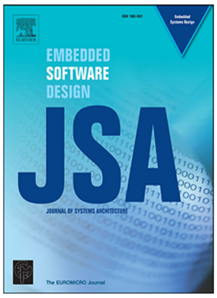
Journal Pre-proof



A cross-platform OpenVX library for FPGA accelerators

Maria Angélica Dávila-Guzmán, Lester Kalms, Rubén Gran Tejero,
María Villarroya-Gaudó, Darío Suárez Gracia, Diana Göhringer

PII: S1383-7621(21)00257-5
DOI: https://doi.org/10.1016/j.sysarc.2021.102372
Reference: SYSARC 102372

To appear in: *Journal of Systems Architecture*

Received date : 24 May 2021
Revised date : 26 October 2021
Accepted date : 30 November 2021

Please cite this article as: M.A. Dávila-Guzmán, L. Kalms, R. Gran Tejero et al., A cross-platform OpenVX library for FPGA accelerators, *Journal of Systems Architecture* (2021), doi: https://doi.org/10.1016/j.sysarc.2021.102372.

A Cross-Platform OpenVX Library for FPGA Accelerators

Maria Angélica Dávila-Guzmán[a,*,1], Lester Kalms[b], Rubén Gran Tejero[a], María Villarroya-Gaudó[a], Darío Suárez Gracia[a] and Diana Göhringer[b]

[a] *DIIS-I3A, Universidad de Zaragoza — HiPEAC Network of Excellence*
[b] *Technische Universität Dresden, Germany*

ARTICLE INFO

Keywords:
FPGA
HLS
OpenVX
Image processing

ABSTRACT

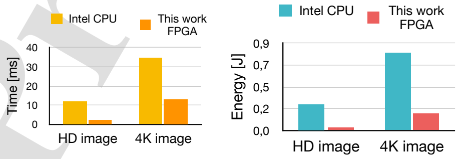
FPGAs are an excellent platform to implement computer vision applications, since these applications tend to offer a high level of parallelism with many data-independent operations. However, the freedom in the solution design space of FPGAs represents a problem because each solution must be individually designed, verified, and tuned. The emergence of High Level Synthesis (HLS) helps solving this problem and has allowed the implementation of open programming standards as OpenVX for computer vision applications on FPGAs, such as the HiFlipVX library developed exclusively for Xilinx devices. Although with the HiFlipVX library, designers can develop solutions efficiently on Xilinx, they do not have an approach to port and run their code on FPGAs from other manufacturers.

This work extends the HiFlipVX capabilities in two significant ways: supporting Intel FPGA devices and enabling execution on discrete FPGA accelerators. To provide both without affecting user-facing code, the new carried out implementation combines two HLS programming models: C++, using Intel's system of tasks, and OpenCL, which provides the CPU interoperability. Comparing with pure OpenCL implementations, this work reduces kernel dispatch resources, saving up to 24% of ALUT resources for each kernel in a graph, and improves performance 2.6 × and energy consumption 1.6 × on average for a set of representative applications, compared with state-of-the-art frameworks.

1. Introduction

FPGAs can increase performance and reduce energy consumption of computer vision (CV) applications. Their pipeline parallelism and massive computing resources are a perfect match to simultaneously process image pixels. For example, Figure 1 shows the execution time and energy efficiency of an OpenVX Canny Edge detector implementation running on a CPU and on an FPGA with the library presented in this work. Compared with the CPU, the FPGA achieves a 5 × speed-up of and improves energy by 9 × (See Section 5 for details). However, the fine grain FPGAs reconfigurability and the complexity of their workflow hinders the ability to easily reach these results without an adequate software programming language support.

Fortunately, programming frameworks, as OpenVX, are designed with the aim of implementing portable CV applications by hiding the hardware complexity behind a simple API. OpenVX presents an open, royalty-free standard for cross-platform acceleration [5] where applications are expressed as graphs to maximize optimization potential because all dependencies are known before the graph is processed. On FPGAs, the acceleration of OpenVX applications remains a challenge because their efficient implementation requires per-device specific optimizations on primitives and communication. Some High-Level Synthesis (HLS) libraries address these requirements; e.g., HiFlipVX, an optimized library of OpenVX functions that exploits streaming capabilities and parametrization for Xilinx FPGAs [15].



(a) Execution Time (lower is better)

(b) Energy efficiency (higher is better)

Figure 1: OpenVX comparison with an Intel Xeon Bronce 3204 CPU and a Intel FPGA Stratix 10 GX for a Canny edge detector with two different image sizes: HD(1920 x 1080 pixels) and 4K(3840 x 2160 pixels). a) Execution time. b) Energy on Intel CPU implementation and the FPGA of this work.

However, HiFlipVX highly-tuned implementation is neither portable nor efficient on other FPGA platforms such as Intel. Other OpenVX acceleration proposal, such as AFFIX, suffers from the same issues, single-vendor support, but in this case for Intel devices [24].

Implementing a portable OpenVX API for FPGA requires to maintain a user-facing API as close as possible to the OpenVX standard. The present paper builds on our previous work [3] with a HiFlipVX implementation to support Intel FPGA devices with different external memories as DDR4 and HBM. This work extension details the key changes required to maintain the library API unchanged, to guarantee portability, and to keep performance.

The proposed implementation leverages Intel's HLS System of Tasks [10] asynchronous model for concurrent execution of OpenVX nodes. Therefore, with this proposal extension, the HiFlipVX graphs can be encapsulated as an OpenCL or SYCL library. The inter-operation of HiFlipVX graphs with OpenCL ensures the host communication with

✉ angelicadg@unizar.es (M.A. Dávila-Guzmán); kalms@tu-dresden.de (L. Kalms); rgran@unizar.es (R. Gran Tejero); mvg@unizar.es (M. Villarroya-Gaudó); dario@unizar.es (D. Suárez Gracia); diana.goehringer@tu-dresden.de (D. Göhringer)
ORCID(S): 0000-0001-7511-2910 (M.A. Dávila-Guzmán)

discrete devices and preserves the asynchronous properties of OpenCL using a unique command queue per OpenVX graph resulting in less runtime overhead.

In summary, the main contributions of this study are:

- A new portable implementation of OpenVX for FPGAs using HiFlipVX, originally designed for Xilinx devices, providing compatibility with Intel devices. [1]
- Interoperation of HiFlipVX applications as OpenCL/ SyCL libraries to support discrete FPGA devices with either DRAM or HBM memories.
- An analysis of the performance and energy efficiency of different graph applications. Compared with previous OpenVX for Intel FPGA implementations, this work improves the performance 2.6 × and saves 1.6 × of energy on average.

The rest of the paper is organized as follows. Section 2 presents related work in this area. Section 3 presents the HLS flow alternatives to implement OpenVX on FPGAs. Section 4 introduces the HiFlipVX library. Section 5 presents the methodology. Section 6 describes the changes to port HiFlipVX to Intel FPGAs. Section 7 discusses the results, and Section 8 sets out our conclusions.

2. Related Work

Computer vision and image processing algorithms require high performance and energy efficiency that can be achieved with FPGAs [6]. Unluckily, the big effort required for programming FPGA is a huge drawback that makes its adoption difficult.

Image processing on a FPGA can be implemented with Domain Specific Languages (DSL). For example, the newly HeteroHalide [16] extends the Halide DSL, formerly used on CPU and GPU, to support Intel and Xilinx devices. Hipacc [22] developed another DSL to support multiple back-ends from different vendors and devices such as FPGA, GPU, and CPU. Also, a Hipacc extension provides support for the OpenVX API [19], but they do not include results for whole application graphs, as this work does. PoliMage [2] and Pu's [21] are two proposals that support Xilinx FPGAs. Despite DSLs are facilitating FPGAs adoption, the steep learning process and the difficulties to enlarge their functionalities remain a challenge.

A more suitable option to ease FPGA implementation is the adoption of a library approach or standard based library. For example, implementing functions from the OpenCV library, Xilinx provides the xfOpenCV library [28]. Other libraries target specific FPGAs vendors, e.g., HiFlipVX and AFFIX are OpenVX libraries for Xilinx and Intel, respectively [15, 24, 25].

Standard libraries together with the adoption of vision standards such as OpenVX could ensure adequate cross-platform portability and performance. Moreover, applications require intensive tuning for each FPGA vendor, even with HDLs. In the case of computer vision applications, as other ones, each type of FPGA requires specific coding style to achieve optimal performance [6, 26]. Among the available options, HiFlipVX and AFFIX are the ones that could offer a more general computer vision library.

Nevertheless, AFFIX is based on OpenCL, which limits the OpenVX functions and graphs implementation. On the other hand, HiFlipVX, through the use of standard C++ language simplifies the graph's implementation, and it focused on portability including explicit data type management to generate optimized hardware. Moreover, HiFlipVX was validated out in numerous embedded applications for Xilinx [23, 20, 1]. This previous analysis recommends to extend HiFlipVX as an standard OpenVX based library to be compatible also with Intel FPGAs.

3. OpenVX Programming Flow Alternatives on FPGA

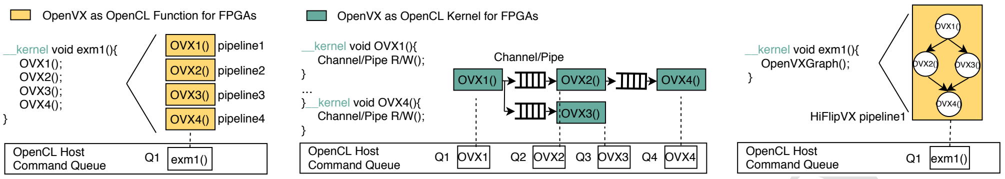
HLS programming languages enable to directly write hardware applications using high-level languages such as C/C++, OpenCL, and SyCL instead of using hardware description languages, reducing the programming entry barrier of FPGAs. HLS have favored the flourishing of a new ecosystem of high level toolkits and programming strategies to reach optimized FPGA pipeline implementations, similarly to what CUDA and OpenCL did to GPUs a decade ago.

One of the most successful approaches in heterogeneous systems is OpenCL, because it unifies the programming language across devices such as CPU and GPU. As an HLS language for FPGA, OpenCL still suffers from a limitation: optimization strategies differ from those from other devices and require choosing the appropriate OpenCL execution model [12, 30]. Furthermore, code written with only the OpenCL standard does not perform well on FPGAs as it requires manufacturer defined extensions.

Programming the OpenVX standard using OpenCL for FPGAs is a challenge since OpenVX applications use a graph-based programming model where nodes, instances of kernels, contain the function code; and edges represent the data movements [5]. This data flow programming model has two main design alternatives in OpenCL:

- *Standard OpenCL*: each OpenVX node is an OpenCL kernel, as shown in Figure 2a. This alternative is portable between manufacturers; but the main disadvantage is the lack of guarantees to generate a deep pipeline connecting the function nodes, because each kernel requires control and communication with the host. Outside of the standard, Xilinx defined their own pragmas and streaming interfaces to generate deep pipelines.
- *OpenCL channels*: each node is an OpenCL kernel, and channels/pipes connect them all. This option allows deep pipelines by the use of streaming communication among kernels, as shown in Figure 2b. In this case, multiple command queues are required to launch every kernel from host to get a concurrent execution of

[1] https://github.com/angelicadavila/HiFlipVX-for-Intel-FPGAs



(a) Standard OpenCL **(b)** OpenCL Channels **(c)** HiFlipVX proposed flow

Figure 2: Programming flow alternatives for OpenVX using HLS for FPGA devices. The yellow boxes show OpenVX functions implemented as OpenCL functions and the green ones the OpenVX functions implemented as kernels. The bottom boxes show host command queues, Qn, that manage the kernels.

the graph. This approach is implemented and named differently by each FPGA vendor; e.g., Intel and Xilinx adopt channels and pipes, respectively.

These two approaches evidence the portability problem between manufacturers and the limitations of standard OpenCL API, whereby each FPGA manufacturer extensions help to optimize and guide the compilers through bitstream generation. Even, sometimes, these extensions are different per FPGA device family limiting portability [26].

In terms of performance, the use of the aforementioned channel approach allows higher throughput and lower latency, but due to restrictions of the OpenCL standard, generating portable and easy to use libraries is a challenge. For example, AFFIX implements OpenVX graphs with single-input single-output host pipes [24] curtailing the OpenVX specification, which defines multiple-input multiple-output edges.

Besides OpenCL, a more flexible HLS language is C/C++. Although C/C++ suffers the portability restrictions between manufacturers, the programming details can be hidden to the programmer under wrapper layers.

Table 1
Programming flow alternatives to implement the OpenVX standard.

Programming Flow	Manufacturer portable	Deep Pipeline	Host Dependency
Standard OpenCL	✓	✗	LOW
OpenCL Channels	✗	✓	HIGH
HiFlipVX	✗	✓	-
This Work	✓	✓	LOW

For Xilinx devices, HiFlipVX implements OpenVX using C/C++, enabling a highly parametrizable library. However, to complete an efficient and portable OpenVX specification, it is necessary to port the library to Intel devices. The differences between C/C++ standards and compiler, such as OpenCL, are not trivial, showing differences between manufacturers. Also, FPGA families present a wide variety of designs, from simple embedded devices to high-performance ones with external memory and ports, specially oriented to HPC applications.

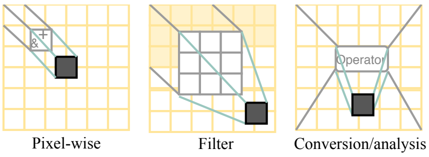
This work overcomes those limitations. Specifically, HiFlipVX achieves both portability, supporting two of the main FPGAs manufacturers, and performance, by coalescing OpenVX nodes in a single OpenCL/RTL element maximizing pipeline deep for Intel FPGAs as shown in Figure 2c. With this strategy, OpenVX applications overcome the pipeline depth limitations in Standard OpenCL (Figure 2a) and reduces the host dependency on OpenCL Channels implementation (Figure 2a). This property is specially crucial for Intel FPGA devices as Table 1 shows.

4. HiFlipVX

HiFlipVX is an open source HLS FPGA library for image processing applications [15]. It has been extended for object recognition, which involves feature detection [14] and neural networks [13]. HiFlipVX is a C++ based library containing 53 functions, which are highly optimized and parametrizable using templates. Most of its functions, or object kernels, are based on the OpenVX standard. They are implemented to be streaming capable with stream data objects, on edges, to link kernel instances as nodes in a graph. It extends the OpenVX based functions by additional parameters, such as vectorization, or more options, such as additional data types.

The functions in HiFlipVX can be categorized in pixelwise, filter, analysis, and conversion functions as Figure 3 shows. Pixelwise functions process the input images pixel by pixel, like adding two images together. Filter functions work in a window on the input image, like in a Gaussian filter. The conversion functions change the image by scaling it or changing the image format. The analysis functions usually have to perform a complete analysis of the input image, such as creating a histogram. Other functions that operate, for example, on feature vectors [14] or on tensors [13] can be classified into the mentioned categories.

The library was designed to be as vendor independent as possible. Since no external libraries are required, it can also run on a normal CPU. Nevertheless, it performs better in terms of resources and execution time than the vendor-specific library, xfOpenCV, on Xilinx FPGAs [15]. Additionally, the library is extended with pragmas and macros for acceleration on Xilinx FPGAs. These directives are used for pipelining, partitioning arrays, selecting specific resources



Pixel-wise Filter Conversion/analysis

Figure 3: Image functions categories implemented in HiFlipVX.

and interfacing between functions. The functions of HiFlipVX were used for various applications, such as in a toolchain. [23], or an operating system [20]. Akgün et al. show that the use of vectorization increases not only performance but energy efficiency as well [1].

5. Evaluation Methodology

All experiments have been run on two high-end FPGAs: an Intel Stratix 10 GX Development Kit (1SG280LU2F50E-2VG) with 2GB of HiLo DDR4 DRAM @933.3MHz and an Intel Stratix 10 MX Development Kit (1SM21BHU2F53E2-VGS1) with 32x 256MB HBM memory banks @800MHz. Both boards use the PCIe Gen3 x8 to connect with the host CPU. Table 2 summarizes FPGA resources specification.

Table 2
FPGA Resources for Stratix 10 GX and MX.

FPGA model	ALUTs	FFs	RAMs	DSP
Stratix 10 GX	1866240	3732480	11721	5760
Stratix 10 MX	1405440	2810880	6847	3960

This work evaluates the performance portability of HiFlipVX with parameters such as latency, initiation interval (II), and resource estimation from RTL compilation using i++ HLS compiler V. 19.4. The FPGA core power measurements use the Board Test System application provided by Intel, with a 1 second sampling rate. To ensure power accuracy, kernels run at least 1 minute to obtain measurements. For most experiments, the Stratix 10 GX was selected as the reference board, since the only difference with the MX is the DRAM vs. HBM banks.

Our benchmark suite comprises four representative OpenVX graphs, including all the categories of Figure 3, from the Intel OpenVX and Khronos samples:

- Canny edge detector: Popular multi-stage algorithm for edge detection and suppressing noise.
- Auto-contrast: Algorithm to improve contrast in images, adjusting the image intensity.
- Census transform: A common algorithm for correspondence problem used in stereo image processing for disparity calculations [29].
- Skin tone detection: Algorithm to detect human white skin tone.

Finally, these benchmarks are also used to compare with existing state-of-the-art approaches running them on the same FPGA, except the skin tone which is not implemented by other works.

6. Tuning HiFlipVX for Intel FPGAs

The OpenVX specification provides a high level abstraction to easily implement computer vision applications on multiple devices. The OpenVX objects are designed for dynamic applications, so the runtime provides support to manage objects during execution. However, since bitstream generation takes a long time, on FPGAs, the verification and optimization of OpenVX graphs has to be statically performed at compile time.

The OpenVX standard leaves the optimization process to vendors. In the case of HiFlipVX, the new implementation supports programmer's optimizations through specialized versions of its template-based API for each vendor. So, programmers can tune the OpenVX applications according to the FPGA platform with minimal changes in the user-facing code. Such portability from Xilinx to Intel implementation has required changes in the implementation of three OpenVX components: execution model, kernels, and edges. Kernel nodes are the compute part of the graphs, while edges have the memory management with virtual and image objects which potentially improves speed up.

6.1. Execution model

For Intel FPGAs, HiFlipVX synthesizes every graph as a single kernel (Figure 2c). The system of task, a propriETARY Intel API, enables task-level pipelining, allowing asynchronous nodes to create a graph for Intel FPGAs. On the contrary, the Xilinx specialization uses the `HLS dataflow` pragma for function or loop level parallelism.

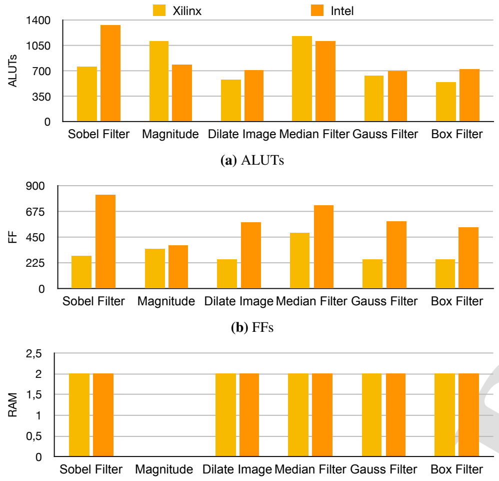
6.2. Kernels

HiFlipVX kernel nodes are implemented with C++ functions, so the first step to maintain kernel performance and properly guide the compilation process is to add specific *translations* of Xilinx's pragmas to their Intel counterparts. Specifically, the next two pragmas and component attributes are used:

- `HLS array_partition`/`hls_register`: forces the compiler to generate variables as registers.
- Loop pragmas: the difference is the location in the code. These pragmas are inserted after and before the loop, for Xilinx and Intel, respectively.

The resource utilization comparison of the HiFlipVX for Xilinx [15] and Intel devices shows that the ALUTs resource usage is similar, less than 15% variation for 6 representative OpenVX functions (all running at the same 100MHz frequency), except Sobel Filter, 27% difference, as depicted in Figure 4a.

Since the core programmable unit in Intel Stratix architecture packs four-input LUTs and registers (FFs), the FFs usage in Figure 4b shows a similar tendency as ALUTs. The RAM usage in Figure 4c, shows the same number of blocks although the RAM sizes are different, 18K in Xilinx and 20K in Intel; the similarities are attributed to SIMD vectorization of 1 which synthesizes arrays and variables as registers. These results evidence the differences between architectures and HLS tools; e.g., Xilinx LUTs are capable of self-split to implement two separated logic functions, unlike Intel that has dedicated ALUTs to improve routing time in complex designs [7].



(a) ALUTs

(b) FFs

(c) RAM

Figure 4: Resource comparison between Intel and Xilinx [15] FPGA at 100MHz and vectorization equal to 1, for 6 sample OpenVX functions.

6.3. Edges

For Xilinx, HiFlipVX implements optimized communications through streaming with *HLS STREAM* pragma. The pragma creates FIFOs or double buffers to transfer data between functions or loops in a data flow area and it uses pass-by-pointers for kernel node parameters. In general, these choices guarantee an Initiation Interval, II, equal to 1 cycle and low latency for filter-type kernels [15].

The lack of equivalent pragmas for streaming communications in Intel API and the pass-by-pointer as parameters can result in kernels with poor performance, constraining the II up to 114 cycles, because the HLS tool generates a single Avalon Memory-Mapped (MM) Master interface with a single arbiter for all variables [4].

When function parameters are passed-by-reference, which are more suitable for Intel [11], the II reduces to 1 cycle, substantially improving the pipeline performance. The first two groups of bars in Figure 5 show a 114× cycle difference between the pass-by-pointer and pass-by-reference for a 3x3 filter.

Reference parameters do not support concurrency requirements of nodes in OpenVX graphs. To support them, the Intel system of task with *stream* as function parameters allows nodes to run asynchronously. Streams reach an II of 1 cycle and, in practice, resulting in Avalon streaming interfaces which provides high-bandwidth and low latency communication.

Comparing the *streams* with *reference*, streams latency is up to 2 × higher because system of task adds control logic in kernel pipeline to communicate among graph nodes. Figure 5 shows the impact on both II and latency of all the interface changes: passing arguments by reference and stream communication among kernels.

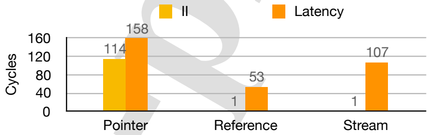


Figure 5: Latency and Initiation Interval for interface optimizations on edges in a 3x3 filter function (lower is better).

In terms of code implementation, Intel stream interface has 3 specific data types: *stream_in* for inputs, *stream_out*, for outputs, and *stream* for general interconnect between kernel nodes. To achieve the portability, the Listing 1 shows the data type redefinition of the vx_image based on templates which allows to adapt the hardware with the vectorization factor (V) and the capacity of stream buffers (buff_cap). This implementation hides the hardware interface details to programmers.

Listing 1: vx_image for virtual image implementation with Intel streams support

```
template<class T, const size_t V,
    int stream_type, uint buff_cap=256>

using vx_image=
    typename conditional<stream_type == vx_streamIn_e,
    ihc::stream_in<vx_image_t<T, V>>,
    typename conditional<stream_type==vx_streamOut_e,
    ihc::stream_out<vx_image_t<T,V>>,
    typename conditional<stream_type==vx_stream_e,
    ihc::stream<vx_image_t<T,V>, ihc::buffer<buff_cap>>,
     vx_image_t<T,V> > ::type>::type>::type;
```

Virtual image objects implemented with *streams* are limited to access by reference, and the use of arrays of *streams* are not allowed. Also, multiple reads from a stream by different nodes require to duplicate the number of edges in the FPGA. For this reason, a custom internal kernel `vxSplit` is needed to concurrently feed multiple kernels with a single copy of the data-stream references.

Contrary to virtual image objects, images references allow direct user access, which creates an opaque reference to an image buffer [5]. In Intel FPGA, the user can access data through external ports using the Avalon MM buses.

In embedded FPGAs, *stream* interfaces with input and output qualifiers are enough to control I/O ports. However, discrete devices with an external memory, as DRAM, require memory IP controllers. To manage external DRAM memories, HiFlipVX takes advantages on existing host drivers for OpenCL/SYCL to perform the required transactions. Also, those transfers are transparently instantiated with two custom kernels.

The DRAM interfaces are created with the *ihc::mm_master* to specify the external Avalon MM data bus interconnection to the OpenCL/SYCL drivers. Listing 2 shows the vx_image to create images for FPGAs with DRAM support. The data bus size (WIDTH_MEM) is parametrized with the specification of DRAM memory controller from the BSP (Board Support Package), and PORT enumerates the bus interface. The last template parameter, emb_x, advises the compiler whether the interface is embedded or not, for Xilinx devices it is always true.

Listing 2: vx_image for image implementation with Intel DRAM support

```
template<class T, const uint WIDTH_MEM, uint V=1, uint
     PORT=1, uint emb_x=1>

using vxCreateImage =
    typename conditional < emb_x == 0,
    ihc::mm_master<vx_image<T, V>,
    ihc::aspace<PORT>, ihc::awidth<WIDTH_MEM>,
    ihc::dwidth<32>, ihc::latency<0>,::maxburst<16>,
    ihc::align<64>, ihc::waitrequest<true>>,
    vx_image<T, V>>::type;
```

Image objects for DRAM generates load/store units for continuous and aligned memory accesses, user defined parameters as burst size with the coalescence parameter are available for user optimizations. Furthermore, the load/store controller allows to adjust technology differences between FPGA boards and maximizes DRAM bandwidth. Figure 6 shows the interfaces and load/store units required to interconnect a DRAM memory to HiflipVX graph.

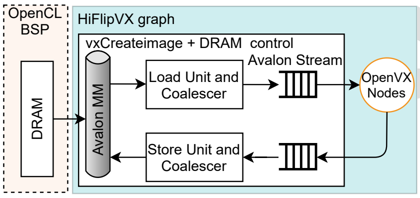


Figure 6: DRAM memory interconnection to a HiFlipVX graph.

To evaluate Load/Store units, Figure 7 plots the performance of Canny edge detector as a representative graph. It shows that high coalescence factors with very wide LSUs, > 512 bits, can reduce performance up to 7 ×, because the compiler heuristic generates a non-aligned controller access. In load/store units with a bus width smaller than 512 bits, the maximum DRAM burst is underused, except in 64 bits which is the same bus width as DRAM (dq).

Once an OpenVX graph has been programmed in C/C++ with HiFlipVX, the compilation flow depends on the target FPGA. For an embedded FPGA, the FPGA IP can be generated after the RTL generation, and for an Intel discrete FPGA, the IP is coupled to a BSP, which is part of the OpenCL and SyCL drivers, to enable communication with a host CPU.

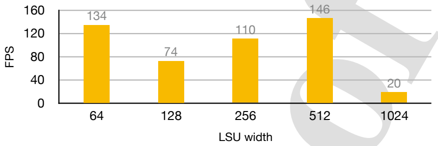


Figure 7: Performance (Frames Per Second) of canny edge detector with a HD image varying coalescing to read DRAM memory (LSU width), higher is better.

OpenCL and SyCL Library feature allows including RTL modules into function kernels packaged into an library object (.lib). However, the system of tasks used in HiFlipVX is not supported yet. To overcome this problem, the compilation flow has an additional step, supported with a tool extension in HiFlipVX that takes two inputs: 1) an XML file with the BSP memory port descriptions, and the RTL from HiFlipVX, both of them compatible with the target FPGA.

The tool extension output enables the library generation with Intel standard *aoc* tools. As result, the HiFlipVX libraries objects are ready to be used in OpenCL/SyCL kernels. Since OpenCL backend implementation is more mature than SyCL, it has been chosen to be evaluated in this work. Figure 8 shows the compilation flow to couple the Intel HiFlipVX graph to a heterogeneous system (right path) and how the Xilinx flow is unaffected (left path).

7. Results

This section starts analyzing how the new HiFlipVX implementation behaves when the graph complexity changes. Then, it evaluates HiFlipVX running 4 representative OpenVX graphs and, finally, compares this work with two state-of-the-art proposal.

7.1. HiFlipVX Scalability Analysis

To assert how system of tasks and deep pipelines impact on graph scalability, the first stage of the SIFT feature detector is used as a synthetic graph benchmark. This multi-Gaussian graph applies multiple times a Gaussian filter to an image stream [17]. The benchmarks allow us to tune the depth of the resulting kernel pipeline by adding Gaussian filtering steps, one after the other.

Figure 9a plots the impact of the number of filter nodes for the multi-Gaussian graph (kernel pipeline depth) on execution time and FPGA frequency. From 2 to 16 filters, memory latency hides computation which flattens the execution time. After that point, 16, execution time increases almost linearly with the number of filters, showing good scalability. Please note the slight frequency reduction for large number of filters also contributes to the larger execution time.

Resource usage is shown in Figure 9b, which increases

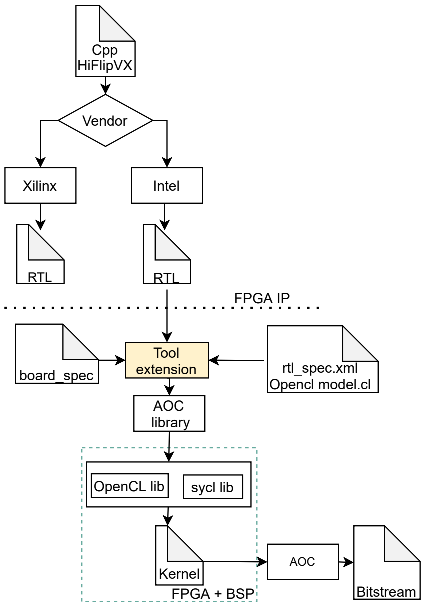
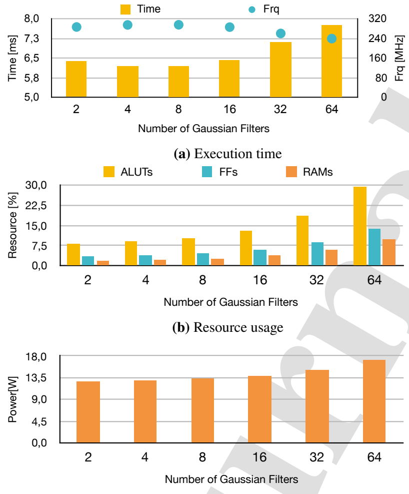


Figure 8: HiFlipVX programming and compilation flow for Xilinx and Intel FPGAs.



(a) Execution time

(b) Resource usage

(c) Kernel Power

Figure 9: Impact of node scalability on a) execution time; b) frequency and resource utilization; and c) power consumption for the Multi-Gaussian synthetic benchmark using the Stratix 10 GX.

linearly, with a growing rate of 0.33, 0.17, and 0.13 for ALUTs, FFs, and RAMs resources, respectively. As a consequence, FPGA power raises with a growing rate of 74mW per additional Gaussian filter stage as is shown in Figure 9c. In summary, HiFlipVX with the system of task scales well without adding any extra overhead increasing the graph complexity.

7.2. OpenVX Application Resource Utilization

This section analyzes resource usage (per-kernel) of four representative applications: Canny edge, Autocontrast, Census transform, and Skin tone detection. Figure 10 shows the graph diagram for all of them. For the sake of clarity, the custom internals kernels, enabling DRAM and splitting data streams (vxSplit) described in Sec. 6, are not depicted in the graphs.

7.2.1. *Canny Edge*

The Canny edge detector, Figure 10a, is a multi-node graph algorithm that extracts the edge information from images. In HiFlipVX, its implementation consists of 5 nodes. Table 3 shows the estimated resource usage from the Intel HLS compiler report for all Canny edge nodes[2].

Table 3
Estimated resource usage for each OpenVX function in Canny edge graph using HiFlipVX with a 4K image and vectorization factor of 8 on a Stratix 10 GX.

Function	ALUTs	FFs	RAMs	DSP
Load Image Object	10553	39508	17	0
vxGauss	3627	5620	18	0
vxSobel	5907	8854	19	0
vxSplit	221	117	2	0
vxMagnitud	5907	8966	4	8
vxPhase[3]	165	133	1	0
vxNonMaxSuppression	4605	5842	18	0
Store Image Object	4177	11949	18	0

The image objects are the most resource demanding function since it track multiple external memory request at a time, trying to group access before being send to the memory controller.

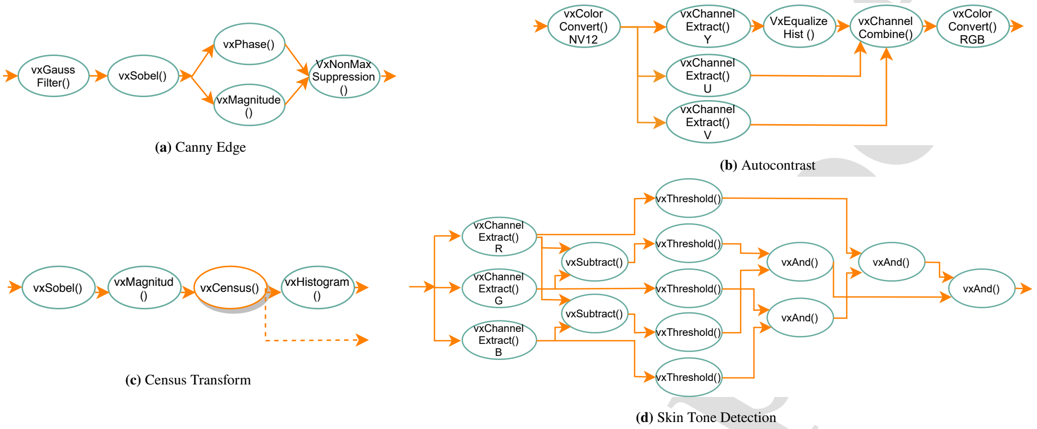
7.2.2. *Autocontrast*

Autocontrast, requiring to extend HiFlipVX to support the graph from Figure 10b with two new kernels for color conversions: NV12 to RGB and RGB to NV12, and EqualizeHist.

Autocontrast requires more RAM resources than other graphs because the intensity channel (Y) is stored in RAM memory until histogram is calculated. This strategy avoids stalls in streams at expense of higher resource usage that mainly depends on the input image size; e.g., if the image size changes from HD to 4K, RAM usage increases by 4 ×. HiFlipVX enables the user to provide FPGA tuning parameters. For example, in this case, the code includes a hint to implement the DRAM access coalescence with a LSU width

[2] Since all FPGAs used in this work are from the same family, Stratix 10, the resource estimation on HiFlipVX graph are equal, and from here on, all results corresponds to the Stratix 10 GX FPGA, except when noted.

[3] Orientation only for 4 gradient directions.



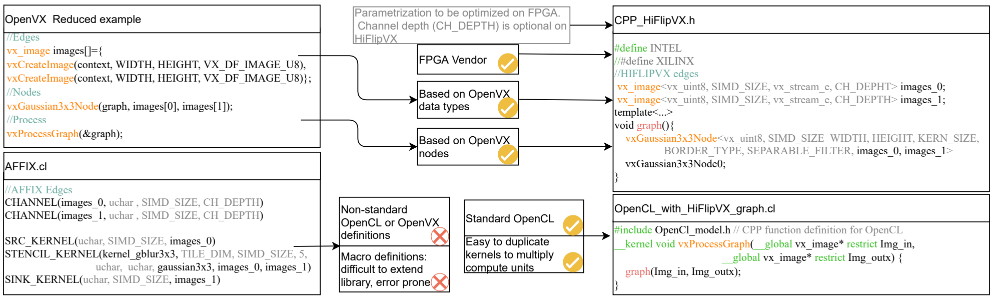
(a) Canny Edge

(b) Autocontrast

(c) Census Transform

(d) Skin Tone Detection

Figure 10: OpenVX application graph diagrams. a) Canny edge detector, b) Autocontrast image, c)Census transform, d) Skin tone detection.



OpenVX Reduced example

```
//Edges
vx_image images[]={
vxCreateImage(context, WIDTH, HEIGHT, VX_DF_IMAGE_U8),
vxCreateImage(context, WIDTH, HEIGHT, VX_DF_IMAGE_U8)};
//Nodes
vxGaussian3x3Node(graph, images[0], images[1]);
//Process
vxProcessGraph(&graph);
```

AFFIX.cl

```
//AFFIX Edges
CHANNEL(images_0, uchar , SIMD_SIZE, CH_DEPTH)
CHANNEL(images_1, uchar , SIMD_SIZE, CH_DEPTH)

SRC_KERNEL(uchar, SIMD_SIZE, images_0)
STENCIL_KERNEL(kernel_gblur3x3, TILE_DIM, SIMD_SIZE, 5,
                    uchar,  uchar, gaussian3x3, images_0, images_1)
SINK_KERNEL(uchar, SIMD_SIZE, images_1)
```

CPP_HiFlipVX.h

```
#define INTEL
//#define XILINX
//HIFLIPVX edges
 vx_image<vx_uint8, SIMD_SIZE, vx_stream_e, CH_DEPHT> images_0;
 vx_image<vx_uint8, SIMD_SIZE, vx_stream_e, CH_DEPTH> images_1;
template<...>
void graph(){
   vxGaussian3x3Node<vx_uint8, SIMD_SIZE  WIDTH, HEIGHT, KERN_SIZE,
              BORDER_TYPE, SEPARABLE_FILTER, images_0, images_1>
   vxGaussian3x3Node0;
}
```

OpenCL_with_HiFlipVX_graph.cl

```
#include OpenCl_model.h // CPP function definition for OpenCL
__kernel void vxProcessGraph(__global vx_image* restrict Img_in,
                             __global vx_image* restrict Img_outx) {
   graph(Img_in, Img_outx);
}
```

Parametrization to be optimized on FPGA. Channel depth (CH_DEPTH) is optional on HiFlipVX
FPGA Vendor ✓
Based on OpenVX data types ✓
Based on OpenVX nodes ✓
Non-standard OpenCL or OpenVX definitions ✗
Macro definitions: difficult to extend library, error prone ✗
Standard OpenCL ✓
Easy to duplicate kernels to multiply compute units ✓

Figure 11: Code comparison between a reduced version of OpenVX, AFFIX, and HiFlipVX. OpenVX definitions and FPGA optimization parameters are marked in orange and grey, respectively.

Table 4
Estimated resource usage for each OpenVX function in Autocontrast graph using HiFlipVX, with a HD image and vectorization factor of 1.

Function	ALUTs	FFs	RAMs	DSP
Load Image Object	934	3025	16	0
vxColorConvert(NV12)	1273	1744	0	1
vxSplit	130	103	0	0
vxChannelExtract	143	119	0	0
vxEqualizeHist	2584	3874	1029	0
vxChannelCombine	173	143	0	0
vxColorConvert(RGB)	1037	1268	0	0
Store Image Object	1102	3605	18	0

of 64 bits to save resources and compensate the extra DRAM usage. The Table 4 shows the resource for each function in the Autocontrast graph.

7.2.3. *Census Transform*

Census transform is not part of the OpenVX standard, so we added it to the HiFlipVX library. The implementation concatenates several filters as Canny does. Table 5 shows the estimated resource usage for Census transform functions, while Table 3 shows the usage for shared functions between Census transform and, above explained, Canny.

Table 5
Estimated resource usage for each function in Census transform using HiFlipVX, with a 4K Image and vectorization factor of 8.

Function	ALUTs	FFs	RAMs	DSP
Load Image Object	10569	39520	17	0
vxCensus	179	208	0	0
vxHistogram	960	16924	2	0
Store Image Object	2095	5501	18	0

7.2.4. *Skin Tone Detection*

The last evaluated graph is Skin tone detection, which requires threshold objects to produce output Boolean images. The graph is composed by 14 nodes of four different OpenVX kernels that process 8 bit data. Table 6 shows the resources for each function in the Skin tone graph.

Table 6
Estimated resource usage for each OpenVX function in Skin tone graph using HiFlipVX, with an HD image and vectorization factor of 1.

Function	ALUTs	FFs	RAMs	DSP
Load Image Object	3675	17257	16	0
vxAndNode	152	119	0	0
vxSubtract	230	153	0	0
vxThresholdNode	154	120	0	0
Store Image Object	2380	6936	18	0

7.3. OpenVX Application Analysis

The new HiFlipVX implementation simplifies the adoption of different FPGAs. For example, this section evaluates execution time, frequency, power, and energy on two FPGA devices: Stratix 10 GX (S10GX) and Stratix 10 MX (S10MX). The main difference between the boards is the global memory. While the S10GX has one DRAM bank with a data port width of 512 bits, the S10MX has a HBM multi-banked memory composed by 32 DRAM banks and a data port width of 256 bits per bank. Running on both boards only required to change the *vxCreateImage* port declaration.

Table 7 shows the execution time, frequency, power and energy of the four graphs. For all of them, the S10GX has higher frequencies and lower execution times, with time gains between 1.4 and 6,8%. Since all graphs are compute bound; e.g., in Canny edge and Census transform the maximum memory bandwidth used is 2.4GBs for S10GX and S10MX, the HBM memory does not provide any advantage in spite of using one memory bank per variable. Most probably, the same Stratix 10 architecture explains the close results. For power and energy, in all but Autocontrast, the S10GX consumes more energy and with the lower execution time increases average power, up to 18%. The higher energy consumption in Autocontrast by the S10MX may be due to the BSP differences and the required extra RAM that increases routing complexity.

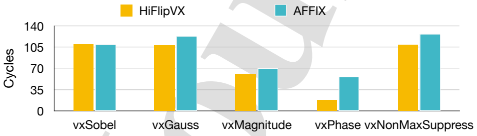


Figure 12: Latency of canny edge for HiFlipVX and AFFIX using an Stratix 10 GX FPGA.

7.4. Comparison with Existing Approaches

AFFIX [24] is a previous proposal that implements OpenVX graphs. It relies on OpenCL channels to offer an implementation based on OpenVX standard, as shown in Figure 2b. The use of OpenCL limits the programmability of AFFIX. Comparing the graph codes from Figure 11, AFFIX, lower left, relies on OpenCL macros that are error-prone, difficult to maintain, and moves away from the clarity of OpenVX, upper left. In contrast, our work allows to use a well-formed C++ code, the same language as OpenVX API, to program graphs using OpenVX standard with templates to optimize hardware generation. In any case, in order to integrate HiFlipVX graphs to a host CPU, HiFlipVX can have a simple OpenCL interface called from a single queue command to execute the graph.

In order to comparatively analyse performance against our proposal, we modified AFFIX to communicate host and FPGA kernels through the on-board DRAM instead of using the Intel host pipe extension to directly communicate between the host and FPGA kernels. Although host pipes reduce latency overhead, they have two limitations: they are only supported on a few Arria 10GX development kits [9], and also, each pipe can only have one input and one output port. This second fact limits graph implementations; e.g., the Census transform was reduced to one output as shown in Figure 10c where the AFFIX implementation follows the dotted line and ignores the solid one. To compare with this work, it is mandatory to replace the pipes with equivalent DRAM input/output to run benchmarks on Stratix10 GX and Stratix 10 MX boards.

In Table 8, it can be observed that HiFlipVX reaches a speed-up of 3.4× and 3.6 × for Canny Edge and Census. In case of Autocontrast, it was not possible to use the same implementation for both AFFIX and ours, so that, they are hardly comparable. Our approach is behaving a 20% worse since synthesized frequency is lower in comparison to AFFIX (Figure 13). This penalization on frequency is due to a higher consumption of resources (RAM) in the HiFlipVX implementation.

Comparing the energy of the proposals, HiFlipVX dissipates 23% less power than AFFIX in Census transform case, our assumption is that, in HiFlipVX, the dispatch circuits to connect nodes with host are minimized[4], also, in Canny edge and Census transform the HiFlipVX frequency is lower than AFFIX, at least 50% in Canny. In Autoconstrast, with the worst performance, it is only 10% less energy efficient.

Comparing the latency between both implementations in Figure 12, HiFlipVX shows a 10% of improvement on average. One of the pipeline speed-up sources comes from the hyper-optimized loop structure which is enabled by default in the HLS compiler. The use of hyper-registers on an application has demonstrated a performance gain of 1.4 × on Stratix10 devices compared with previous FPGAs generation [27]. Although the compiler tries to apply this technique in both AFFIX and HiFlipVX, in case of AFFIX, the use of

[4] Compilation reports state this difference in the dispatch logic between AFFIX and HiFlipVX

Table 7
HiFlipVX results on a Intel Stratix 10 GX and Intel Stratix 10 MX using a 4K image.

OpenVX Application	Stratix 10 GX				Stratix 10 MX			
	Time[ms]	Frq[MHz]	Power [W]	Energy[mJ]	Time[ms]	Frq[MHz]	Power[W]	Energy[mJ]
Canny edge	6.8	310	13.2	89.8	7.3	293	11.3	76.5
Census	6.8	331	12.9	87.7	6.9	326	10.9	74.6
Autocontrast	23.1	301	13.1	302.6	23.9	294	13.8	318.1
Skin tone	33.2	343	12.8	424.9	35.7	315	10.9	361.5

Table 8
Comparison between HiFlipVX and AFFIX on a Intel Stratix 10 GX and Intel Stratix 10 MX using a 4K image.

OpenVX Application	Stratix 10 GX		Stratix 10 MX	
	$\frac{Time\_HiFlipVX}{Time\_AFFIX}$	$\frac{Energy\_AFFIX}{Energy\_HiFlipVX}$	$\frac{Time\_HiFlipVX}{Time\_AFFIX}$	$\frac{Energy\_AFFIX}{Energy\_HiFlipVX}$
Canny edge	3.2	2.4	3.6	1.9
Census	3.6	1.7	3.4	1.8
Autocontrast	0.8	0.9	0.8	0.7

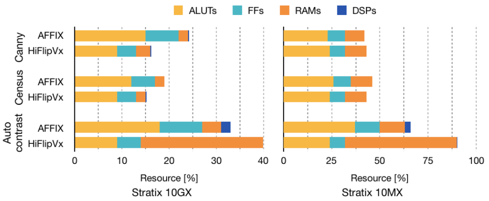


Figure 13: Resource usage per logic unit relative the total units on Stratix 10 GX and Stratix 10 MX for AFFIX and HiFlipVX implementations.

OpenCL channels is inhibiting this optimization.

Figure 13 shows the resource consumption: ALUT, RAM, FF, and DSP; of AFFIX and HiFlipVX. In case of Canny edge, Census transform, and Autocontrast, AFFIX has a higher utilization of ALUTs and FFs resources than HiFlipVX. In opposition to HiFlipVX, in AFFIX, OpenCL generates a "kernel dispatch logic" for each OpenVX kernel to communicate with the host, which is responsible for an increase of 1463 ALUTs and 1467 FFs per kernel node. In the case of HiFlipVX, kernel nodes are collapsed in a single kernel with a single dispatch logic with saves from 4 to 24% of resources per kernel in the evaluated graphs on the Stratix 10 GX. On the Stratix 10 MX, the difference is less than 5% between implementations.

In Autocontrast, the amount of RAM resources in HiFlipVX is 4 × bigger as it is sensitive to image size. In contrast, AFFIX implementation prefers to split the pipeline and read twice from external memory instead of using RAM resources. Concerning to DSPs resources, in HiFlipVX the color conversion is implemented with a 8-bit approximation [18] that does not require DSPs for float operations in contrast to AFFIX.

At last, we compare our proposal against the traditional OpenCL model, depicted in Figure 2a, used by the Chai benchmark [8] for Canny edge. Our approach, HiFlipVX reaches a speedup of 9 × in comparison to Chai's. There are two limiting factors that justify this results in Chai's: communication between nodes through external memory is slower and shallow kernels (short pipelines) do not fully exploit FPGA paralelism.

8. Conclusions

One of the main features of OpenVX is the portability among devices. However, on FPGA devices, providing cross-platform support remains a challenge. This paper presents a cross-platform OpenVX library for FPGAs based on HiFlipVX library, which originally only targeted Xilinx devices. This new version efficiently supports Intel FPGAs exploiting the novel Intel's system of tasks to coalesce OpenVX nodes into accelerated graphs on Intel FPGAs.

The new implementation introduces a novel compilation flow that integrates the expressiveness of OpenVX graphs in C/C++ with the performance of OpenCL kernels. Also, applications can interoperate with OpenCL and SyCL code. With these 3 aspects, the library gains flexibility to support multiple FPGA architectures and devices with conventional and High Bandwidth Memories.

In terms of resource utilization, on Intel devices, the enabled optimizations save around 1.5% of ALUTs usage per node in graphs versus the standard OpenCL approach with one kernel per node, since the host hardware control communication is only generated for the complete HiFlipVX graph application. Compared with the state-of-art, HiFlipVX performs up to 3.6 and 9.6 × faster than AFFIX and Chai, respectively. Energy results also reflects the successful implementations with savings up to 2.4 ×.

Acknowledgment

All authors acknowledge support from grants HiPEAC collaboration grant 2019, PID2019-105660RB-C21 / AEI / 10.13039/501100011033 from Agencia Estatal de Investigación (AEI) and European Regional Development Fund (ERDF), gaZ: T58_20R research group from Aragón Government and European Social Fund (ESF), 2014-2020 "Construyendo Europa desde Aragón" from European Regional Development Fund (ERDF), and Santander-UZ grants program.

References

[1] Akgün, G., Kalms, L., Göhringer, D., 2020. Resource efficient dynamic voltage and frequency scaling on xilinx fpgas, in: International Symposium on Applied Reconfigurable Computing (ARC), Springer. pp. 178–192. doi:10.1007/978-3-030-44534-8_14.

[2] Chugh, N., Vasista, V., Purini, S., Bondhugula, U., 2016. A DSL Compiler for Accelerating Image Processing Pipelines on FPGAS. PACT doi:10.1145/2967938.2967969.

[3] Dávila-Guzmán, M.A., Tejero, R.G., Villarroya-Gaudó, M., Gracia, D.S., Kalms, L., Göhringer, D., 2021. A cross-platform openvx library for FPGA accelerators, in: 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2021, Valladolid, Spain, March 10-12, 2021, IEEE. pp. 75–83. URL: https://doi.org/10.1109/PDP52278.2021.00020, doi:10.1109/PDP52278.2021.00020.

[4] Dávila-Guzmán, M.A., Gran Tejero, R., Villarroya-Gaudó, M., Suárez Gracia, D., 2020. Analytical model of memory-bound applications compiled with high level synthesis, in: FCCM, pp. 218–218.

[5] Giduthuri, R., Pulli, K., 2016. OpenVX: A Framework for Accelerating Computer Vision, in: SIGGRAPH ASIA 2016 Courses, pp. 14:1–14:50. doi:10.1145/2988458.2988513.

[6] HajiRassouliha, A., Taberner, A.J., Nash, M.P., Nielsen, P.M., 2018. Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms. Signal Processing: Image Communication 68, 101–119. doi:10.1016/j.image.2018.07.007.

[7] HardwareBee, 2020. Xilinx vs. Intel High-End FPGA Series Comparison. URL: https://hardwarebee.com/xilinx-vs-intel-high-end-fpga-series-comparison/.

[8] Huang, S., Chang, L.W., El Hajj, I., Garcia de Gonzalo, S., Gómez-Luna, J., Chalamalasetti, S.R., El-Hadedy, M., Milojicic, D., Mutlu, O., Chen, D., Hwu, W.m., 2019. Analysis and modeling of collaborative execution strategies for heterogeneous cpu-fpga architectures, in: ICPE '19, pp. 79—-90.

[9] Intel, 2020a. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide 19.4.

[10] Intel, 2020b. Intel® High Level Synthesis Compiler Pro Edition 19.4.

[11] Intel, 2020c. Intel® High Level Synthesis Compiler Pro Edition 19.4, Best Practice Guide.

[12] Jiang, J., Wang, Z., Liu, X., Gómez-Luna, J., Guan, N., Deng, Q., Zhang, W., Mutlu, O., 2020. Boyi: A systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs. FPGA 2020 , 299–309doi:10.1145/3373087.3375313.

[13] Kalms, L., Amini Rad, P., Ali, M., Göhringer, A.I.D., 2021. A parametrizable high-level synthesis library for accelerating neural networks on fpgas. Journal of Signal Processing Systems , 1–27doi:10.1007/s11265-021-01651-5.

[14] Kalms, L., Göhringer, D., 2021. Accelerated high-level synthesis feature detection for fpgas using hiflipvx, in: Towards Ubiquitous Low-power Image Processing Platforms. Springer International Publishing, pp. 115–135. doi:10.1007/978-3-030-53532-2_7.

[15] Kalms, L., Podlubne, A., Göhringer, D., 2019. Hiflipvx: an open source high-level synthesis fpga library for image processing, in: International Symposium on Applied Reconfigurable Computing (ARC), Springer. pp. 149–164. doi:10.1007/978-3-030-17227-5_12.

[16] Li, J., Chi, Y., Cong, J., 2020. HeteroHalide: From image processing DSL to efficient FPGA acceleration. FPGA 2020 - 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays , 51–57doi:10.1145/3373087.3375320.

[17] Lowe, D.G., 2004. Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vision 60, 91–110.

[18] Microsoft, 2018. Recommended 8-Bit YUV Formats for Video Rendering.

[19] Özkan, M.A., Ok, B., Qiao, B., Teich, J., Hannig, F., 2020. Hipaccvx: wedding of openvx and dsl-based code generation. Journal of Real-Time Image Processing URL: https://doi.org/10.1007/s11554-020-01015-5, doi:10.1007/s11554-020-01015-5.

[20] Podlubne, A., Haase, J., Kalms, L., Akgün, G., Ali, M., Khan, H., Kamal, A., Göhringer, D., 2018. Low power image processing applications on fpgas using dynamic voltage scaling and partial reconfiguration, in: International Conference on Design and Architectures for Signal and Image Processing (DASIP), IEEE. pp. 64–69. doi:10.1109/DASIP.2018.8596910.

[21] Pu, J., Bell, S., Yang, X., Setter, J., Richardson, S., Ragan-Kelley, J., Horowitz, M., 2017. Programming heterogeneous systems from an image processing DSL. ACM Transactions on Architecture and Code Optimization 14, 1–25. doi:10.1145/3107953, arXiv:1610.09405.

[22] Reiche, O., Ozkan, M.A., Membarth, R., Teich, J., Hannig, F., 2017. Generating FPGA-based image processing accelerators with Hipacc: (Invited paper). ICCAD doi:10.1109/ICCAD.2017.8203894.

[23] Sadek, A., Muddukrishna, A., Kalms, L., Djupdal, A., Podlubne, A., Paolillo, A., Göhringer, D., Jahre, M., 2018. Supporting utilities for heterogeneous embedded image processing platforms (sthem)): An overview, in: International Symposium on Applied Reconfigurable Computing (ARC), Springer. pp. 737–749. doi:10.1007/978-3-319-78890-6_59.

[24] Taheri, S., Behnam, P., Bozorgzadeh, E., Veidenbaum, A., Nicolau, A., 2019. AFFIX: Automatic acceleration framework for FPGA implementation of OpenVX vision algorithms, in: FPGA'19, p. 252–261. doi:10.1145/3289602.3293907.

[25] Taheri, S., Heo, J., Behnam, P., Chen, J., Veidenbaum, A., Nicolau, A., 2018. Acceleration framework for fpga implementation of openvx graph pipelines, in: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 227–227.

[26] Voss, N., Becker, T., Tilbury, S., Gaydadjiev, G., Mencer, O., Nestorov, A.M., Reggiani, E., Luk, W., 2020. Performance portable fpga design, in: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, p. 324.

[27] Woods, R., McAllister, J., Lightbody, G., Yi, Y., 2017. FPGA-based implementation of signal processing systems. Wiley Online Library.

[28] Xilinx, 2019. Xilinx OpenCV User Guide.

[29] Zabih, R., Woodfill, J., 1994. Non-parametric local transforms for computing visual correspondence, in: Computer Vision — ECCV '94, Springer. pp. 151–158.

[30] Zohouri, H.R., Podobas, A., Matsuoka, S., 2018. 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, p. 153–162.

Maria Angélica Dávila-Guzmán. has been a PhD student in the Department of Computer Science and System Engineering at the University of Zaragoza, Spain, since 2017. She received her Master's and Bachelor's degrees in Electronic Engineering from the University of Valle, Colombia, in 2010 and 2015, respectively. Her research interests lie in heterogeneous systems, high-level syntheses, and load balancing.

Lester Kalms is a PhD student in Chair of Adaptive Dynamic Systems at Technische Universität Dresden since 2017. He received his Master's and Bachelor's degrees in Computer Engineering from the Technical University of Berlin in 2013 and 2015. His research interests include methods and algorithms for an efficient programming and distribution of applications to heterogeneous systems, including FPGAs, GPUs, and CPUs using C++, OpenCL, compilers, and high-level synthesis.

Rubén Gran Tejero. graduated in Computer Science from the University of Zaragoza, Spain. He received his Ph.D. from the Polytechnic University of Catalonia (UPC), Spain, in 2010. Since 2010, he has been an Associate Professor at the Department of Computer Science and Systems Engineering, University of Zaragoza. His research interests include hard real-time systems, hardware for reducing worst-case execution time and energy consumption, efficient processor microarchitecture, and effective programming for parallel and heterogeneous systems.

María Villarroya-Gaudó. obtained her Ph.D. in 2005 at the Department of Electronics Engineering at the Autonoma University of Barcelona. She is an Associate Professor in Computer Architecture and Technology in the Department of Computer and Systems Engineering at the Universidad de Zaragoza. Her research interests include memory hierarchy and heterogeneous systems. Dr Vilarroya-Gaudó is member of Spanish Society of Computer Architecture (SARTECO).

Darío Suárez Gracia. (S'08–M'12) received his PhD degree in Computer Engineering from the Universidad de Zaragoza, Spain, in 2011. From 2012 to 2015, he was at Qualcomm Research Silicon Valley. Currently, he is an Associate Professor at the Universidad de Zaragoza. His research interests include parallel programming, heterogeneous computing, memory hierarchy design, energy-efficient multiprocessors, and fault-tolerance. Dr. Suárez Gracia is a member of the Aragon Institute of Engineering Research (I3A), the IEEE, the IEEE Computer Society, the ACM, and the HiPEAC European NoE.

Diana Göhringer is professor and holds the Chair of Adaptive Dynamic Systems at Technische Universität Dresden since 2017. She received her PhD (summa cum laude) in Electrical Engineering and Information Technology from the Karlsruhe Institute of Technology (KIT), Germany in 2011. She is author and co-author of over 150 publications in international journals, conferences and workshops. Her research interests include reconfigurable computing, multiprocessor systems-on-chip (MPSoCs), networks-on-chip, simulators/virtual platforms, hardware-software codesign and runtime systems.

Darío Suárez

Diana Goehringer

Lester Kalms

Click here to access/download;Author Photo;LesterKalms.JPG

María Angélica Dávila

María Villarroya

Click here to access/download;Author Photo;MariaVillarroyaGaudo.png

Ruben Gran

Declaration of interests

X The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: