
Sajjad Taheri\textsuperscript{1}, Payman Behnam\textsuperscript{2}, Eli Bozorgzadeh\textsuperscript{1}, Alexander Veidenbaum\textsuperscript{1}, Alexandru Nicolau\textsuperscript{1}

\textsuperscript{1}Department of Computer Science, UC Irvine

\textsuperscript{2}School of Computing, University of Utah

27th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays
Motivation
FPGAs for Computer Vision Acceleration

Computer Vision

✓ Important applications in automation, entertainment, healthcare, etc.
✗ Complex algorithms and demanding workloads

FPGAs offer

✓ Inherent parallelism, high performance, low latency, energy efficiency.

Efficient FPGA acceleration requires hardware design expertise and considerable amount of engineering man-hours.
AFFIX Goal

- FPGA acceleration flow for high level graph-based computer vision algorithms
- "Conventional" computer vision algorithms based on OpenVX specification.
  - Related work include domain specific languages to represent image processing pipeline: e.g., Halide (2016), PolyMage (2016)

Diagram:
- Application Specification
- OpenVX Algorithm Design
- Software Implementation
- Hardware Implementation
Our Approach

Domain Knowledge

- Design accelerator architecture by considering image processing kernel behaviors
  - Domain Specific Representation and implementation
- Apply algorithm-specific optimizations on the algorithm graphs

FPGA Design Methodology

- Use High Level Synthesis (OpenCL).
  - Portability
  - Maintainability
- Apply Hardware specific optimizations
Outline

1. Motivation

2. Customizable library of vision functions

3. AFFIX framework

4. Evaluation

5. Conclusion and future direction
Overview of OpenVX

OpenVX

- Open, royalty-free standard for cross platform acceleration of computer vision applications
- Performance and power-optimized computer vision processing
- Graph-based execution model to enable task and data-independent execution
Overview of OpenVX

OpenVX defined objects

- Kernel: Abstract representation of a vision functions, predicates, and delay objects
- Node: An instance of a kernel
- Virtual Image: Represents an image
- Graph: A set of nodes connected in a directed acyclic fashion.

Graph Lifecycle
Customizable library of vision functions
Vision functions are categorized based on their data access patterns.

- Pixel wise
- Downsample
- Geometric
- Statistics
- Stencil
- Table Lookup

We have implemented streaming kernels for each category in OpenCL.
## Kernels from OpenVX Specification 1.2

<table>
<thead>
<tr>
<th>Category</th>
<th>Formal Definition</th>
<th>OpenVX Vision Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel-wise</td>
<td>$out(x, y) = f(in(x, y))$</td>
<td>Absolute difference, Accumulate, Accumulate squared, Bitwise operations, Channel combine, Channel extract, Color convert, Convert bit depth, Magnitude, Phase, Pixel-wise multiplication, Threshold, Min, Max</td>
</tr>
<tr>
<td>Fixed-rate Stencil</td>
<td>$out(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} g(in(x + i, y + j))$</td>
<td>Box filter, Sobel, Non-maxima suppression, Custom convolution, Erode, Dilate, Gaussian blur, Nonlinear filter, Integral image, Median filter</td>
</tr>
<tr>
<td>Multi-rate Stencil</td>
<td>$out(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} g(Nx + i, Ny + j))$</td>
<td>Down-sample, Scale image</td>
</tr>
<tr>
<td>Statistical</td>
<td>$out = \sum_{i=0}^{\text{Width}} \sum_{j=0}^{\text{Height}} g(in(i, j))$</td>
<td>Histogram, Mean, Standard deviation, Min,max location</td>
</tr>
<tr>
<td>Geometric</td>
<td>$out(x, y) = in(h(x, y), h'(x, y))$</td>
<td>Remap, Warp affine, warp perspective table lookup</td>
</tr>
<tr>
<td>Table lookup</td>
<td>$out(x, y) = table[in(x, y)]$</td>
<td>Equalize histogram, Fast corners, Harris corners, Gaussian image pyramid, Canny edges, LBP, HOG, HoughLinesP</td>
</tr>
<tr>
<td>Non-primitive</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>
Template-based FPGA Kernel Implementation

Templates based on vision function categorization.

- Easier testing and optimization (5 cases vs 50+ cases)
- Easier to extend OpenVX with user defined functions

General implementation of different kernel categories
Template-based FPGA Kernel Implementation

Kernels can be specialized with

- Specific compute function (similar to function pointers)
- Input and output types (OpenCL standard types)
- SIMD size (1 to 32)
- Sliding Window size
- Local memory configuration (banking, etc.)
- Arithmetic precision (double, float and potentially fixed point)

Channels can be specialized with

- Channel type and width
- Channel depth
A Domain Specific Language (DSL) on top of OpenCL.

- C-style macros are used to instantiate and specialize generic templates in OpenCL.

Vision functions such as Gaussian blur, erode, dilate, and box filter can be implemented with this template.
C-style macros are used to instantiate and specialize channels in OpenCL as well.

- Channels are dynamic FIFOs
- Kernels communicate through channels

**Example**

```c
#define SIMD_SZ 8
CHANNEL(ch_con_col, uchar, SIMD_SZ)
CHANNEL(ch_thresh, uchar, SIMD_SZ)
THRESH(ch_conv_col, SIMD_SZ, thresh_val, ch_thresh)
```
AFFIX framework
Automatic generation of accelerator systems from input algorithms.

1. Checks input algorithm graph for correctness
   Input graph is represented in a textual format
Automatic generation of accelerator systems from input algorithms.

1. Checks input algorithm graph for correctness
   Input graph is represented in a textual format

2. Analyses, partitions, and optimizes the algorithm graph
AFFIX Framework Flow

Automatic generation of accelerator systems from input algoritms.

1. Checks input algorithm graph for correctness
   Input graph is represented in a textual format

2. Analyses, paritions, and optimizes the algorithm graph

3. Generates code for both FPGA and CPU components
OpenVX Example: Lane Detection Algorithm

Inputs: Image in_img, Matrix transform_mat, Matrix filter, Integer thresh_val
Output: Array line_segments
begin
  y_img ← convert_to_grayscale(in_img)
  b_img ← warp_perspective(y_img, transform_mat)
  f_img ← filter(b_img, filter)
  t_img ← threshold(f_img, thresh_val)
  line_segments ← houghlinesp(t_img)
return line_segments
end

Algorithm

OpenVX Graph

Lane Detection Algorithm Demonstration (Input video obtained from software.intel.com)
Simplify and optimize OpenVX graphs

- Decomposition of OpenVX vision functions into simpler primitives
- Removal of nodes that are not connected to an output node
- Separable and symmetric 2D filter implementation
- More steps can be incorporated...

Step 1: Input Lange Detection Algorithm Graph.
Simplify and optimize OpenVX graphs

- Decomposition of OpenVX vision functions into simpler primitives
- Removal of nodes that are not connected to an output node
- Separable and symmetric 2D filter implementation
- More steps can be incorporated...

Step 2: Lowered Lane Detection Algorithm. RGB-to-YUV node is replaced with RGB2Y, RGB2U, and RGB2V nodes. Channel extract node drops U and V images.
High-Level Analysis and Optimization

Simplify and optimize OpenVX graphs

- Decomposition of OpenVX vision functions into simpler primitives
- Removal of nodes that are not connected to an output node
- Separable and symmetric 2D filter implementation
- More steps can be incorporated...

Step 3: Optimized Lane Detection Algorithm
Algorithm graph is partitioned based on vision functions data dependencies.

Graph with only pixel-wise, stencil, and Lookup nodes can be fully pipelined.

Statistical Nodes (h,f) must be last nodes of any pipeline.

Statistical Nodes (h,f) must be last nodes of any pipeline.
Not all vision functions are accelerated on the FPGA

- Kernels with irregular data access
- Kernels with high resource usage or complex implementations

Graph partitioning in case of CPU nodes: predecessors and successors of CPU nodes (h, f) cannot be mapped to a same pipeline.
FPGA partitions are implemented in OpenCL

Partitions are executed in topological order.
FPGA Partitions are described in OpenCL

```c
#define SIMD_SZ 8
#define WIN_SZ 240

// Partition 1
CHANNEL(ch_in, uint, SIMD_SZ)
CHANNEL(ch_y, uchar, SIMD_SZ)
SRC(ch_in)
RGBTOY(SIMD_SZ, ch_in, ch_y)
SAVE(ch_y)

// Partition 2
CHANNEL(ch_warped, uchar, SIMD_SZ)
CHANNEL(ch_conv_row, uchar, SIMD_SZ)
CHANNEL(ch_con_col, uchar, SIMD_SZ)
CHANNEL(ch_thresh, uchar, SIMD_SZ)
float[9] conv_col = {...};
float[3] conv_row = {...};
WARP_LOAD(ch_warped, SIMD_SZ)
CONV_ROW(ch_warped, ch_conv_row, 9, conv_row, ...)
CONV_COL(ch_conv1, ch_con_col, 3, conv_col, ...)
THRESH(ch_conv_col, SIMD_SZ, thresh_val, ch_thresh)
SINK(ch_thresh)
```

Simplified OpenCL Implementation of Lane Detection
They implement *Algorithm* class interface:

- Describe both hardware and software pipeline and their ordering
- Implement pre- and post- processing steps for each partition
- Guide OpenCL runtime (Memory allocation, etc)
- USes OpenCV to implement CPU vision functions
Overall System Components

Compilation

$\texttt{aoc [flags]}$

$\texttt{HW code (.cl)}$

$\texttt{SW Code (cpp)}$

$\texttt{HW component (.aocx)}$

$\texttt{SW plugin (.so)}$

Runtime System

$\texttt{OpenCL run-time/PCIe driver}$

$\texttt{host path-to-aocx-file path-to-so-file input-stream [flags]}$

Software Components

Compilation

$\texttt{$g++ -shared -fPIC$}$
Diverse set of vision algorithms developed
- Application domain, number of vision functions, and combination of vision function types

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Domain</th>
<th>No VX Fns</th>
<th>No Extension Fns</th>
<th>No CPU Fns</th>
<th>No Geo Fns</th>
<th>No Stats Fns</th>
<th>No Graph Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canny Edges</td>
<td>Image Processing</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Automatic Contrast</td>
<td>Image Processing</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Lane Detection</td>
<td>Image Processing</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Color Copy</td>
<td>Color Printing</td>
<td>42</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Census Transform</td>
<td>Visual Descriptor</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SIFT keypoints</td>
<td>Visual Descriptor</td>
<td>116</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Workload Characterization

Test Platform Spec

Intel Arria-10GX board connected to host with Intel Core i7 4770 processor via PCIe Gen3x8.
Arria 10 Resource utilization and FMax
Evaluation: Results

Arria 10 Resource utilization and FMax

1. No significant drop on FMax by increasing SIMD size
2. Resources usage grows linearly by increasing SIMD size
## Evaluation: Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size</th>
<th>CPU (AVX+8 Cores)</th>
<th>FPGA SIMD=1</th>
<th>FPGA SIMD=2</th>
<th>FPGA SIMD=4</th>
<th>FPGA SIMD=8</th>
<th>FPGA SIMD=16</th>
<th>FPGA SIMD=32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canny Edges</td>
<td>3840x2160x1</td>
<td>15 ms</td>
<td>28 ms</td>
<td>15 ms</td>
<td>8 ms</td>
<td>5 ms</td>
<td>4 ms</td>
<td>4 ms</td>
</tr>
<tr>
<td></td>
<td>3840x2160x4</td>
<td>21 ms</td>
<td>84 ms</td>
<td>45 ms</td>
<td>22 ms</td>
<td>13 ms</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Automatic Contrast</td>
<td>3840x2160x4</td>
<td>46 ms</td>
<td>57 ms</td>
<td>27 ms</td>
<td>14 ms</td>
<td>12 ms</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Lane Detection</td>
<td>3840x2160x4</td>
<td>83 ms</td>
<td>45 ms</td>
<td>32 ms</td>
<td>23 ms</td>
<td>19 ms</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Color Copy</td>
<td>3840x2160x4</td>
<td>12 ms</td>
<td>27 ms</td>
<td>14 ms</td>
<td>7 ms</td>
<td>4 ms</td>
<td>3 ms</td>
<td>3 ms</td>
</tr>
<tr>
<td>Census Transform</td>
<td>3840x2160x1</td>
<td>223 ms</td>
<td>56 ms</td>
<td>27 ms</td>
<td>14 ms</td>
<td>10 ms</td>
<td>10 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>SIFT keypoints</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average total execution time of CPU only vs CPU (Intel Core i7)+Arria10 accelerated algorithms with different SIMD sizes

- **SIMD_SIZE** is limited by PCIe width (256 bits)
- FPGA performance increases linearly by improving SIMD until hitting PCIe max bandwidth (Gen3x8)
Conclusion and future direction
## Conclusion

### Contributions

1. Library of customizable OpenVX vision functions implemented in OpenCL
   - Support for a wide variety of OpenVX vision elements (functions, data types, control structures)
   - Extendable with new user-defined functions
2. Algorithm graph and low level hardware optimization
3. Heterogeneous Graph partitioning and implementation

- Salable and efficient hardware generation
- Ease of development
Future Direction

- Power and energy optimization
- More sophisticated CPU scheduling using multiple processor cores
- Neural Network Integration

Questions?

contact sajjadt@uci.edu

This work is supported by the Intel Corp.