Acceleration Framework for FPGA Implementation of OpenVX Graph Pipelines

Sajjad Taheri, Jin Heo, Payman Behnam, Alexander Veidenbaum and Alexandru Nicolau

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

sajjadt@uci.edu

CECS Technical Report 18-01
Feb 25, 2018

Acceleration Framework for FPGA Implementation of OpenVX Graph Pipelines

Sajjad Taheri, Alexander Veidenbaum, Alexandru Nicolau
Department of Computer Science
University of California, Irvine
Irvine, CA
{*sajjadt, alexv, anicolau*}*@uci.edu*

Jin Heo
College of Information and
Computer Engineering
Ajou University
Suwon, Korea
jin993@ajou.ac.kr

Payman Behnam
School of computing
University of Utah
Salt Lake City, UT
payman.behnam@cs.utah.edu

***Abstract*—Computer vision processing is computationally expensive and several acceleration solutions have been proposed. Among them, FPGAs offer a promising direction. Vision application are typically written in languages such as C/C++ and they are often difficult to compile into an efficient FPGA implementation. OpenVX is a set of basic, widely used vision kernels. Vision pipelines can be defined as graphs of such kernels, all in C++. Accelerator vendors may support OpenVX kernel implementations, as in nVidia's VisionWorks. This work presents a framework for synthesizing an OpenVX graph-level specification of a vision pipeline into an optimized FPGA implementation using high level synthesis. Three vision pipelines are developed using the framework. Preliminary results show that their performance is higher than either a multi-core CPU or an OpenCL-based FPGA implementation. The proposed framework extracts sub-graphs and compile them into FPGA pipelines, which is the main reason for the performance and resource usage improvement compared to the OpenCL implementation.**

***Keywords*-FPGA; Computer Vision; OpenVX;**

I. Introduction

Computer vision algorithms are widely used in fields like computational photography, medical imaging, autonomous driving, virtual and augmented reality and astronomy [1]. These applications process a large amount of visual data, often in real-time, which has high computation, power, and memory bandwidth requirements. Many vision algorithms have been or can be specified as computational pipelines with massive data parallelism [2]. Prior work has proposed providing a framework for image processing pipelines to deal with these problems and take advantage of intrinsic parallelism: PolyMage [2] and Halide [3] on GPUs, [4] and Rigel [5] on FPGAs. Many of these frameworks provide a Domain Specific Language (DSL) that can be translated into specific architecture by modifying compiler, scheduling and binding phases during high level synthesis.

These frameworks may require manual optimization to exploit locality and parallelism, which requires a lot of expertise [6]. In addition, popular libraries, such as OpenCV[1], only provide efficient software implementations for a limited set of specific architectures. Besides, the user is expected to know at least one programming language to describe the application in the high level. For example, Halide can be used to generate image processing pipelines from a custom DSL. It is however limited to single rate kernels and cannot support more advanced kernels. In addition, it leaves finding the proper schedule which needs a lot of time, effort and expertise from the user. PolyMage uses a polyhedral compiler to optimize computer vision DSL applications using tiling, kernel fusion and memory allocation targeting parallel processors, but the generated code is not optimized for FPGAs.

OpenVX [7] is an open standard for cross platform acceleration of computer vision applications. It was created to address the challenge of implementing efficient, portable and easy to use vision processing algorithms by separating application specification and implantation. It offers a set of basic, widely used vision kernels that accelerator vendors are supposed to provide. One can implement an application by simply connecting OpenVX kernels in a directed acyclic graph as shown in Fig.2.

This paper presents a framework for turning a high-level OpenVX specification into an efficient FPGA implementation. It contains novel optimizations and is divided into Verification, Analysis, and Acceleration phases shown in Fig.1. It is described in more detail next.

II. FPGA Acceleration Framework

Table I lists all OpenVX 1.1[1] vision kernels categorized by their pixel access patterns. In addition, OpenVX allows the definition of user kernels. This classification allows one to define a specific approach for each class.

Fig.1 depicts the proposed framework for FPGA implementation of OpenVX-based graph pipelines. A user provides an application graph using OpenVX kernels as nodes. The framework has three main phases: Verification, Analysis and Acceleration.

A. Verification

OpenVX requires checking whether an input graph is valid. For instance, OpenVX forbids cycles in a graph and

[1]OpenVX 1.2 was released last year when this work was already in progress.

Category	Pixel Access Pattern	VX Kernels
Point-wise	f(x,y) = g(x,y)	thresholding, absdiff, accumulate, accsq, accw, add, pixel-wise-mul, bitwise, channel-combine, channel-extract, color-convert, convert-bit-depth, phase, magnitude, table lookup
Fixed-rate Stencil	$f(x,y) = \sum_{i=-k}^{i=k} \sum_{j=-k}^{j=k} g(x+i, y+j)$	box filter, sobel, non-max suprema, conv, erode, dilate, gaussian blur, non-linear-filter, integral, median-filter
Multi-rate Stencil	$f(x,y) = \sum_{i=-k}^{i=k} \sum_{j=-k}^{j=k} g(Nx+i, Ny+j)$	down-sample, scale-image
Statistical	$F = \sum_{i=0}^{i=Width} \sum_{j=0}^{j=Height} g(i,j)$	histogram, mean, standard-dev
Geometric	$f(x,y) = g(h(x,y), h'(x,y))$	remap, warp-affine, warp-perspective
Non-primitive	N/A	equalize-histogram, fast-corners, harris-corners, gaussian-image-pyramid, laplacian-image-pyramid, optical-flow-pyramids

Table I: OpenVX 1.1 Kernel Categorization and Definition

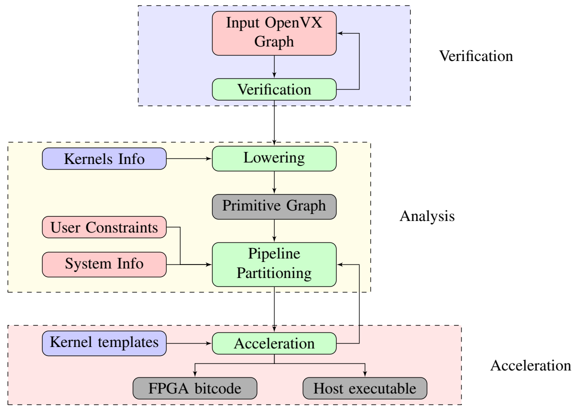

Figure 1: FPGA Acceleration framework for OpenVX

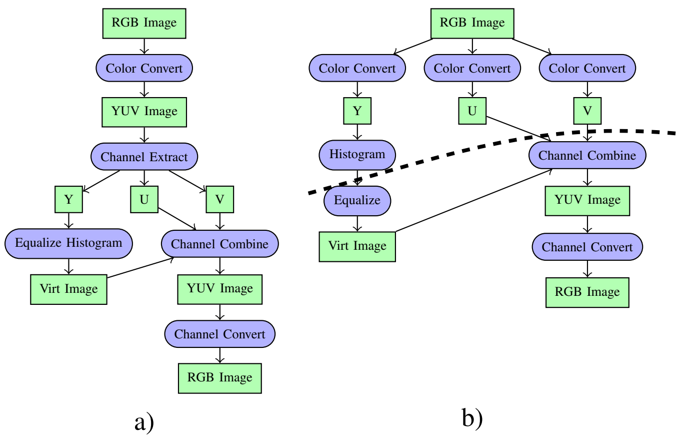

Figure 2: AutomaticContrast Graph: a) user graph b) primitive graph and its partitioning

enforces valid parameters used for kernels and images.

B. Analysis

A verified graph is lowered by replacing non-primitive kernels (if any) with simpler kernels. The framework looks for sub-graphs in the input graph to be off-loaded on an FPGA. It might not be possible to have a complex kernel in a pipeline, so graph partitioning is used. For example, the 'EqulizeHistogram' kernel in the figure is a complex kernel. It consists of Histogram followed by Equalize over full image. One cannot start Equalize until Histogram is calculated for the whole image. The graph is therefore cut into two sub-graphs, which form two pipelines that are invoked on the FPGA sequentially. As Figure 2 shows, 'EqualizeHistogram' kernel is lowered into two separate Histogram and Equalize kernels.

Pipelines are formed for graphs consisting of point-wise and both fixed-rate and multi-rate stencil kernels. Statistical kernels can be part of pipelines, but they must not be followed by any other kernel. The Pipeline Partitioning module divides the Primitive Graph into sub-graphs. Generated pipelines can operate on image tiles (sub images) and can be replicated work on different tiles simultaneously. Multiple sub-graph cuts may share kernels and the framework attempts to reuse them and reduce FPGA resource usage. For instance, in the Auto-contrast example, DSPs that are used in two color conversion kernels (i.e. rgb2yuv and yuv2rgb) to perform floating point arithmetic are shared. The kernels are implemented on FPGA with support of buffering hand shake, which is helpful for Multi-rate Stencil kernels.

C. Acceleration

Parameterized C++ functions to model OpenVX kernel architectures were developed which are verbose enough to guide HLS tools to generate efficient FPGA hardware. Figure 3 shows a template architecture that is flexible enough to represent different types of OpenVX kernels. For example, input data type, buffer sizes and types, degree of parallelism in computation and memory access can be adjusted at the source level. Based on the *Pipeline Partitioning* step, the kernel templates are specialized with appropriate values. These are then fed into Intel i++ HLS tool to generate QSys IP components. QSys IP components are implemented in Verilog and use Avalon bus interfaces to connect to on-chip memories, DMAs and other kernels. TCL scripting is used to form the final design by combining the generated vision kernels and the rest of the system. The final design is synthesized using Intel Quartus software. Acceleration step will start by taking tge best possible pipline partitioning from Analysis step. If the synthesis fails, it comes back to Pipline Partitioning step to take another possible partitioning of the graph.

The resulting accelerator architecture incorporates a standard PCI interface to the host, main memory interface, on chip buffers, DMAs, a central controller unit, and a synthesized OpenVX graph based on data streaming.

It is worth mentioning the i++ compiler, as a one of the latest and strongest compilers, offers numerous options to precisely describe the hardware in C++. For instance, it allows one to take advantage of native FPGA DSP blocks and embedded RAMs by using predefined pragmas. In addition, usage of on-chip memory resources with required attributes (e.g. number of banks and ports) can be enforced. please note that that our approach can be adopted to support other modern HLS tools.

In the proposed framework, time-space trade-offs are possible by adjusting the parallelism level. We consider several levels of parallelism

- Pixel-wise: FPGA resources are consumed to perform concurrent operations on super pixels (i.e. groups of adjacent pixels). This is implemented by having kernels that have wider input and output ports which are processed simultaneously. The level of parallelization is decided at compile time during C++ code generation and can no longer be adjusted at run time.
- Tile-wise: Many image processing operations allow partitioning of an input image into smaller, independent tiles that can be processed independently in parallel. Note that, some kernels, such as those from stencil category need the tile boundary pixels to precisely compute results. This incurs extra computations to the system. However, it is negligible in comparison to the gain obtained from Tile-wise parallelism.
- Graph-wise: parallelism can be extended beyond a kernel by considering a graph of kernels. Multiple instances of kernels withing graph (or the whole graph) can be executed in parallel. For example, multiple instances of kernels may process independent images.

  The Tile-wise and Graph-wise parallelism is applied after generation of Verilog code for kernels. An image is automatically tiled if tiling is supported by the kernel. Tile size is provided by the user.

III. Experimental Evaluation

This section describes the experimental setup used and presents the results of applying the proposed framework. Three different approaches to accelerate OpenVX graphs are implemented and compared: using a multi-core CPU, using an FPGA and Intel OpenCL flow and using an FPGA and the proposed framework.

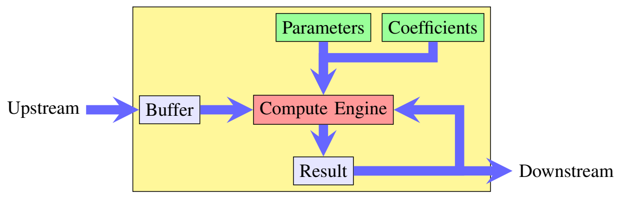

Figure 3: Template Kernel Architecture

The CPU is Intel Core i7-4770 processor with 8 logical threads and the AVX2 instruction set. The FPGA is Intel Arria10 with 1,150K logic units, 1,500 DSPs connected to 2GB DDR4 local memory and Gen.3 PCIe interface with 8 lanes. The applications formed by OpenVX vision graphs are AutoContrast, CensusTransform, and SIFT Keypoints.

AutoContrast automatically adjusts the brightness of a given color image by evenly distributing the brightness. **CensusTransform** performs a census transform operation on magnitude of image gradients and reports the histogram of output image. **SIFT Keypoints** searches for interesting points in multiple scales of the input image which are created using a Gaussian image pyramid.

The FPGA implementation using Intel OpenCL SDK does not automatically support pipeline kernels and run one kernel at a time. They communicate through main memory. However, kernel loops are unrolled to support pixel-level parallelism up to 16 levels. Although, direct kernel communication is available since last year in OpenCL 2.2, e.g. using "pipes", but it requires expertise and a lot of changes to the algorithm and program source code. It was not available when this project started. Intel also provides a non-standard OpenCL extension called "channels" that could be used to directly connect kernels, but it was not used in this paper.

Table II shows the characteristic and the performance of the three applications when they work at the same frequency. The CPU implementation used multi-threading and SIMD instructions. The FPGA implementations performed the whole computation on the FPGA, with the CPU host program just setting up the computation. Overall, the proposed framework produced the hardware that achieves the highest performance. The speedup of the CPU implementation ranges from approx. 3.5x to 32x. The speedup over the OpenCL flow is lower, from approx. 2.3 to 8. Both FPGA implementations used the same pixel-level parallelism.

Table III shows maximum FPGA frequency and resource usage for both OpenCL (the left number) and the proposed framework (the right number) implementations. Compared to the OpenCL implementation, our framework uses a little bit more on-chip memory, but it reduces the logic cells and dsp blocks usage significantly for AutoContrast and CensusTransform benchmarks. This is mostly due to optimizations in the *Analysis* step. They allow the color conversion kernel to be reused in AutoContrast, while for CensusTransform

				Execution Time (ms)	
Benchmark	# OpenVX Kernels	Input Image Size	CPU (8 Cores + Vec)	OpenCL-FPGA	Proposed Framework
AutomaticContrast	5	6592x5120x3	205	143	62
CensusTransform	5	6592x5120x3	192	120	45
SIFTKeypoints	61	6592x5120x1	864	208	25

Table II: Application information and performance results

Benchmark	FMax(MHz)	Logic(%)	BRAM(%)	DSP(%)
AutomaticContrast	(192, 330)	(58, 41)	(15, 18)	(64, 32)
CensusTransform	(135, 335)	(31, 18)	(12, 16)	(40, 20)
SIFTKeypoints	(190, 351)	(52, 55)	(19, 23)	(22, 22)

Table III: Resource usage and maximum operational clock frequency for OpenCL implementation and our framework

they allow removal of color conversion to U and V channels, which were present in OpenCL implementation. Moreover, it cam work with much higher frequency.

The main reason for lower performance of OpenCL implementation is lack of kernel pipelining. In the off-chip DRAM module on the FPGA board is OpenCL's device memory and all the communication between kernels goes through it. It becomes a major bottleneck as each kernel reads/writes the image to it. In our proposed framework, pipelines with multiple concurrent kernels are formed that communicate through fast on-chip FIFOs. In addition, in our framework, there is direct communication between design on the FPGA and a host program. Accordingly, transferred data are not stored in the DRAM before/after computation, while the OpenCL implementation stores data in DRAM before/after computation. SIFTKeypoint has less communication between the host and the FPGA and the performance increases significantly.

Larger tile size increases the on-chip memory usage for stencil kernels as they are implemented using the sliding window technique. It also increases the latency of each kernel. On the other hand, the smaller the tile, the more re-computation is needed to calculate output tile boundaries. We used a fixed size of a tile, but it could be optimized.

The scalability of the proposed framework can be seen in Fig. 4. It shows resource usage for different level of parallelism. The static region of the design, which includes the PCIe IP, DDR controller, and run-time management consumes 8% of logic resources and 14% of BRAM resources on the Arria10. The remaining resource utilization grows linearly with parallelism. However, in some cases it becomes quite high, e.g. over 50% of the DSPs are used. This is due to usage of floating point arithmetic in some kernels such as color conversion which is specified by OpenVX.

IV. Future Work

The framework does not currently support some of the OpenVX 1.1 kernels, i.e. geometric, and some of the kernels of OpenVX 1.2. This will be addressed in the future work. In addition, some annotations were used as user constraints

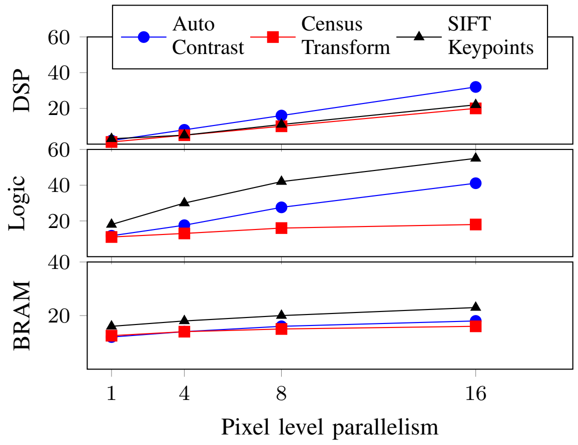

Figure 4: Arria10 resource usage vs level of pixel parallelism

to guide optimization and system generation (e.g. tile size) that in the future will be automatically extracted through the implementation of cost analysis. OpenCL implementation will be optimized to use pipes. Finally, power consumption of different implementations will be compared.

V. Conclusion

Computer vision processing is intrinsically parallel and thus ideal for FPGA processing. However, it is difficult to produce efficient FPGA designs for complex vision pipelines described at the high level, automatically. This paper presented a framework to efficiently map OpenVX vision graphs to FPGA pipelines, which simplifies such development. Higher performance and less resource usage were achieved for three non-trivial OpenVX graphs comparing with other possible implementations.

References

[1] G. Bradski, "The opencv library." *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, vol. 25, no. 11, pp. 120–123, 2000.

[2] R. T. Mullapudi *et al.*, "Polymage: Automatic optimization for image processing pipelines," in *ACM SIGPLAN Notices*, vol. 50, no. 4, 2015, pp. 429–443.

[3] R. Kelley *et al.*, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[4] P. S. Rawat *et al.*, “Resource conscious reuse-driven tiling for gpus,” in *IEEE International Conference on Parallel Architecture and Compilation Techniques*, 2016, pp. 99–111.

[5] J. Hegarty *et al.*, “Rigel: Flexible multi-rate image processing hardware,” *ACM Transactions on Graphics*, vol. 35, no. 4, p. 85, 2016.

[6] F. Grull *et al.*, “Accelerating image analysis for localization microscopy with fpgas,” in *International Conference on Field Programmable Logic and Applications*, 2011, pp. 1–5.

[7] E. Rainey *et al.*, “Addressing system-level optimization with openvx graphs,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 644–649.