An Accelerated OpenVX Overlay for Pure Software Programmers

Hossein Omidian
Xilinx Inc.
San Jose, CA, United States
hosseino@xilinx.com

Nick Ivanov
VectorBlox Computing, Inc.
Vancouver. BC, Canada
nick@vectorblox.com

Guy G.F. Lemieux
University of British Columbia
Vancouver, BC, Canada
lemieux@ece.ubc.ca

***Abstract*—This paper presents an FPGA-based overlay for accelerating computer vision applications written in OpenVX. A software programmer simply writes an application using the standard OpenVX API. The OpenVX overlay consists of an architecture and a runtime system that runs any OpenVX application, unmodified, in an accelerated manner on an FPGA. The architecture uses a Soft Vector Processor (SVP) for general acceleration, and a library of Vector Custom Instructions (VCIs) to further accelerate specific OpenVX kernels in the FPGA fabric. The VCIs are predesigned in advance by a skilled FPGA designer. The runtime system analyzes the OpenVX computational graph and selects some kernel nodes to be executed by VCIs, with the remaining kernel nodes to be executed by the SVP. In making the selection, the runtime system uses an optimization algorithm and relies upon bitstream relocation and bitstream merging to fit multiple VCIs into a single, fixed-size Partially Reconfigurable Region (PRR). The optimization algorithm must select the VCIs that satisfy the area constraint of the PRR and give the best overall application acceleration. For example, on a Canny-blur OpenVX application, an 8-lane SVP achieves speedup of 5.3 over the hard ARM Cortex-A9. Selecting some nodes as VCIs provides another 3.5 times speedup, for an overall speedup of 18.5. The overlay enables OpenVX programmers with no FPGA design knowledge to accelerate their application.**

***Keywords*-Computer Vision; FPGAs; Soft Vector Processor; Partial Reconfiguration; OpenVX; FPGA Overlay**

I. INTRODUCTION

Software applications written in OpenVX are portable, optimized and power-efficient across many accelerator back-end targets. Portability is achieved by recompiling the application together with a target-optimized OpenVX runtime system. By supporting operations on image tiles, OpenVX adds a memory management layer missing from OpenCV, the de facto computer vision framework. This allows OpenVX to save significant power and offer speedups by keeping data on-chip for as many operations as possible.

Computer Vision (CV) is an important class of computationally demanding applications that are ideally suited for FPGA acceleration. The general approach for accelerating algorithms on FPGAs is to use high-level synthesis tools that compile C and OpenCL into pipelined datapaths on the FPGA fabric. However, these tools are only usable by someone with hardware experience and knowledge about FPGA design. It is unreasonable to expect software programmers to be able to use these tools to accelerate an OpenVX application without extensive training and an opportunity to develop experience.

To help reduce the barrier to entry of accelerating computer vision applications, Xilinx has has created an OpenCV library, called xfOpenCV, as part of their reVISION stack. This library uses Vivado HLS to accelerate 45 vision kernels that are also in OpenVX. Unfortunately, performance for most functions is limited to only 1 or 8 pixels in parallel. Hence, fine-grained scaling of the size or performance of an accelerated application is not possible. In addition, users need experience with hardware design and the FPGA tools.

Several approaches for synthesizing image processing or computer vision applications onto FPGAs have been published, including domain-specific languages such as Darkroom [1], Rigel [2] and Halide [3], as well as HLS-based OpenVX toolkits including Intel's OpenVINO [4], a research platform from UC Irvine [5], and JANUS [6], [7]. Among the FPGA-based OpenVX toolkits, JANUS is the only one that uses an optimization framework for design space exploration to automatically meet an area budget (with maximum throughput) or a throughput target (with minimum area). All of these approaches result in a high-performance, custom-generated pipelined solution. However, the user must also learn the FPGA tools (eg, to achieve timing closure).

This paper presents an OpenVX overlay that allows pure software programmers, with no hardware knowledge or FPGA tool experience, to develop complete FPGA-accelerated OpenVX applications. Given an area target, the OpenVX runtime makes fine-grained space/time tradeoffs to achieve the highest possible throughput. The overlay starts with either a hard ARM core, or a slower soft core to execute the OpenVX runtime environment. For acceleration, it uses the VectorBlox MXP Soft Vector Processor (SVP) [8] and Vector Custom Instructions (VCIs) [9].

Unfortunately, the performance and area of the SVP falls short of what can be achieved with custom logic. For example, Table I shows the throughput of the $vxMagnitude$ OpenVX kernel on different platforms. The vectorized software implementation running on a SVP with four vector

Table I: $vxMagnitude$ kernel performance

Platform	Throughput (megapixel/s)	Speedup vs. A9
Cortex-A9 (667MHz)	10.3	1.0
SVP V4 (100MHz / 13k LUTs)	65.5	6.3
SVP V8 (100MHz / 22.5k LUTs)	129.	12.5
Custom logic (100MHz / 3k LUTs)	1180.	114

lanes (V4) uses about 13,000 LUTs and achieves 6.3 times higher throughput than the ARM Cortex-A9 despite its lower clock speed. Increasing the number of vector lanes to 8 doubles the speedup to 12.5 and uses 23,000 LUTs. However, a custom logic implementation that is area-constrained to just 3,000 LUTs is 114 times faster than the ARM Cortex-A9.

While it may appear that custom logic is best, there are two main limitations. First, and most importantly, it requires knowledge of the FPGA tools, making it inaccessible to software programmers. Second, unlike the SVP, it is completely inflexible and cannot be 'reprogrammed' to do other tasks.

Thus, to accelerate OpenVX compute graphs in a way that requires no FPGA or hardware skills, we have designed an OpenVX overlay consisting of a processor resource, such as a host ARM processor and SVP for some acceleration, and a library of VCI modules. Each VCI module is a custom pipeline that is predesigned by an FPGA expert to accelerate one OpenVX kernel function. On a kernel-by-kernel basis, each custom pipeline can achieve the same level of performance as custom logic. At application load time, the OpenVX runtime will generate an application-specific bitstream using partial reconfiguration to activate multiple VCIs at the same time.

However, choosing the right VCI modules is a difficult problem. Not all of the compute kernels available in the OpenVX VCI library will be used by an application, so it would be wasteful to implement all of them at once. Instead, to fit a limited budget, we can reserve a partially reconfigurable region (PRR) in the FPGA and load a partial bitstream for each required VCI into the PRR. This step requires relocatable bitstreams and a tool like GoAhead [10]. In this way, software programmers can accelerate OpenVX applications on FPGAs within an area budget without any hardware or FPGA design skills.

The rest of this paper covers the process in detail.

II. System Overview

An OpenVX application consists of the part written by the programmer, plus a runtime environment provided with a target-specific OpenVX SDK. At runtime, an OpenVX application must build a graph of compute kernels, verify the graph, and process (run) the graph. Because of this flexibility, the graph contents are unknown before runtime.

The inputs and outputs of the graph are images. OpenVX exploits on-chip buffers and pipelining in accelerators by allowing an image to be divided into smaller tiles, where one tile at a time is processed to completion through the graph. A few OpenVX kernels cannot be tiled, so they require complete image reconstruction at their inputs before those graph nodes can be executed.

Figure 1 provides a general view of the OpenVX overlay. It consists of an ARM Cortex-A9 host processor, the VectorBlox MXP SVP [8], an empty/reserved PRR in the FPGA fabric, and a customized OpenVX SDK. The SDK includes a runtime environment, a library of OpenVX kernels written as C/C++ functions accelerated by MXP, and a library of prebuilt Vector Custom Instructions (VCIs) which are custom logic pipelines implemented in the FPGA fabric to further accelerate some OpenVX kernels.

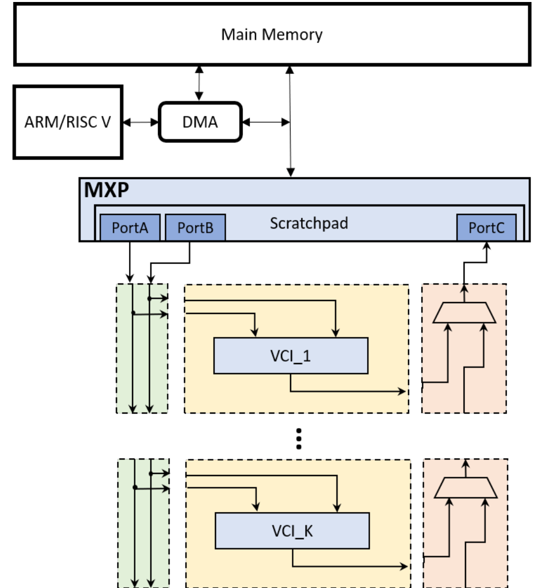

Figure 1: System overview

The user selects the size of the MXP (the number of parallel vector lanes L, where L must be a power of 2) and the size of the PRR (a rectangular region of the fabric). The user also selects the size of the MXP scratchpad, which typically ranges from tens of kB to a few MB. When used with a VCI, the scratchpad provides two streaming source operands on PortA and PortB, and accepts results on PortC. Input sources are broadcast to all VCIs on PortA and PortB. A multiplexer chain selects which result is written back to the scratchpad on PortC. The multiplexer chain is implemented as part of each VCI and connects it with the SVP. As part of MXP configuration, each VCI opcode must specify its own fixed throughput level between 1 and $4L$ bytes per clock cycle; all three ports use the same rate.

The VCI library consists of select OpenVX kernels, each with multiple implementations corresponding to each desired level of throughput (from 1 to $4L$ bytes per cycle). To use a VCI, the associated bitstream must first be loaded into the PRR; more complex kernels that process more bytes per cycle will consume more space within the PRR. Up to K VCIs can be merged into a single PRR using bitstream relocation and merging features of GoAhead [10]. Because MXP supports only 16 VCI opcodes, $K \leq 16$.

The OpenVX runtime system, described further below, analyzes the application graph and uses an optimization algorithm to determine which kernels are to be executed as a VCI, with the remaining kernels to be executed on the MXP. The optimization algorithm aims for maximum speedup,

subject to the constraint that all VCIs must fit within the PRR. Hence, kernels that represent the most computation will be executed as a VCI.

Through this use of a predefined VCI library, GoAhead, partial reconfiguration, and the OpenVX runtime optimization framework, regular software programmers do not need any FPGA-specific knowledge.

A. Mapping OpenVX Applications to an FPGA

The process of mapping an OpenVX application involves three steps. The first step, designing the overlay, is done in advance by FPGA experts and involves creating the SVP-accelerated kernels and the VCIs. Each OpenVX kernel is given a pure software implementation on the SVP, where the throughput, in terms of pixels it can produce in a time unit, is recorded for various image tile sizes. In addition, each OpenVX kernel is written as heavily parameterized C++ for Vivado HLS. For each level of throughput, the JANUS framework is used to search the design space enabled by the parameterization and find the smallest implementation that meets the throughput target. Instead of JANUS, manual optimization is also possible. Each VCI implementation is noted with its throughput, area, and partial bitstream.

The second step, building an overlay instance, is also done an FPGA expert who defines the SVP, scratchpad, and PRR sizes. This can be done independently of the third step.

The third step, creating an application, is done by a software programmer following the OpenVX standard. The programmer selects an overlay instance to run the application along with the OpenVX runtime system. Given an OpenVX graph and an image size, the runtime uses an execution time model to determine the best tile size, which nodes should be implemented as a VCI, and which VCI implementation (throughput level) to use. It also generates the PRR bitstream and loads it into the fabric.

These steps are described in more detail below.

B. Finding Different Implementations

Consider an application described as a graph G:

$$G = (V, E)\ ,\ V = \{f_1, f_2, ..., f_N\} \tag{1}$$

For each node f_m we can find N_{SVP} different SVP implementations $S_m^1, S_m^2, ..., S_m^{N_{SVP}}$ as well as N_{VCI} different VCI hardware implementations $P_m^1, P_m^2, ..., P_m^{N_{VCI}}$. Each SVP implementation S_m^s can perform functionality of f_m on an image tile in $t(S_m^s)$ time. Each VCI implementation P_m^s can perform functionality of f_m with area cost $A(P_m^s)$.

Considering available resources, such as the size of the PRR and the speed of SVP and VCI implementations, the OpenVX runtime system decides which nodes should be run as SVP software and which nodes should be accelerated with a VCI. After enumerating different implementations for each OpenVX node, to minimize the search space, the OpenVX runtime system prunes any dominated implementation points. Moreover, it uses "execution time analysis" and tile/image DMA time to estimate the overall execution time for each implementation.

C. Execution Time Analysis

To execute a general OpenVX graph, the runtime system needs to fetch each image tile from main memory to the scratchpad and execute the whole graph, one node at a time (either as SVP or VCI implementations). In every stage of traversing the graph, the intermediate data is saved in the scratchpad. This means the tile size must be calculated based on the available scratchpad size. After executing all the graph nodes on a tile, results are written back to main memory before fetching the next tile. The execution time for these components is discussed below.

1) SVP Software Implementation: The execution time of a graph G with N nodes $f_1, f_2, ..., f_N$ and a set of selected SVP implementations $S_1, S_2, ..., S_N$ on image tile T_j is t_{T_j}. This is calculated as:

$$t_{T_j} = t_{DMA_{M2S}} + [\sum_{i=1}^{N} t(Si)] + t_{DMA_{S2M}} \tag{2}$$

The overall execution time for N_T tiles in the image is:

$$t_A = \sum_{j=1}^{N_T} t_{T_j} \tag{3}$$

2) Accelerated VCI Implementation: Consider node f_m in the compute graph G. Instead of using SVP implementation S_m with execution time $t(S_m)$, it is possible to use a VCI hardware implementation P_m with execution time $t(P_m)$ to improve the execution time.

For the VCI, we can define kernel throughput $\Theta(P_m)$ as the number of bytes consumed/produced in one clock cycle. This can be used to calculate VCI execution time $t(P_m)$:

$$t(P_m) = \frac{TileSize}{\Theta(P_m) \cdot F_{max}} \tag{4}$$

where F_{max} is the clock speed of the SVP (eg, 100MHz).

3) Pruning Slow Standalone VCIs: To reduce the search space, VCI implementations that are slower than SVP implementations are pruned. Hence, we only consider VCIs that satisfy $t(S_m) > t(P_m)$.

III. Experimental Results

In this section, we estimate the speedup provided by the SVP and various VCI configurations. Three different hardware configurations are considered: the baseline, consisting of the ARM Cortex-A9 processor running at 667MHz; the SVP, in two configurations V4 and V8 with four and eight 32-bit lanes, respectively, running at 100MHz without any VCI; and the SVP (in two configurations) with VCI also running at 100MHz. We generated V4 and V8 versions of the SVP hardware on a ZedBoard, and using that hardware we gathered runtime data for the SVP kernels.

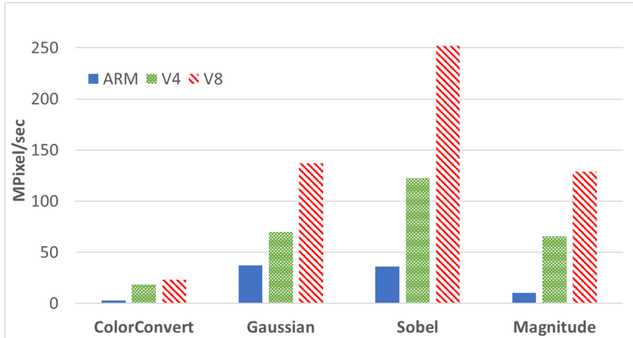

Figure 2: Speed of ARM Cortex-A9 compared to SVP

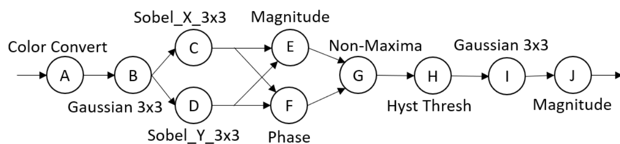

Figure 3: Graph representation of $Canny - blur$

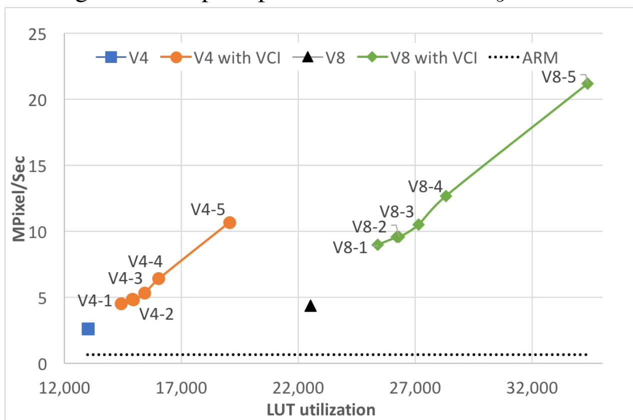

Figure 4: $Canny - blur$ speed versus area

More than 25 different OpenVX kernels are implemented as SVP and HLS versions. For the HLS versions, our JANUS space/time scaling tool [6] builds a library of area-minimized VCI implementations at different throughput levels.

The throughput results for four specific kernels are shown in Figure 2. On average across those kernels, the SVP V4 achieves a 4.6 times speedup over the Cortex-A9, and the SVP V8 achieve an 8 times speedup.

Performance of a Canny-blur filter, shown in Figure 3, is given in Figure 4. Although SVP V4 and V8 are able to achieve better throughput than ARM, the use of VCIs offers significant improvement for only a modest additional area increase. In Figure 4, results for five different PRR sizes are labeled as '-1' through '-5'. The VCIs selected to fit within each of the labeled PRR sizes are given in Table II. In this table, the letter indicates which node(s) from the graph in Figure 3 are accelerated with a VCI. As the PRR size increases, more VCIs are fit into the PRR and greater speedup is achieved. Although not shown in the table, the throughput of each VCI is also selected to maximize throughput without exceeding the capacity of the PRR.

Table II: List of kernels implemented as VCIs in Figure 4

Implementation	Kernels implemented as VCIs
V4-1 and V8-1	A B
V4-2 and V8-2	A B, D
V4-3 and V8-3	A B C D
V4-4 and V8-4	A B C D, H, I
V4-5 and V8-5	A B C D, H, I, J

IV. Conclusion

This paper presents a method for accelerating OpenVX applications on an FPGA using an overlay. The overlay has a runtime component which analyzes the application, determines which OpenVX kernels should be implemented as vector custom instructions (VCIs) within a partially reconfigurable region (PRR), and loads an application-specific bitstream. The customized OpenVX SDK contains an optimization framework as part of the runtime system, as well as a pre-generated a library of area-minimized VCI implementations defined at multiple throughput levels. The overlay achieves speedups beyond what a plain SVP can accomplish. For example, on *Canny-blur*, the 8-lane SVP without VCIs is 5.3 times than an ARM Cortex-A9; using a PRR about half the SVP size boosts speedup to 18.5 times with VCIs. This allows OpenVX programmers to achieve hardware-like speeds with no FPGA design knowledge.

References

[1] J. Hegarty *et al.*, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, pp. 144:1–144:11, 2014.

[2] J. Hegarty *et al.*, "Rigel: Flexible multi-rate image processing hardware," *ACM Trans. Graph.*, pp. 85:1–85:11, 2016.

[3] J. Pu, S. Bell *et al.*, "Programming heterogeneous systems from an image processing DSL," *ACM Trans. Archit. Code Optim.*, pp. 26:1–26:25, 2017.

[4] Intel, "Openvino toolkit," 2018. [Online]. Available: https://software.intel.com/en-us/openvino-toolkit

[5] S. Taheri, J. Heo, P. Behnam, P. Veidenbaum, and A. Nicolau, "Acceleration framework for FPGA implementation of OpenVX graph pipelines," Center for Embedded and Cyber-Physical Systems, UC Irvine, Tech. Rep., 2018.

[6] H. Omidian and G. G. Lemieux, "Exploring automated space/time tradeoffs for OpenVX compute graphs," in *ICFPT*, 2017, pp. 152–159.

[7] H. Omidian and G. G. Lemieux, "JANUS: A compilation system for balancing parallelism and performance in OpenVX," in *J. of Physics: Conf. Series*, vol. 1004. IOP Pub., 2018.

[8] A. Severance and G. G. F. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *CODES+ISSS*, 2013, pp. 1–10.

[9] A. Severance, J. Edwards *et al.*, "Soft vector processors with streaming pipelines," in *FPGA*, 2014, pp. 117–126.

[10] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A partial reconfiguration framework," in *FCCM*, 2012, pp. 37–44.