A Cross-Platform OpenVX Library for FPGA Accelerators

Maria Angélica Dávila-Guzmán*, Rubén Gran Tejero‡, María Villarroya-Gaudó§ and Darío Suárez Gracia¶
DIIS-I3A, Universidad de Zaragoza
— HiPEAC Network of Excellence
e-mail: *angelicadg@unizar.es, ‡rgran@unizar.es, §mvg@unizar.es, ¶dario@unizar.es

Lester Kalms† and Diana Göhringer‖
Technische Universität Dresden
— HiPEAC Network of Excellence
e-mail: †lester.kalms@tu-dresden.de, §diana.goehringer@tu-dresden.de

***Abstract*—In Computer Vision, open programming standards such as OpenVX have emerged to bring together portability and acceleration across devices. Unfortunately, achieving both goals on FPGAs remains a challenge because FPGAs still require to adapt the code with proprietary extensions. Exclusively for Xilinx devices, the HiFlipVX open source library partially solves this problem by offering a clean C++ OpenVX API that offers the performance of proprietary extensions without exposing its complexity to programmer.**

While HiFlipVX enables portability within Xilinx devices, portability between FPGA manufacturers remains an open challenge. This work extends the HiFlipVX's capabilities with a twofold goal: i) to support Intel FPGA devices with different memory configurations, and ii) to enable execution on FPGAs as discrete accelerators. To accomplish these goals, the proposed implementation combines two HLS programming models: C++, using Intel's system of tasks that enables to coalesce nodes and reduce control overhead, and OpenCL, which provides efficient compute kernel nodes. On Intel FPGAs, compared with pure OpenCL implementations, the proposed implementation reduces kernel dispatch resources, saving up to 24% of ALUT resources for each kernel in a graph, and improves performance. Gains are 2.6× on average for representative applications, such as Canny edge detector, or Census transform, compared with state-of-the-art frameworks.

***Index Terms*—FPGA, HiFlipVX, HLS, OpenVX, OpenCL.**

I. Introduction

The introduction of High-Level Synthesis (HLS) is attracting software programmers towards FPGAs. HLS tools enable to directly write software applications using high-level languages such as C/C++, OpenCL, and SyCL instead of using hardware description languages. Although the HLS tools abstract the programmers from the hardware, they still require an expert domain knowledge that most programmers still lack.

While HLS helps to improve programmer's productivity, standards and frameworks targeting specific domains aims to provide performance portability. In the field of computer vision, OpenVX is presented as an open, royalty-free standard for cross-platform acceleration [1] where applications can be expressed as graphs to maximize optimization potential because all dependencies are known before the graph is processed.

In the FPGA domain, the acceleration of computer vision primitives is well understood, but the acceleration of OpenVX applications still remains as a challenge. Their efficient implementation requires specific optimizations on primitives and communication. Some HLS programming models address these requirements; e.g., HiFlipVX, an optimized library of OpenVX functions, exploits streaming capabilities and parametrization for Xilinx FPGAs [2]. However, its highly-tuned implementation is neither portable nor efficient on other FPGA platforms such as Intel.

Cross-platform acceleration is one of the main OpenVX features. But, since the required implementation differences between the main FPGA manufactures: Xilinx and Intel are large, to date, proposals for accelerating OpenVX on FPGAs only target one of them. To cope with this issue, the present paper extends HiFlipVX implementation to support Intel FPGA devices with different external memories as DDR4 and HBM. This work details the key changes required to guarantee portability and to keep performance.

The proposed implementation leverages Intel's HLS System of Tasks [3] asynchronous model for parallel execution of OpenVX functions. Therefore, with the required software support, any HiFlipVX graph can now be encapsulated as an OpenCL or a SYCL library. This novel approach to program OpenVX on Intel FPGAs raises traditional OpenCL workflow on discrete devices attached to a host, providing less runtime overhead since the entire graph representation is inside a single kernel.

In summary, the main contributions of this paper are:

- A new portable implementation of HiFlipVX for Intel FPGAs that keeps the compatibility with Xilinx devices. Up to our knowledge, this is the first OpenVX portable implementation between Xilinx and Intel devices.
- Support for Intel FPGAs devices with either DRAM or HBM memories.
- The proposed implementation relies on Intel's System of Tasks which improves the performance on average $2.6\times$ in comparison to OpenCL implementation.

This work was supported by MINECO/AEI/ERDF (EU) (grants TIN2016-76635-C2-1-R and PID2019-105660RB-C21), Aragón Government (T58_20R research group), ERDF 2014-2020 "Construyendo Europa desde Aragón" and HiPEAC Collaboration grant 2019.

The rest of the paper is organized as follows. Section II describes the motivations and HLS flow alternatives for the OpenVX implementation on FPGAs. Section III provides background on the related work. Section IV describes the HiFlipVX library and the included changes to port it to Intel FPGA. Section V presents the methodology. Section VII discuses the results, and Section VIII set out our conclusions.

II. Background and Motivation

HLS programming languages reduce the programming entry barrier of FPGAs. They have favored the flourishing of a new ecosystem of high level toolkits and programming strategies to achieve optimized FPGA pipeline implementations, similarly to what CUDA and OpenCL did to GPUs a decade ago.

One of the most successful HLS approaches in heterogeneous systems is OpenCL, because it unifies the programming language with CPU and GPU devices. However, OpenCL still suffers from a limitation: the programming strategy to achieve an optimal implementation for FPGAs differs from other devices and requires to choose the appropriate OpenCL execution model [4], [5]. Furthermore, code written with only the OpenCL standard does not perform well on FPGAs and requires manufacturer defined extensions.

On the contrary, implementing the OpenVX standard can take advantage of FPGA pipeline-based implementations. In OpenVX, applications use a graph-based programming model where nodes, instances of kernels, contain the function code; and edges represent the data movements [1]. This data flow programming model has two main design alternatives with FPGAs using OpenCL:

- *Standard OpenCL*: each OpenVX node is an OpenCL kernel following the standard OpenCL API, as shown in Fig. 1a. This alternative can be easily portable between manufacturers; the main disadvantage is the lack of guarantees to generate a deep pipeline connecting the function nodes, because the compiler separates the pipeline stages by control operations. 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 thanks to the streaming communication between kernels, as shown in Fig. 1b. In this case, the host should launch every kernel in multiple command queues to get a concurrent execution of 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 dramatically reducing portability [6].

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 [7] 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.

For Xilinx devices, HiFlipVX implements OpenVX using C/C++, enabling a highly parameterizable library. However, to complete an efficient and portable OpenVX specification is necessary to port the library to Intel devices. The differences between C/C++ standards and compiler, such as OpenCL, are not trivial, generating performance differences between manufacturers. Also, FPGA board design differences among families show a heterogeneous FPGA environment, 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 Fig. 1c. With this strategy, OpenVX applications overcome the pipeline depth limitations in Standard OpenCL (Fig. 1a) and reduces the host dependency on OpenCL Channels implementation (Fig. 1a). This property is specially crucial for Intel FPGA devices as Table I shows.

TABLE I. Programming flow alternatives to implement the OpenVX standard.

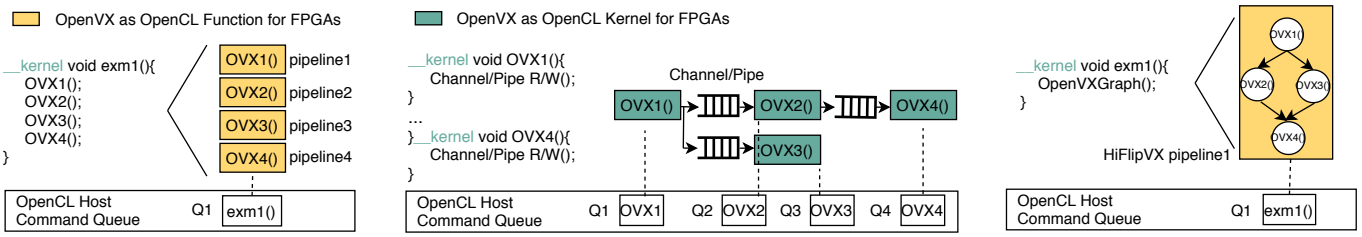
Programming Flow	Manufacturer portable	Deep Pipeline	Host Dependency
Standard OpenCL	✓	✗[a]	LOW
OpenCL Channels [7]	✗	✓	HIGH
HiFlipVX [2]	✗	✓	-
This Work	✓	✓	LOW

[a] Xilinx can enable deep pipelines with streaming pragmas.

III. Related Work

The suitability of FPGAs for accelerating computer vision algorithms is clear, and previous results show high performance and energy efficiency [8]. However, implementation complexity and development time are still FPGA main adoption barriers.

DSL (Domain Specific Language) is one of the proposals for image processing on FPGAs. For example, HeteroHalide [9] uses Halide DSL and an external compiler back-end to support Intel and Xilinx devices. Hipacc [10] developed a DSL to support multiple back-ends from different vendors and devices such as FPGA, GPU, and CPU. Other DSLs such as PoliMage [11] and Pu's [12] have evolved to support Xilinx FPGAs. The two main disadvantages of DSLs are the learning process for the programmer, and the difficulties for extending their functionality.



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

Fig. 1. Programming flow alternatives for OpenVX using HLS for FPGA devices. The yellow boxes show OpenVX functions as OpenCL functions and the green ones the OpenVX functions as kernels. The bottom boxes show host command queues, Qn, that manage the kernels.

More general than DSL is the library approach or standard based libraries; e.g., Xilinx provides the xfOpenCV library, implementing functions from the OpenCV library [13]. Other libraries targeting OpenVX are HiFlipVX and AFFIX for Xilinx and Intel FPGAs, respectively [2], [7], [14].

Along with introducing computer vision standards as OpenVX, another desirable characteristic is the performance portability, but none of the available libraries support both major FPGA vendors: Xilinx and Intel. In fact, transferring codes between vendors is not straightforward, even with HDLs, because each vendor requires a different coding style for maximizing performance, and computer vision applications are not an exception of this problem [6], [8].

The most viable options to provide a general computer vision library seem HiFlipVX and AFFIX. However, AFFIX relies on OpenCL limiting the implementation of OpenVX functions and graphs, in some cases. However, HiFlipVX simplifies the usage of graphs using the standard C++ language, and it is oriented towards portability with explicit management of data types to generate optimized hardware. Furthermore, HiFlipVX was verified in multiple embedded applications for Xilinx [15]–[17].

IV. HiFlipVX

HiFlipVX is an open source HLS FPGA library for image processing applications [2]. It is a C++ based library, 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.

The functions in HiFlipVX can be categorized in pixelwise, filter, analysis, and conversion functions as Fig.2 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 Sobel 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.

The library was designed to be as vendor independent as possible. Since no external libraries are required, it can also run on a CPU. Additionally, the library is extended with pragmas and macros for acceleration on Xilinx FPGAs.

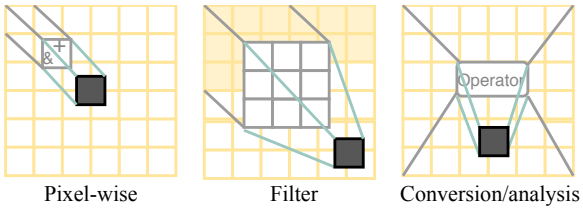


Fig. 2. Image functions categories implemented in HiFlipVX.

These directives are used for pipelining, partitioning arrays and interfacing between functions. The functions of HiFlipVX were used for various applications, such as in a toolchain [15], or an operating system [16]. Akgün et al. show that the use of vectorization increases not only performance but energy efficiency as well [17].

V. Methodology

All experiments have run on two high-end FPGAs: an Intel Stratix 10 GX Development Kit (1SG280LU2F50E2VG) with 2GB of HiLo DDR4 DRAM @1866MHz and an Intel Stratix 10 MX Development Kit (1SM21BHU2F53E2VGS1) with 32x 256MB HBM memory banks @12800MHz. Both boards uses the PCIe Gen3 x8 to connect with the host CPU. Table II summarizes FPGA resources specification.

TABLE II. FPGA Resources for Stratix 10 GX and Stratix 10 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 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 three representative OpenVX graphs, including all the categories of Fig 2, from the Intel OpenVX samples:

- Canny edge detector: Popular multi-stage algorithm for edge detection and suppressing noise.
- Census transform: A common algorithm for correspondence problem used in stereo image processing for disparity calculations [18].
- Auto-contrast: Algorithm to improve contrast in images, adjusting the image intensity.

Finally, these benchmarks are also used to compare with existing state-of-the-art approaches running them in the same FPGA as this work.

VI. OpenVX with HiFlipVX for Intel FPGAs

The OpenVX specification defines: i) a high-level set of abstractions so programmers can easily write computer vision applications, and ii) a runtime that helps optimizing the applications. In most cases, those optimizations are specific for each hardware vendor.

On CPU and GPU devices, the runtime dynamically verifies and processes the graphs on-the-fly, providing more flexibily and opportunities for optimization, but, on the other hand, this processing has to be statically performed before the costly bitstream generation on FPGAs.

Thus, kernel node optimization depends on vendor specific guidelines that are not part of OpenVX standard, yet very helpful for FPGA compilers. In the case of HiFlipVX, we implemented optimization support for programmers through specialized versions for each vendor of its template-based API. So, with minimal changes in the user-facing code, switching the vendor, programmers can benefit from the performance portability promised by OpenVX.

Providing such performance portability requires substantial changes in the original HiFlipVX implementation of three key OpenVX components: execution model, kernels, and edges.

The kernels nodes comprise the compute part of the graphs, while the edges manage the memory layer, which includes keeping data on-chip to save power and offer potential speedups.

1) Execution model: The OpenVX execution model in HiFlipVX for Intel FPGAS synthesizes every graph as a unique kernel executed sequentially in a pipeline way using the system of task which allows asynchronous nodes to create a DAG for Intel FPGAs. Kernel nodes are called with a proprietary Intel API. On the contrary, the Xilinx specialization only needs the `HLS INLINE` pragma.

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 comparison of the HiFlipVX for Xilinx [2] 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 Fig. 3. These results evidence the differences between architectures and HLS tools; e.g., Xilinx ALUTs are capable of self-split to implement two separated logic functions, unlike Intel that has dedicated ALUTs to improve routing time in complex designs [19].

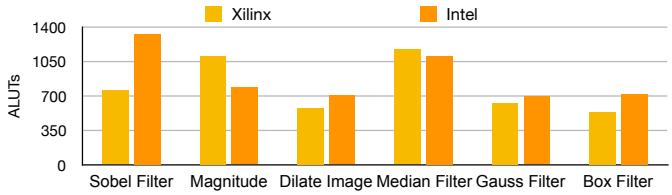


Fig. 3. ALUTs resource comparison between Intel and Xilinx FPGA at 100MHz and vectorization equal to 1, for 6 sample OpenVX functions.

3) Edges: For Xilinx, HiFlipVX implements optimized communications through streaming and uses pass-by-pointers for parameter passing between functions. In general, these choices guarantee an Initiation Interval, II, equal to 1 cycle and low latency for filter-type kernels [2].

The lack of equivalent pragmas for streaming communications in Intel FPGAs and the poor performance of pass-by-pointer parameters can result in components with up to 114 cycles II, because the HLS tool generates a single Avalon Memory-Mapped (MM) Master interface with a single arbiter for all variables [20].

To improve the *pointer* access, function parameters are passed-by-reference, which are more suitable for Intel [21], reducing the II to 1 with a simpler implementation. The two first groups of bars of Fig. 4 show the 114× cycle difference between the pass-by-pointer and pass-by-reference for a 3x3 filter.

Within the RTL generation, depending on function interface, the functions parameters change from *reference* to *stream* to support Intel's System of Tasks, due to concurrency necessities to implement graphs. With system of tasks, nodes run asynchronously and communicate with streams among them. Streams allow a II of 1 and, in practice, resulting in an Avalon streaming interfaces.

Comparing the *streams* with *reference*, streams increase the latency up to 2× because system of task adds hardware to control kernel pipeline for achieving concurrency. Fig. 4 shows the impact on both II and latency of all the interface changes: passing arguments by reference and stream communication among kernels.

With the proper FPGA interfaces validated, another important part of the standard is the OpenVX data objects, which allow the access to memory to users or only connect various nodes within a graph via data references without access to intermediate data [1]. Virtual objects are implemented with *streams* which are limited to access by references, this means the implementation of array of *streams* are not allowed.

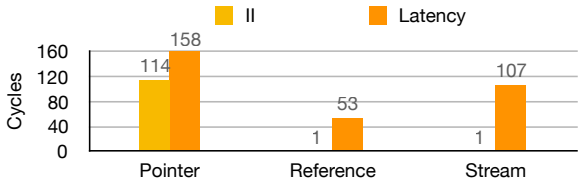


Fig. 4. Latency and Initiation Interval for interface optimizations on edges in a 3x3 filter function (lower is better).

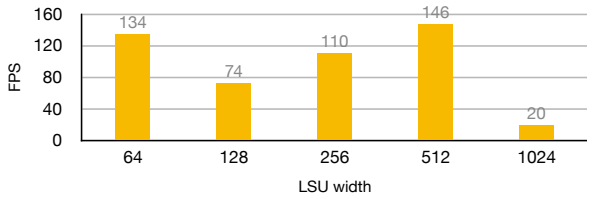


Fig. 5. Efficiency (Frames Per Second) for canny edge with a HD image varying coalescing to read DRAM memory (LSU width), higher is better.

Linking more than two nodes with the same Virtual Objects requires duplication of the number of buses in an FPGA. For this, the stream signals are multiplied with a custom internal kernel: `vxSplit`. It enables to feed concurrently multiple kernels with the same data-stream.

Contrary to Virtual Objects, Images objects allow user access. This creates an opaque reference to an image buffer [1]. In Intel FPGA, the user access data through external ports connected with Avalon MM buses.

In embedded FPGAs, *stream* objects with input and output qualifiers are enough, but discrete devices have external memory as DRAM to increase the memory capacity, and all these external memories require to generate and manage their IP controllers. To manage external DRAM memories with discrete devices, HiFlipVX leverages existing host drivers for OpenCL/SYCL to perform the required transactions. And, internally, those transfers are instantiated with two custom kernels.

Image objects for DRAM connection allow optimizations for contiguous and aligned memory accesses, and modifications of the burst size with the coalescence parameter, accordingly. Furthermore, parameters allow to adjust technologies difference between FPGA boards and maximizes DRAM bandwidth. Fig. 5 plots performance of a Canny edge graph as an example. It shows that increasing the coalescing factor to reach a load unit larger than 512 bits reduces the performance 7×, because the compiler heuristic infers a non-aligned access. If a load unit width smaller than 512 bits, the maximum DRAM burst is underused, except in 64 bits that correspond to dq, which is the DRAM bus size.

Once OpenVX graphs are programmed in C/C++ with HiFlipVX, the next step is to evaluate the target FPGA. For an embedded FPGA, the bitstream should be generated after the RTL generation, and for an Intel discrete FPGA, which needs communication with a host, it should be coupled to a BSP (Board Support Package).

Intel has tools to generate OpenCL and SyCL libraries from

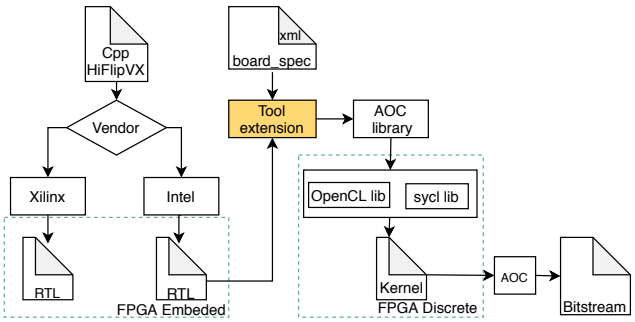


Fig. 6. HiFlipVX programming and compilation flow for Xilinx and Intel FPGAs.

HLS. However, the system of tasks is not supported yet, which is necessary to implement OpenVX graphs. To circumvent this problem, the proposed compilation flow requires an extra tool that takes two inputs: an XML file with the DRAM memory port description, through an Avalon Master interface, and the RTL from HiFlipVX, both of them compatible with the target BSP. The generated output file enables the library generation with Intel standard *aoc* tools. As result, the output libraries are ready to be used in OpenCL/SyCL kernels. Fig. 6 shows the new programming flow to couple the HiFlipVX graph to a heterogeneous system. Please note that the Xilinx paths remain unaffected, and how the RTL from the HiFlipVX library combines with the BSP on the Intel path.

VII. Results

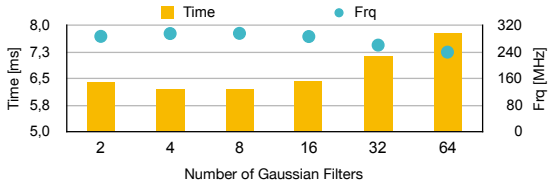
This section analyzes the main characteristics and the performance of HiFlipVX running on Intel FPGA devices. First, it covers the scalability of the implementation, then continues with the evaluation of the library running representative OpenVX graphs, and finally presents the comparison one state-of-the-art proposal.

A. HiFlipVX Scalability Analysis

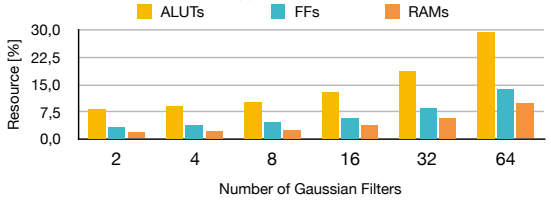
To evaluate how deep pipelines and system of tasks affect graph scalability, a synthetic graph derived from the first stage of the SIFT feature detector is used. This multi-Gaussian graph applies multiple times a Gaussian filter to an image stream [22]. Specifically, the pipeline depth increases by adding Gaussian filtering steps, one after the other.

Fig. 7a shows how execution time and FPGA frequency scale with the number of filter nodes for the multi-Gaussian graph. From 2 to 16 filters, memory access time dominates, flattening 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.

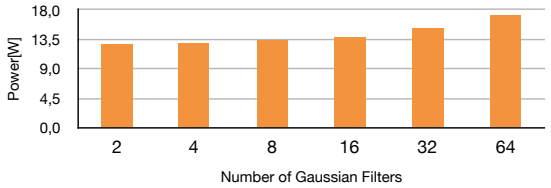
Fig. 7b shows how the resource usage increases linearly, as expected, with a growing rate of 0.33, 0.17, and 0.13 for ALUTs, FFs, and RAMs resources, respectively. With the rise of resources usage, FPGA power presents a similar tendency with a growing rate of 74 mW per additional Gaussian filter



(a) Execution time

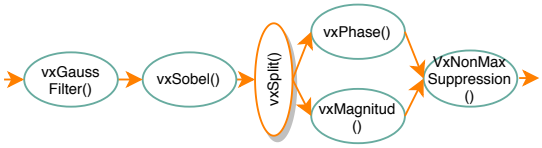


(b) Resource usage

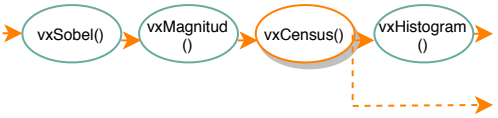


(c) Kernel Power

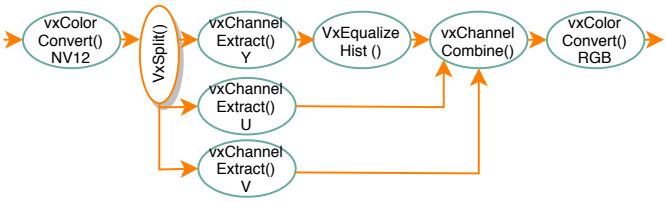
Fig. 7. Impact of node scalability on a) execution time; b)frequency and resource utilization; and c) power consumption for the Multi-Gaussian synthetic benchmark using the Stratix 10 GX.



(a) Canny Edge



(b) Census Transform



(c) Autocontrast

Fig. 8. OpenVX application graph diagrams. a) Canny edge detector, b)Census transform and d) autocontrast image

stage, as Fig. 7c shows. In summary, HiFlipVX with the system of task scales well without adding any extra overhead increasing the graph complexity.

B. *OpenVX Applications*

This section analyzes resource usage (per-kernel) and execution time of three representative applications: Canny edge, Census transform, and Autocontrast, as depicted in Fig. 8.

The Canny edge detector, Fig. 8a, is a multi-node graph algorithm that extracts the edge information from images. In HiFlipVX, its implementation consists of 8 nodes, including the extensions which enables DRAM as User objects and Virtual extension to support more than two connected nodes (vxSplit), as described in Sec. VI. Table III shows the estimated resource usage from the i++ report for all Canny edge nodes. 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.

Census transform, like Canny, uses filter functions, but since Census is not part of the OpenVX standard, we added it as a kernel extension. Table IV shows the estimated resource usage for Census transform functions, while Table III shows the usage for shared functions between Census and, above explained, Canny.

The last evaluated graph is Autocontrast, requiring to extend HiFlipVX to support the graph from Fig. 8c with two new kernels for color conversions: NV12 to RGB and RGB to NV12, and EqualizeHist.

TABLE III. 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
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[a]	165	133	1	0
vxNonMaxSuppression	4605	5842	18	0
Image Object	4177	11949	18	0

[a] Reduce orientation for 4 gradient directions.

Autocontrast requires more RAM resources than other graphs because intensity channel (Y) is stored in RAM memory until histogram is calculated. This strategy avoids stalls in streams at expense of higher resource usage and dependence on the input image size; e.g., if the image size changes from HD to 4k, RAM usage increases by 4×. To save resources, the coalescence of DRAM access is reduced with a LSU width of 64. The Table V shows the resource for each function in the Autocontrast graph.

The execution time is evaluated on both FPGAs devices: Stratix 10 GX and Stratix 10 MX, from here on, S10GX and S10MX, respectively. The main differences between the implementations for both FPGA are in the global memory. While the S10GX has one DRAM bank with a data port width of 512 bits, the S10MX has a heterogeneous memory composed by 32 DRAM banks and a data port width of 256 bits. To switch between both, a programmer has only to change the DRAM port declaration.

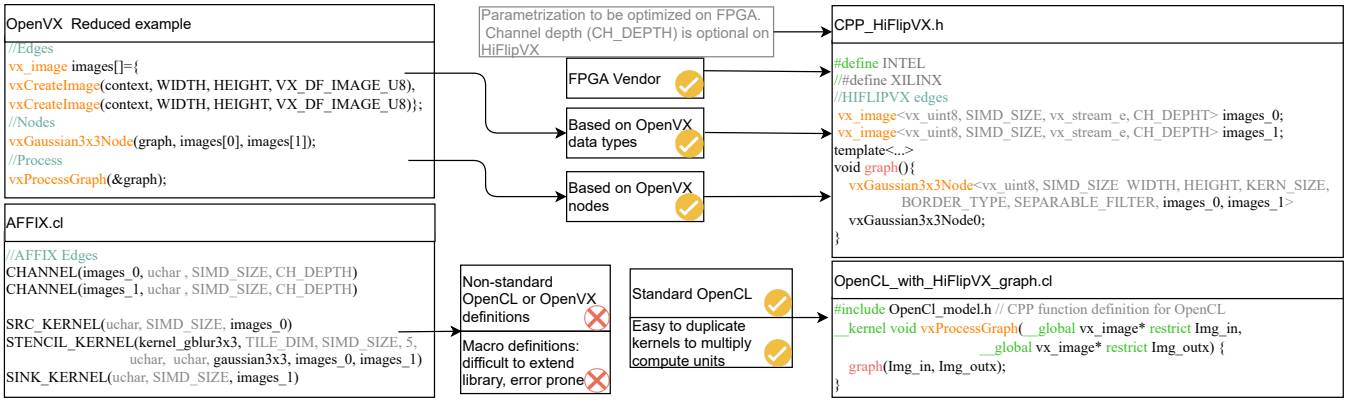


Fig. 9. Code comparison between a reduce version of OpenVX, AFFIX, and HiFlipVX. OpenVX definitions and FPGA optimization parameters are marked in orange and grey, respectively.

TABLE IV. Estimated resource usage for each function in Census graph using HiFlipVX, with a 4k Image and vectorization factor of 8.

Function	ALUTs	FFs	RAMs	DSP
vxCensus	179	208	0	0
vxHistogram	960	16924	2	0

The execution time of the three graphs is shown in Table VI. It should be noted that Canny edge and Census transform have similar execution time. Although Canny edge has to calculate one additional filter, the pipeline architecture hides extra operations. On the S10MX the kernel frequency is lower than S10GX, and, hence, its execution time is slower in spite of the use of one memory bank per variable. A maximum 7% of execution time difference between FPGAs as well evidence architecture and BSP differences, where S10MX has an experimental support and the synthesis effort is arduous due the amount of HBM banks.

All the evaluated applications are compute bound, e.g. in Canny and Census the maximum memory bandwidth for S10GX and S10MX it is 2.4GBs, so the extra HBM bandwidth does not provide any advantage.

C. Comparison with Existing Approaches

One of the most relevant proposals that implement OpenVX graphs is AFFIX [7]. It is inspired in OpenVX standard and

TABLE V. Estimated resource usage for each OpenVX function in Autocontrast graph using HiFlipVX, with an HD image and vectorization factor of 1.

Function	ALUTs	FFs	RAMs	DSP
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
Image Object	1102	3605	18	0

TABLE VI. Execution time of HiFlipVX 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]	Time[ms]	Frq.[MHz]
Canny edge	6.8	310	7.3	293
Census	6.8	331	6.9	326
Autocontrast	23.1	301	23.9	294

implemented using OpenCL channels, as shown in Fig. 1b. The use of OpenCL limits the programmability of AFFIX. Comparing the graph codes from Fig. 9, 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, this 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. Finally, 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.

For performance comparison, AFFIX had to be modified because it uses the Intel host pipe extension to directly communicate between the host and FPGA kernels, instead of going through the on-board DRAM. Host pipes reduce latency overhead in communication but have two main limitations: Its limited support on a few Arria 10 GX development kits [23], and 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 from benchmark, as shown in Fig. 8b 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.

As result, HiFlipVX reaches a speed-up of 3.4× and 3.6× on the graphs using filters as Table VII shows. Autocontrast is an implementation with large design differences between the libraries. Although both approaches suffer from large

TABLE VII. Comparison between HiFlipVX and AFFIX on a Intel Stratix 10 GX and Intel Stratix 10 MX usign a 4k image

OpenVX Application	Stratix 10 GX speed-up	Stratix 10 MX speed-up
Canny edge	3.2	3.6
Census	3.6	3.4
Autocontrast	0.8	0.8

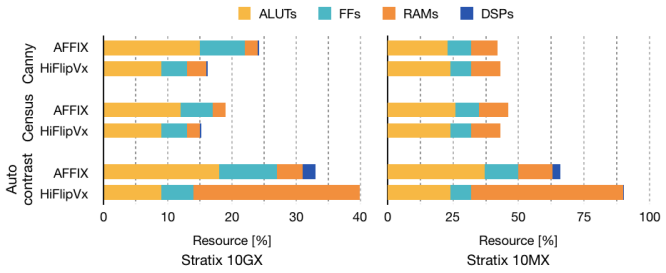


Fig. 10. Resource usage per logic unit relative the total units on Stratix 10 GX and Stratix 10 MX for AFFIX and HiFlipVX implementations.

pipeline stalls in the sequential processing, the eagerness for RAMs resources of HiFlipVX penalizes circuit frequency, and consequently, performance is degraded by 20%.

Regarding area usage, the Fig. 10 shows the resource consumption of AFFIX and HiFlipVX. One of the main differences is that OpenCL generates "kernel dispatch logic" for each OpenVX kernel to communicate the kernel with the host on AFFIX. This extra logic increases the resource usage per node by 1463 ALUTs and 1467 FFs.

In HiFlipVX, the kernel nodes are compressed in a single kernel that shares the same dispatch logic and saves 4 to 24% of area 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.

Differences in ALUT and FFs in Autocontrast are bigger than that of the other evaluated applications due to the increment of the number of kernels in a graph. With the use of HiFlipVX, Autocontrast saves a 1.4 and 0.9% of total ALUTs and FFs resources, respectively. As expected, the amount of RAM resources in HiFlipVX is 4× bigger, and 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. Finally, in this work the color conversion is implemented with a 8-bit approximation [24] that does not require DSPs for float operations.

At last, we compare the traditional OpenCL model, depicted in Fig. 1a, used by the Chai Benchmark [25] for Canny edge. For the comparison, Chai's code was compiled for a single frame running exclusively on the FPGA. Chai's communication between nodes through external memory and limited deep of pipeline per function makes HiFlipVX 9× faster than Chai for Canny edge.

VIII. Conclusions

This paper presents a cross-platform OpenVX library for FPGAs built by extending the former HiFlipVX implementation, an OpenVX library designed for Xilinx devices. This new version supports Intel FPGAs with two different memory subsystems: DDR4 and HBM. To ensure performance portability across Xilinx and Intel devices, the user-facing OpenVX API has not be changed, but the new implementation exploits the novel Intel's System of Tasks to coalescing OpenVX nodes into accelerated graphs on Intel FPGAs.

As a result, the new implementation saves around 1.5% of ALUTs usage per node in graphs versus the standard approach with a kernel per node in OpenCL. Additionally, the paper show how the library provides performance portability, this is less than 7% difference in execution time, on two different FPGA models, one with DDR4 and other with HBM memories.

Compared with the state-of-art, HiFlipVX performs up to 3.6 and 9.6 × faster than AFFIX and Chai, respectively. And, even more important, HiFlipVX does not modify the OpenVX standard, ensuring better programmability and performance portability properties.

References

[1] R. Giduthuri and K. Pulli, "OpenVX: A Framework for Accelerating Computer Vision," in *SIGGRAPH ASIA 2016 Courses*, 2016, pp. 14:1–14:50.

[2] L. Kalms, A. Podlubne, and D. Göhringer, "Hiflipvx: An open source high-level synthesis fpga library for image processing," in *Applied Reconfigurable Computing*, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, Eds. Springer, 2019, pp. 149–164.

[3] Intel, "Intel® High Level Synthesis Compiler Pro Edition 19.4," 2020.

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

[5] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl," in *FPGA '18*, 2018.

[6] N. Voss, T. Becker, S. Tilbury, G. Gaydadjiev, O. Mencer, A. M. Nestorov, E. Reggiani, and W. Luk, "Performance portable fpga design," in *FPGA '20*, 2020.

[7] S. Taheri, P. Behnam, E. Bozorgzadeh, A. Veidenbaum, and A. Nicolau, "AFFIX: Automatic acceleration framework for FPGA implementation of OpenVX vision algorithms," in *FPGA'19*, 2019, pp. 252–261.

[8] A. HajiRassouliha, A. J. Taberner, M. P. Nash, and P. M. Nielsen, "Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms," *Signal Processing: Image Communication*, vol. 68, no. June, pp. 101–119, 2018.

[9] J. Li, Y. Chi, and J. Cong, "HeteroHalide: From image processing DSL to efficient FPGA acceleration," *FPGA 2020 - 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 51–57, 2020.

[10] O. Reiche, M. A. Ozkan, R. Membarth, J. Teich, and F. Hannig, "Generating FPGA-based image processing accelerators with Hipacc: (Invited paper)," *ICCAD*, 2017.

[11] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A DSL Compiler for Accelerating Image Processing Pipelines on FPGAS," *PACT*, 2016.

[12] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 3, pp. 1–25, 2017.

[13] Xilinx, "Xilinx OpenCV User Guide," 2019.

[14] S. Taheri, J. Heo, P. Behnam, J. Chen, A. Veidenbaum, and A. Nicolau, "Acceleration framework for fpga implementation of openvx graph pipelines," in *FCCM*, Apr 2018.

[15] A. Sadek, A. Muddukrishna, L. Kalms, A. Djupdal, A. Podlubne, A. Paolillo, D. Göhringer, and M. Jahre, "Supporting utilities for heterogeneous embedded image processing platforms (sthem): An overview," in *ARC*, 2018.
[16] A. Podlubne, J. Haase, L. Kalms, G. Akgün, M. Ali, H. Ulhasan Khar, A. Kamal, and D. Göhringer, "Low power image processing applications on fpgas using dynamic voltage scaling and partial reconfiguration," in *DASIP*, 2018.
[17] G. Akgün, L. Kalms, and D. Göhringer, "Resource efficient dynamic voltage and frequency scaling on xilinx fpgas," in *ARC*, 2020.
[18] R. Zabih and J. Woodfill, "Non-parametric local transforms for computing visual correspondence," in *ECCV '94*. Springer, 1994.
[19] HardwareBee, "Xilinx vs. Intel High-End FPGA Series Comparison," 2020. [Online]. Available: https://hardwarebee.com/xilinx-vs-intel-high-end-fpga-series-comparison/
[20] M. A. Dávila-Guzmán, R. Gran Tejero, M. Villarroya-Gaudó, and D. Suárez Gracia, "Analytical model of memory-bound applications compiled with high level synthesis," in *FCCM*, 2020.
[21] Intel, "Intel® High Level Synthesis Compiler Pro Edition 19.4, Best Practice Guide," 2020.
[22] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, p. 91–110, Nov. 2004.
[23] Intel, "Intel FPGA SDK for OpenCL Pro Edition: Programming Guide 19.4," 2020.
[24] Microsoft, "Recommended 8-Bit YUV Formats for Video Rendering," 2018.
[25] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, and W.-m. Hwu, "Analysis and modeling of collaborative execution strategies for heterogeneous cpu-fpga architectures," in *ICPE '19*, 2019.