FPGA’s connections and settings. Here the input and output’ origin/destination, variable type, and length are defined. As for the settings, it is possible to specify parameters like the clock frequency, the total amount of clock cycles, and some compiler settings for each kernel. It is also in the device manager configuration that the programmer sets up the device-related functions to be called in the host code.

Finally, the host code, written in C/C++, contains all the device-function calls, besides the usual pre- and post-processing operations. It is also responsible for fetching the final results from the device’s memory, using functions designed in the device manager.

4.1.2. OpenCL. The OpenCL project contains two parts: kernels for device and programs for the host. Similar to the MaxCompiler development, the device kernel is for the FPGA, however, it is written in OpenCL, which is based on standard ANSI C (C99). The AOC1 conforms to OpenCL specification version 1.0 and some functions of version 1.2 and 2.0. Beside these, it has optimization techniques specific for Altera FPGAs, such as unrolling loops, using channels...
4.2. Time-domain FIR Filter Structure

Let us assume that $4K'$ SPF multiplications can be done in parallel by a target FPGA. Since one complex multiplication requires four SPF multiplications, the maximum length of parallelized taps is $K'$.

4.2.1. Na"ıve TDFIR Kernel. For the Na"ıve TDFIR filter structure, input array and coefficient array are loaded into the off-chip memory of the FPGA board before launching. When the kernel is launched, it loads $K'$ input samples and coefficients to on-chip memory, which is faster than off-chip memory. Since the core computation part can be completely parallelized, the structure can achieve pipelining. The FPGA can execute $K'$ complex multiplications and $2K'$ complex additions and generate one output every clock cycle. To illustrate the ease of the OpenCL approach to implement this filter, and its resemblance to high-level C code, Figure 3 gives the OpenCL code of the NDRange kernel (for brevity, some details are omitted). Similarly, Figure 4 shows the core of the Maxeler’s kernel code for the TDFIR filter.

The Na"ıve TDFIR kernel can be employed to implement filters that exceed the resources of the FPGA, i.e. where $K$ is too large to execute coefficient multiplication in parallel. For that, the overlap-add principle is used, where the host system calls the Na"ıve TDFIR kernel several times using different coefficient sub-arrays each time. The host system then needs to apply the overlap-add principle on the sub-results. Alternatively, a Na"ıve TDFIR kernel can be used by not completely unrolling the convolution multiplications.
But in that case the filter cannot be pipelined properly leading to significantly poorer performance.

4.2.2. Overlap-Add TDFIR Kernel. The alternative to the Naïve TDFIR is the Overlap-Add TDFIR, which is based on the Naïve TDFIR kernel, Figure 5. The kernel will be launched \( R \) times to implement a \( K \)-tap TDFIR. For each coefficient sub-array, the new output array is shifted by \( K \) elements and added to previous output array in the same memory bank. Unlike the Naïve TDFIR kernel, the overlap-add operation of the output array is executed in the FPGA.

4.3. Frequency-domain FIR Filter Structure

The Frequency-domain FIR filter consists on three major parts: FFT, element-wise multiplication, and IFFT. Since it is inefficient to implement a large FFT engine, we studied the Overlap-Save FDFIR kernels. In our work, the applied reconfigurable FFT engine is based on radix-4 feedforward FFT architecture [15] and can be used for both Fourier transform and inverse Fourier transform. The input array for this engine is in regular order and the output array is in a bit-reversed order. The output delay of this FFT engine is \( \frac{N}{2K} - 1 \) clock cycles, where \( N_F \) is the length of Fourier transform and \( N_P \) is the number of points that the FFT engine executes per clock cycle.

To take full advantage of the FPGA resources, two reconfigurable FFT engines are instantiated and configured as FFT and IFFT, respectively. In this case, two engines can work together instead of waiting for the Fourier transformed output array to be generated completely. The coefficient array is pre-processed by a Fourier transform. The structure of the Overlap-Save FDFIR is depicted in Figure 6. As it can be seen, the kernel only needs to be launched once. The Fourier transformed output from the FFT engine is stored in on-chip memory and input array and output array are kept in different off-chip memory banks.

4.4. Analysis

In this section, we discuss the theoretical time and memory costs of all FIR filter kernels introduced before. For the execution cost, we consider the amount of clock cycles required to the actual processing of \( N \) complex numbers.

Additionally there is the kernel initiation and buffer handling times. For the memory cost, the input, coefficient, and output arrays are stored in off-chip DDR memory (global memory) before launching kernels. In the discussion below it is assumed that the input length \( N \) is orders of magnitudes larger than the Fourier transform length \( N_F \) and FIR filter length \( K \), which means \( N \gg N_F \) and \( N \gg K \).

4.4.1. Time-domain FIR Kernels. To process \( N \) complex SPF numbers using a \( K' \)-tap FIR filter kernel, at least \( N \) clock cycles are needed. Using the \( K' \)-tap Naïve FIR filter kernel to implement a \( K \)-tap FIR filter, the kernel has to be launched \( R \) times and each of output arrays has to be sent back to the host. The total number of clock cycles is \( RN \), where \( R = \frac{K}{K'} \). But the overlap-add operations have to be done by the host and the latency depends on the performance of host hardware.

The Overlap-Add TDFIR kernel is based on the Naïve TDFIR kernel. The kernel has to be launched \( R \) times as well, so the clock cycle cost of it is the same as that of Naïve TDFIR kernel. However, the buffer handling time of the Overlap-Add TDFIR kernel is much shorter than that of the Naïve TDFIR kernel. Its output array needs to be loaded from off-chip memory only once, while the output array of Naïve TDFIR kernel has to be loaded \( R \) times.

As for the memory cost, because each output array of Naïve TDFIR kernel has to be sent back to host, one input array and one output array are needed to be stored in off-chip memory. The memory cost of off-chip memory of Naïve TDFIR kernel is \( 2N \cdot C \) bytes, where \( C \) is a constant dependent on the required algorithm precision. As complex SPF number (32+32-bit) for example, it contains 64 bits, which equals to eight bytes \((C = 8)\). For the Overlap-Add TDFIR kernel, the output of one sub-FIR filter has to be added to previous (shifted) output sub-array. So only one buffer set is required to store the output array. The memory cost of off-chip memory is same to that of Naïve TDFIR kernel.

4.4.2. Frequency-domain FIR Kernels. To take full advantage of memory throughput while maintaining a high kernel frequency, the FFT engine is set to execute eight SPF complex numbers per clock cycle, which means \( N_P = 8 \), resulting in an output delay of \( \frac{N}{2K} - 1 \) clock cycles. Assuming there are \( P \) segments, then the number of clock cycles required to compute the Overlap-Save FDFIR is \( \frac{N}{2}N_F(P+2) - 2 \), where \( N_F \) is the Fourier transform length.

In terms of memory cost, the lengths of the input and output of the Overlap-Save FDFIR structure are both...
TABLE 1. MEMORY AND TIME COST OF TDFIR AND FDFIR KERNELS

<table>
<thead>
<tr>
<th>Domain</th>
<th>Kernel</th>
<th>Time cost (cycles)</th>
<th>Memory cost (complex num.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD</td>
<td>Naïve</td>
<td>$2N_{F}$/$N_{F}$</td>
<td>$2N_{F}$/$2N_{F}$</td>
</tr>
<tr>
<td></td>
<td>OLS</td>
<td>$2N_{F}$/$N_{F}$</td>
<td>$2N_{F}$/$2N_{F}$</td>
</tr>
<tr>
<td>FD</td>
<td>OLS</td>
<td>$2N_{F}$/$N_{F}$</td>
<td>$2N_{F}$/$2N_{F}$</td>
</tr>
</tbody>
</table>

$N_{F}$ $\left[\frac{N}{N_{F}+K}\right]$. Only input and output are needed to be stored in off-chip memory, which makes its off-chip memory to be $2N_{F}$ $\left[\frac{N}{N_{F}+K}\right]$. $C$ bytes.

To summarize, the time and memory costs of all considered FIR kernels are given in Table 1. In real execution, the kernel frequency will affect the execution latency as well.

5. Experimental Evaluation

To evaluate the different approaches, we used the requirements of the FDAS module of Pulsar Search in the CSP package of the SKA1-MID radio telescope [3]. It contains a large group of FIR filters, with the largest having 422 taps, and, for each FIR filter, the average time limitation of processing $2^{22}$ complex SPF numbers is $1.04 ms$. We evaluate the FPGA execution latency of processing $2^{22}$ complex SPF numbers by using a 422-tap FIR filter and a smaller 64-tap filter (largest fitting completely on target FPGAs), implemented on the Maxeler and OpenCL based platforms.

To study the portability of OpenCL across devices (one of its promises) and to rank the performance of the FPGA boards, we ported the OpenCL kernels to a GPU platform and measured the performance.

5.1. Platform Details

Two different FPGA boards are the target platforms: the Terasic DE5-Net and the Maxeler Galava, which specifications are shown in Table 2. It can be seen that both devices feature an Altera Stratix V GX FPGA, with the same amount of DSP blocks, which makes it reliable to compare Maxeler and OpenCL kernels. OpenCL is used for the DE5-Net board and the compiler is AOCL, version 15.0.0.145. Maxeler is employed on the Galava board and the compiler is MaxCompiler 2015.1.1.

TABLE 2. FPGA DEVICES’ SPECIFICATIONS

<table>
<thead>
<tr>
<th>Devices</th>
<th>Terasic DE5-Net</th>
<th>Maxeler Galava</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>Stratix V GX5A7</td>
<td>Stratix V GX5A5</td>
</tr>
<tr>
<td>Logic Elements</td>
<td>288m</td>
<td>288m</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>622,000</td>
<td>490,000</td>
</tr>
<tr>
<td>M20K memory blocks</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Memory interface width</td>
<td>128-bit (2 banks)</td>
<td>192-bit (3 banks)</td>
</tr>
<tr>
<td>DDR3 DRAM</td>
<td>4 GB (119 GB/s)</td>
<td>12 GB (9.6 GB/s)</td>
</tr>
<tr>
<td>Connection to the host</td>
<td>PCIe (16 GB/s)</td>
<td>PCIe (2 GB/s)</td>
</tr>
</tbody>
</table>

The GPU platform is an NVIDIA GeForce GTX 850M GPU. This is a laptop GPU, hence will be comparatively slow, but it is based on the same 28nm technology, has the same memory interface width as the FPGA boards, and with 45Watt TDP in the same power class as the boards. However, the off-chip memory is 2GB GDDR5 and the basic graphics clock is 936 MHz.

5.2. MaxCompiler versus OpenCL

The TDFIR structures in Section 4.2, were implemented with the OpenCL and Maxeler approaches for a 64-tap and a 422-tap FIR filter. The kernels for both approaches can achieve pipeline operation with the results presented in Table 3. For OpenCL, the FIR filters are implemented using both single work-item kernel and NDRange kernel, and the details regarding OpenCL in Table 3 are of NDRange kernels. It can be found that the OpenCL kernels consume less logic resource and RAM blocks and achieved a maximum frequency of $f_{max}$, which is higher than the one for Maxeler kernels.

Observe that the different implementations (except for Maxeler 422-tap) exhaust all available DSPs (256) to perform the complex multiplications in parallel. Hence the high-level approaches achieve the same parallelism one could expect from a low-level HDL implementations for complex SPF numbers, verifying the high-level approach.

For the Maxeler design, the kernel frequency has to be set manually, while the OpenCL compiler sets the kernel frequency automatically. It is flexible for Maxeler approach to control the kernel frequency, however, the maximum frequency might be lower than OpenCL kernel frequency, and this will affect the Maxeler kernel performance.

5.3. Time-domain versus Frequency-domain

Since high-performance is achieved for OpenCL kernels, we evaluate variants of the OpenCL TDFIR and FDFIR kernels now, which is supported through the flexible OpenCL high-level approach. The resource usage and performance of processing $2^{22}$ complex numbers are presented in Table 4 for five different approaches. Kernel Naïve/OLA 64_S and Naïve/OLA 64_N refer to a 64-tap Naïve/OLA TDFIR filter using the single work-item kernel and NDRange kernel, respectively. Kernel OLS_1024 denotes the Overlap-Save FDFIR kernel, with an FFT length of 1024. The relative root mean square error (rRMSE) is calculated by comparing with the Matlab results, which are obtained using double precision floating-point operations.

NDRange kernels have higher frequency and performance, meanwhile, cost slightly more logic and RAM blocks compared with single work-item kernels. The performance in GFLOPS and frequency of Naïve _64_N is the

with 45Watt TDP in the same power class as the boards. However, the off-chip memory is 2GB GDDR5 and the basic graphics clock is 936 MHz.
best among these kernels. However, the overall operations workload of TDFIR is significantly higher than that of FDFIR. The more relevant latency of all these kernels in processing $2^{22}$ complex numbers are given in Figure 7. The given times are for the total kernel execution only and do not include the buffer transfers. This is meaningful for the targeted SKA application, as pre- and post-processing of the data will also happen on FPGAs. The performance of kernel OLS_{1024} is the best and the performance of kernel OLA_{64_N} is the worst among these kernels. By using a 422-tap FIR filter, the execution latency of the OLS_{1024} kernel is 7.07 ms, which is up to 39 times faster than OLA_{64_N} kernel.

5.4. FPGA versus GPU

Porting the OpenCL kernels to use on a GPU can be relatively straight forward, under the condition that the original (FPGA) kernel is amicable to it. For example, the NDRange kernel was easy to port, however the single work-item formulation, which is recommended for FPGAs, would be more complex. To port the FDFIR kernel, the essential task was to replace the FPGA specific FFT to use a generic OpenCL FFT (clFFT). The latencies of the ported kernels are given in Figure 8, where kernel GPU_TD is based on the NDRange OpenCL kernels, and GPU_FD is based on Overlap-Save FDFIR (FFT length 2048, which was best). On the GPU, the TDFIR implementations are significantly faster than the FPGA counterparts, which is not surprising given the floating point strength of GPUs. However, and that is surprising, the FDFIR implementation on GPUs is slower than the TDFIR GPU implementation and the FDFIR FPGA implementation. It seems that FFTs are a strength of FPGAs.

6. Conclusion

This paper investigated the development efficiency and achievable performance of two popular high-level methods, MaxCompiler and OpenCL. To evaluate them a case study of implementing lengthy FIR filters was employed. Evidence for their development efficiency is given by (a) the ease with which we implemented numerous variants of the filter (TDFIR, FDFIR, single work-time, NDRange etc.) and (b) the ease with which we ported the OpenCL kernels to a completely different hardware architecture (GPU). When it comes to performance, both the Maxeler and the OpenCL approach use the critical resource of the FPGA (DSPs) almost perfectly, not different to a lower level HDL approach. The performance of OpenCL was superior to that of Maxeler in all cases. Also, the work flow of OpenCL is simpler as the best clock frequency is automatically determined. Using the high-level OpenCL approach, the best FPGA implementation (FDFIR) clearly outperformed a GPU with

<table>
<thead>
<tr>
<th>Kernels</th>
<th>Naïve_{64_S}</th>
<th>Naïve_{64_N}</th>
<th>OLA_{64_S}</th>
<th>OLA_{64_N}</th>
<th>OLS_{1024}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic utilization</td>
<td>49%</td>
<td>51%</td>
<td>50%</td>
<td>51%</td>
<td>83%</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>88%</td>
</tr>
<tr>
<td>RAM blocks</td>
<td>15%</td>
<td>18%</td>
<td>16%</td>
<td>20%</td>
<td>77%</td>
</tr>
<tr>
<td>$f_{\text{max}}$ (MHz)</td>
<td>254.77</td>
<td>270.05</td>
<td>236.01</td>
<td>255.29</td>
<td>168.26</td>
</tr>
<tr>
<td>Performance (GFLOPS)</td>
<td>118.93</td>
<td>137.33</td>
<td>54.60</td>
<td>129.74</td>
<td>133.68</td>
</tr>
<tr>
<td>rRMSE ($\times 10^{-7}$)</td>
<td>0.695</td>
<td>0.699</td>
<td>1.94</td>
<td>1.77</td>
<td>3.26</td>
</tr>
</tbody>
</table>
a comparable power-envelop. We take these results as a verification that a high-level approach can be development efficient and performance competitive even for simple and well understood applications like an FIR filter. This recommends them for a project like the SKA, where many researchers and engineers need to develop long processing flows jointly on hardware platforms that have not been fully specified and can involve FPGAs and GPUs.

Acknowledgments

The authors acknowledge discussions with the TDT, a collaboration between Manchester and Oxford Universities, and MPIfR Bonn and the work benefitted from their collaboration.

References