Exploring Automated Space/Time Tradeoffs for OpenVX Compute Graphs

Hossein Omidian
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
Email: hosseino@ece.ubc.ca

Guy G.F. Lemieux
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
Email: lemieux@ece.ubc.ca

***Abstract*—With the rise of FPGA-based implementation of Computer Vision (CV) applications, the need for a programming method that achieves the target throughput or area-budget while retaining flexibility is magnified. High-level synthesis (HLS) tools provide this opportunity while eliminating the necessity of hardware engineering knowledge. Existing methods of programming FPGAs with HLS require the user to explicitly manage resources at every stage in their algorithm in order to meet a specified area target or throughput target. In this paper, we provide a framework for meeting such targets with compute graphs specified in OpenVX, a C-based programming environment for computer vision. To do this, we build our own OpenVX system using Xilinx Vivado HLS, and add an algorithmic layer which allows the user to specify an area budget (while maximizing throughput) or a throughput target (while minimizing area). Our OpenVX system consists of a series of compute kernels, prewritten in C++ for HLS and heavily parameterized as well as an Intra-node Optimizer to enable the creation of different size/throughput targets using different image tile-sizes. It also uses a heuristic algorithm with an Inter-node Optimizer step which combines/splits kernels and then replicates them to minimize the area cost. We evaluate the system on typical OpenVX benchmarks under a variety of fixed area constraints, and find that our system is able to automatically achieve over 95% area utilization. We also evaluate the system with same benchmarks under variety of fixed throughput targets, and find our system saves up to 30% in area cost compared to manually parallelized implementations. Our results show our heuristic approach is able to hit the same throughput targets and save 19% area on average compared to existing ILP approaches. While existing methods can easily achieve a single design point, they are unable to automatically generate a set of solutions from the same source; a prominent capability embedded in our tool. Moreover our tool uses Inter-node Optimizer to find better space/time tradeoffs.**

I. Introduction

Implementing Computer Vision (CV) applications on different embedded systems has became a big focus of both industrial and academic communities in recent years. Since most CV applications are computationally intensive, using parallelism techniques for implementing them is considered a must. Although in theory CPU and GPU platforms can perform such intensive computations, power remains the main challenge since the power density of a single chip is reaching its limits [1]. Previous studies such as [2] suggest custom fixed-function hardware implementation is often the only way to implement image processing applications within the power limits of a mobile platform. Furthermore, custom hardware implementations can be used as accelerators in datacenters in order to solve the power issue [3].

Since an FPGA is significantly more energy efficient compared to a CPU or GPU, it can be a great alternative platform for running computationally intensive applications. Using FPGAs in Bing search by Microsoft [4] and the recent acquisition of Altera by Intel are just a few examples showcasing the increasing demand for FPGAs in different applications. Unfortunately, implementing custom hardware and FPGA designs are not widely popular amongst application developers mainly due to specific requirements of HDL programming, such as hardware engineering knowledge. Several industrial and academic HLS tools have been developed in order to provide an environment for users to describe their application in high-level languages such as C/C++ to avoid the difficulty of HDL programming. Existing methods of programming FPGAs with HLS require the user to explicitly manage resources at every stage in their algorithm in order to meet a specified area target or throughput target by manual loop unrolling or adding different pragmas to the code.

In this work, we introduce a novel approach to explore the space/time tradeoffs for OpenVX [5] compute graphs in order to find optimum solutions, meeting different area or throughput targets. OpenVX has been introduced as a C-based cross-platform standard for imaging and vision application domains, where a workload can be tiled and compute nodes can run on tiles instead of the whole image. We have implemented an infrastructure which automatically explores the area/throughput problem domain while leveraging commercial HLS tool advantages. We analyzed OpenVX kernels (nodes) to increase parallelism based on pipeline opportunities and different loop transformation strategies [6], [7]. After analyzing each kernel, we rewrote them in C++ while heavily parameterizing for HLS. Users can describe a CV computation as an OpenVX compute graph and then define either a throughput target, or an area budget. Our OpenVX system analyzes the compute graph and generates different implementations for each node with different area, IO and throughput characteristics by creating different HLS projects and passing them to Xilinx Vivado HLS. In order to get precise throughput/area information for different FPGA targets and avoid the implementations with deadlock, our tool automatically generates Vivado projects including a System-Verilog testbench for each implementation. In addition our tool uses node replication and node combining

978-1-5090-4825-0/17/$31.00

to cover more possible solutions by either increasing the tile size or improving throughput for existing implementations.

Our tool uses two internal optimization approaches. The first, based upon integer linear programming (ILP), is similar to previous work on task graph optimizations by Cong et al [8]. The second is a heuristic approach based on [9]. Although the ILP approach works well, maintaining the ILP optimization model within the tool prohibits the use of certain optimizations. Instead, the heuristic approach is able to perform object coalescing which cannot be done as an ILP formulation. This leads to area saving and less runtime compared to the ILP approach. There have been several recent studies on implementing image processing and OpenVX applications on FPGAs and exploring the area/throughput trade-off for them such as [10], [11] and [12]. These existing approaches either use a specific programming model which requires the user to learn a new programming language, or they implement a soft multi-core platform on FPGA and then run the application on it. Automatically exploring area/throughput tradeoffs using C++ as well as automatically finding optimum implementations for different area budgets or throughput targets, make our approach unique.

II. OpenVX based HLS

Figure 1 illustrates the detailed flow of the proposed tool. It receives an OpenVX compute graph and analyzes it to obtain a matched kernel for each node, all of which are heavily parameterized C++ functions with Xilinx AXI-Stream [13] in/out arguments. Taking into account the user area budget or throughput target, our tool creates different Vivado HLS projects in order to generate different implementations for each kernel. To minimize the search space, our tool prunes the dominated points in the design space using Zhong et.al. [14] approach. Previous studies [15] showed that the report generated by the HLS tool is not accurate and should therefore not be used for exploring the problem space. To obtain a precise throughput and area cost, our tool also generates Vivado projects using the HDL output generated from HLS tool. Using Vivado design suite, the tool is able to find throughput/area correlation for each kernel which will be eventually used in trade-off finding process. In addition, our tool uses a step called Intra-node Optimizer to generate more implementations to widen the trade-off solution space by filling the throughput/area correlation gaps in it. Finally the tool uses Trade-off Finder to find a good compromise between area and throughput. Below we discuss all of these steps in detail.

A. OpenVX programming model

OpenVX is a cross-platform C-based API standard for Computer Vision applications. It enables performance and power-optimized CV processing, especially important in embedded and real-time use-cases such as face, body and gesture tracking, smart video surveillance, advanced driver assistance systems (ADAS), and more. OpenVX specifies a higher level of abstraction which makes it suitable for targeting different

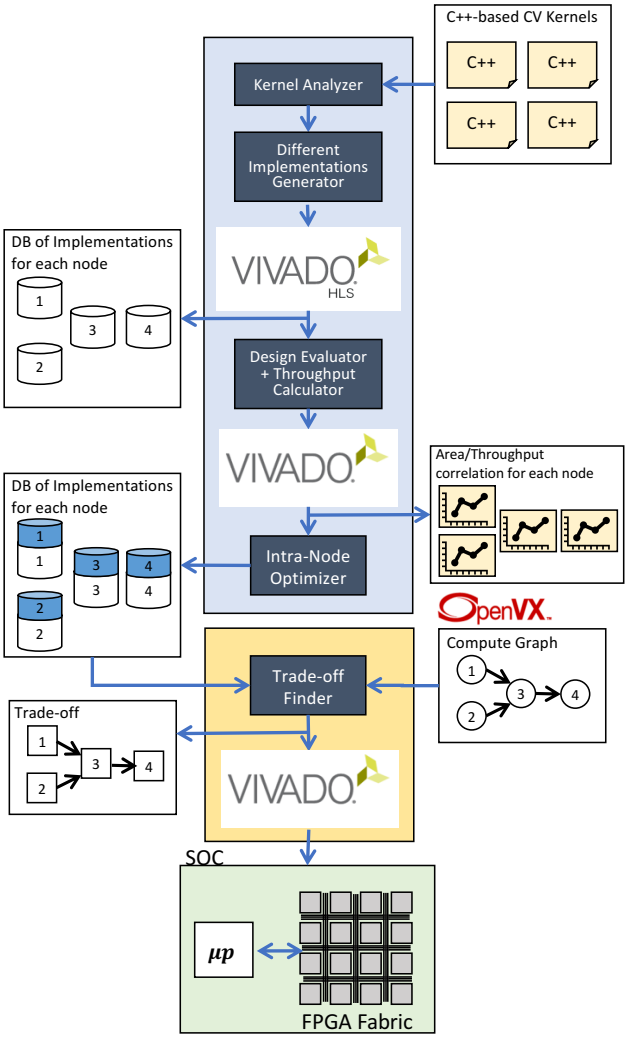


Fig. 1: Tool flow

computing architectures. Most CV applications can be described as a set of vision kernels (nodes) which communicate through input/output data dependencies. OpenVX describes this set of vision kernels in a graph-oriented execution model (Compute Graph) based on Directed Acyclic Graphs (DAGs). Figure 2a shows an OpenVX code example ($vxSobel3x3$) and 2b shows the corresponding graph for it.

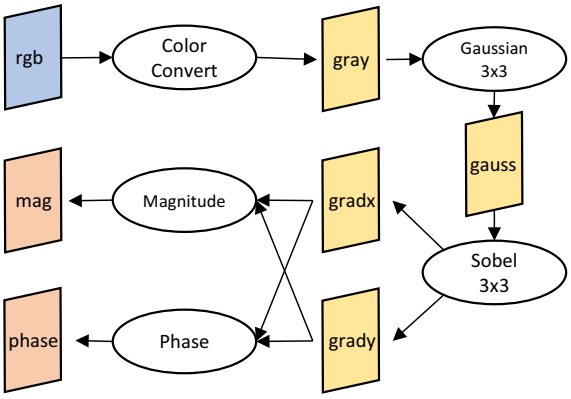
Since OpenVX compute graphs are DAGs, it makes them suitable candidates to be implemented as pipelined hardware accelerators on FPGAs. Below we discuss how our tool generates variety of different implementations for those hardware accelerators.

B. Finding different implementations

Consider an application described as a compute graph with N nodes $f_1, f_2, ..., f_N$. For each node f_m our tool tries to find different implementations $P_m^1, P_m^2, ..., P_m^{S_m}$ where each implementation P_m^s can perform functionality of f_m with area cost $A(P_m^s)$, number of pixels it can consume/produce

```
//vxSobel3x3 example
vx_node nodes[] = {
  vxColorConvertNode(graph,rgb, gray),
  vxGaussian3x3Node(graph,gray,gauss),
  vxSobel3x3Node(graph,gauss,gradx,grady),
  vxMagnitudeNode(graph,gradx,grady,mag),
  vxPhaseNode(graph,gradx,grady,phase)
};
```

(a) OpenVX source code



(b) Sobel graph

Fig. 2: OpenVX source code and graph representation

$NP(P_m^s)$ and initial interval $II(P_m^s)$. For implementation P_m^s, the area cost on FPGAs is calculated as:

$$A(P_m^s) = w_{lut}.LUT(P_m^s) + w_{dsp}.DSP(P_m^s) + w_{bram}.BRAM(P_m^s) \quad (1)$$

where $LUT(P_m^s)$, $DSP(P_m^s)$ and $BRAM(P_m^s)$ are the LUT, DSP and BRAM cost of implementation P_m^s. Note LUT weight (w_{lut}), DSP weight (w_{dsp}) and BRAM weight (w_{bram}) are different for various FPGA architectures. In this study we have set these weights differently for Xilinx, Altera and VPR architectures.

For node f_m and its implementation P_m^s, input "inverse throughput" $\vartheta_{in}(P_m^s)$ and output inverse throughput $\vartheta_{out}(P_m^s)$ for each input/output are calculated as:

$$\vartheta_{in}(P_m^s) = \frac{II(P_m^s)}{In(f_m)}, \vartheta_{out}(P_m^s) = \frac{II(P_m^s)}{Out(f_m)} \quad (2)$$

where $In(f_m)$ and $Out(f_m)$ are the number of data tokens that f_m consumes on the input data channel and produces on the output data channel during each firing, respectively. Note that inverse throughput shows the number of cycles to consume/produce per datum in its input/output channel. For most CV kernels, their input/output channels have matched inverse throughput, $\vartheta_{IO}(P_m^s)$. Kernel throughput is number of pixels consumed/produced in each clock cycle:

$$\Theta(P_m^s) = \frac{NP(P_m^s)}{\vartheta_{IO}(P_m^s)} \quad (3)$$

The Different Implementation Generator (DIG) module in our tool automatically finds the above mentioned implementations using heavily parametrized C++ based kernels. The DIG needs to be able to automatically find a wide range of different implementations to cover the solution space as much as possible. To have a better understanding of the complexity of this problem and the variety of possible solutions let's look at a simple example of a 3-node graph shown in Figure 3a. Figures 3b and 3c show two different approaches to satisfy a target throughput of 5; one reads 20 pixels and picks implementations with $II = 4$ for each node, the other reads 5 pixels and picks implementations with $II = 1$. Further, Figure 3c shows another approach for node f_2 in which instead of picking an implementation with $NP = 5$, it picks two implementations with $NP = 3$ and $NP = 2$ and splits the data between them. Figures 3b and 3c are just two examples of various iterations of II, NP and splitting/joining nodes which can be utilized to find the solution. In addition, the strategy of reading image data from the main memory can vary for different implementations when NP and II change. The strategy impacts DMA configuration and alignment network design which leads us to different overhead cost for each.

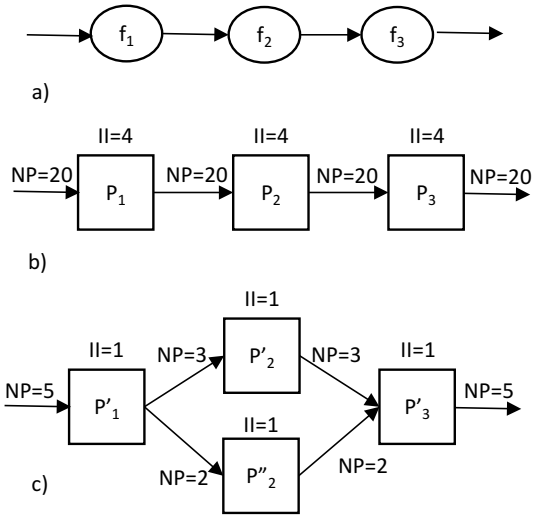


Fig. 3: Two different approaches for satisfying $\Theta = 5$

The above-mentioned example shows that for every area budget or throughput target, there are a variety of different acceptable solutions. This makes the trade-off finding problem a complex and nontrivial problem. It also shows the importance of generating a variety of different implementations with different NP, II and area cost to cover the solution space as much as possible.

C. CV Accelerator on FPGA

Before describing the tool flow in more detail, it is beneficial to go through the overall system description first. As mentioned earlier, the main goal of this study is to find a good area/throughput trade-off for CV applications by generating different implementations which is done through changing tile-width and/or using different function implementations inside

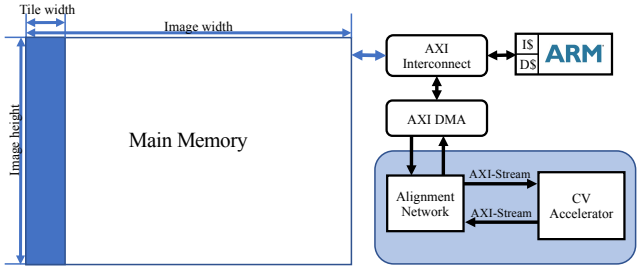


Fig. 4: System view implemented on Xilinx FPGA

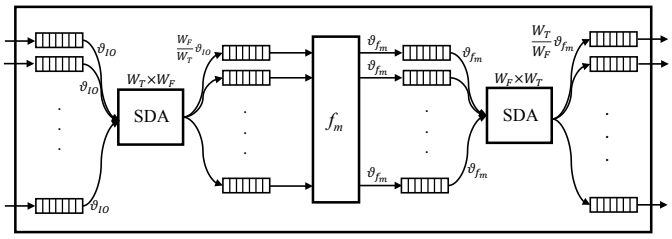


Fig. 5: Internal view of a general node in CV hardware accelerator

the kernel. Figure 4 gives a high-level system view of a CV accelerator implemented on Xilinx FPGAs. The host processor is responsible for configuring DMA (i.e. Xilinx AXI DMA IP core) to read/write image data in strides from the main memory. On the other end, DMA sends/receives image data to/from the accelerator in AXI-Stream protocol. Since the DMA data-width should be a power of two and the CV accelerator may read data at a different width in pixels, there should be a Data Alignment Network implemented as mixed-width FIFOs in between to align the data sent back and forth between DMA and accelerator.

Figure 5 provides an internal view of a general node in a CV hardware accelerator. Representing tile-width with W_T, a general node m with implementation P_m^s consumes W_T pixels ($NP_{in}(P_m^s) = W_T$) as stream in and generates W_T pixels as stream out with inverse throughput equal to ϑ_{IO}. Since the hardware function inside the node might consume/produce different number of pixels, two Stream Data Adjusters (SDAs) are added to either end. Assuming the hardware function can consume/produce W_F pixels in its input/output channels, the input SDA should get W_T pixels from input and pass W_F pixels to the function with inverse throughput equal to $\frac{W_F}{W_T}\vartheta_{IO}$. On the other end, output SDA gets W_F pixels with inverse throughput ϑ_{f_m} and provides W_T pixels with inverse throughput equal to $\frac{W_T}{W_F}\vartheta_{f_m}$. As Figure 5 shows, four layers of FIFO are added in between in order to match different throughputs in various stages. To prevent losing data, FIFO depths should be calculated carefully to prevent any data losses.

1) Stream Data Adjuster in more details: SDA can deal with two different types of kernels; $Pixel2Pixel$ kernels and $Window2Pixel$ kernels.

$Pixel2Pixel$ kernels such as $vxConvertColor$ produces

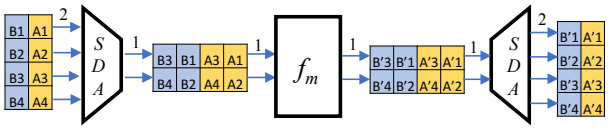


Fig. 6: Pixel2Pixel kernel example, $W_T = 4, W_F = 2$

one pixel for each pixel received:

$$\tilde{P_{i,j}} = f(P_{i,j}) \tag{4}$$

Figure 6. This shows how SDA functions as an adjuster for a simple Pixel2Pixel kernel which receives/produces 4 pixels in every 2 clock cycles and its hardware function consumes/produces 2 pixels in every clock cycle. In order to match the stream rate between IO and the function, SDA simply uses upstream to downstream transformation by splitting data in its input and sending it to the function. On the other end it joins data coming from the function and sends it to the output. In this example $\frac{W_T}{\vartheta_{IO}}$ is equal to $\frac{W_F}{\vartheta_{f_m}}$, so it only needs to capture W_T pixels in its FIFO (2×2 FIFO) every two clock cycles.

$Window2Pixel$ kernels such as $vxSobel3x3$ consume a window of pixels for every produced pixel in their outputs:

$$\tilde{P_{i,j}} = f(\begin{bmatrix} P_{i-\delta,j-\delta} & \dots & P_{i-\delta,j} & \dots & P_{i-\delta,j+\delta} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{i,j-\delta} & \dots & P_{i,j} & \dots & P_{i,j+\delta} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{j+\delta,j-\delta} & \dots & P_{i+\delta,j} & \dots & P_{i+\delta,j+\delta} \end{bmatrix}), \delta = \frac{w-1}{2} \tag{5}$$

A kernel with a tile-width of W_T and window size of $w \times w$ consumes $W_T + w - 1$ pixels and produces W_T pixels in every firing.

Figure 7 shows how SDA handles the stream adjustment for a Window2Pixel kernel with W_T equal to 4, W_F equal to 2 and window size equal to 3×3. This kernel receives 6 pixels and produces 4 pixels every 2 clock cycles. The hardware function produces 2 pixels every clock cycle which means it needs to get 4 pixels every clock cycle. In this case, SDA splits the input stream data maintaining some data overlap. This data overlap has two consequences; overhead of 2 pixels for every 6 pixels consumed by the kernel in each firing, and the need for a line-buffer with minimum depth of 5. Figure 8 illustrates a general Window2Pixel kernel with tile-width of W_T and window size of 3×3 with W_F equals to $\frac{W_T}{N}$. Since the function needs to receive $\frac{W_T}{N} + 2$ pixels in its firing the overhead is $2N$. Also to produce the first output, the function needs to have a line buffer with a minimum depth of $2N + 1$.

D. Heavily parameterized C++-based OpenVX kernels

A set of heavily parameterized C++ based OpenVX kernels with AXI-Stream input/output have been implemented to generate different implementations for each kernel. Each kernel is parameterized in two levels, IO level and core level.

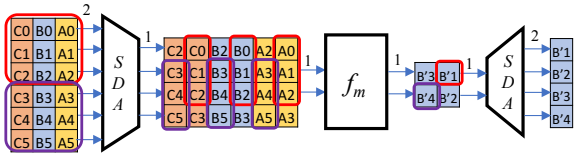


Fig. 7: Window2Pixel kernel example, $W_T = 4, W_F = 2$

The number of pixels that a kernel consumes/produces in its IO as well as the number of pixels needed to provide/gather to/from its core are parameterized in IO level. For each kernel, the main core function has been analyzed to find all degrees of parallelism and then heavily parameterized. This can be done by labeling all loops and generating a set of suitable $pragmas$ saved as a JSON file for each kernel. Considering this parameterization, the tool can generate different implementations with different number of inputs ($NP(P_m^s)$), area cost ($A(P_m^s)$) and initiation interval ($II(P_m^s)$). Figure 9 shows the area, throughput per input ($\frac{1}{II}$) and tile-width correlation of different implementations for $Gaussian3x3$ kernel. Each dot represents an implementation.

E. Intra-node Optimizer

In addition, an Intra-node Optimizer step in the tool generates a wider range of implementations. Intra-node Optimizer replicates and combines existing implementations in order to widen the solution space. Node replication can be used to either increase the throughput or widen the tile-width. Figure 10a demonstrates a general $Pixel2Pixel$ kernel with inverse throughput ϑ_{IO} and tile-width W_T. In order to improve the throughput (reduce the ϑ_{IO}), the tool replicates the node and sends data to each replica with a round-robin order. Figure 10b shows how the tool improves the throughput N-times by making N replicas of the original kernel. Figure 10c shows how the tool increase the tile-width by replicating the kernel. Because of data dependencies in $Window2Pixel$ kernels, replication only can be used for increasing tile-width. Our tool replicates $Window2Pixel$ kernels considering the window size and handles the data passing. Figure 11 demonstrates how the tool passes data to each replica when windows must overlap, e.g. in 2D convolutions.

Intra-node Optimizer also combines existing implementations and then replicates the combined node on the fly. Figure 12 shows a simple node combining example. Assume node f_n's throughput is N times bigger than node f_m's throughput. Two different approaches are shown in Figure 12 to match the throughputs: the first approach is replicating node f_m, N times and using a $1 \rightarrow N$ data splitter so that node f_n can send data to those replicas in a round-robin order (Figure 12a). In the second approach shown in Figure 12b, another implementation for node f_n with a throughput equal to twice node f_m's throughput is found (f'_n). Then the nodes f'_n and f_m are combined and the combined node is replicated $\frac{N}{2}$ times. Note that second approach needs a $1 \rightarrow \frac{N}{2}$ data splitter and is much smaller than the $1 \rightarrow N$ data splitter in the first approach.

All the above mentioned techniques are used in Intra-node Optimizer to find a wide range of implementations for each kernel which widens the solution space for the area/throughput scaling problem. Below we discuss our trade-off formulation and solutions.

F. Trade-off Finding Formulation and Solutions

Trade-off finding has two different modes in our tool.

- Given an available area on chip A_C and different implementations for each node f_j, which implementation P_j^i should be selected and how many replicas nr_j^i are needed in order to maximize application throughput Θ_A subject to the constraint the application area cost A_A is not bigger than A_C.
- Given a throughput target Θ_{tgt}, and different implementations for each node f_j, which implementation P_j^i should be selected and how many replicas nr_j^i are needed in order to minimize area cost A_A subject to the constraint the application throughput Θ_A is bigger than Θ_{tgt}.

To solve described trade-off finding problem we used an ILP approach as well as a heuristic approach.

1) Using Integer Linear Programming: This problem can be defined as an Integer Linear Programming (ILP) model similar to Cong et al [8] formulation and solved with an ILP solver such as GLPK [16] which goes through all the possibilities in the solution space and find the optimum solution, subject to the constraints. Although ILP solvers can solve these problems, the approach does have two shortcomings:

- Lack of flexibility: the problem must be defined in advance and it's not possible to change the problem's structure while solving it by ILP. In other words, combining or splitting nodes are not possible while using ILP.
- Time inefficient: In our experiments, ILP was usually slower than the heuristic algorithm used.

2) Using Heuristic Approach: This problem can be solved using heuristic approaches instead of using ILP. Omidian et al [9] used throughput analysis and throughput propagation as well as node replication and node combining to implement stream application on a coarse grain architecture. We adopted this approach and modified it for implementing CV applications on FPGAs using HLS.

III. Experimental Results

Our experiments are carried out in two parts. We evaluate our strategies of finding a good area/throughput tradeoff by targeting different FPGA architectures. Then we evaluate our tool by setting different throughput targets to examine whether we can find the optimum solution for each target.

To show the capabilities of our approach, we have utilized the following benchmarks implemented as OpenVX compute graph:

- *Sobel* is a Sobel-filter based edge detection with 5 nodes.
- *Canny* implements Canny edge detector with 6 nodes.
- *Harris* implements Harris corner detector with 6 nodes.

All the kernels inside each of the aforementioned benchmarks are analyzed and rewritten as parameterized C++ based

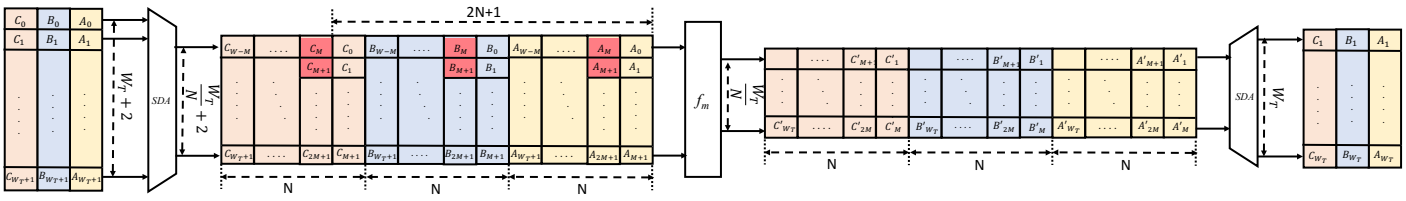


Fig. 8: Window2Pixel kernel

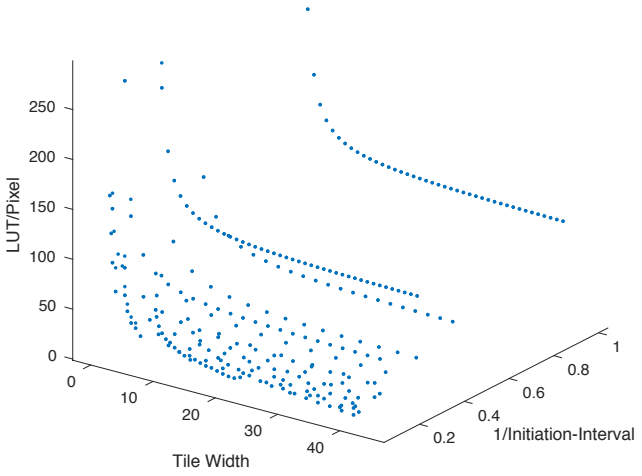


Fig. 9: Area, throughput and tile-width correlation for $Gaussian3x3$ kernel

kernels with stream-in/stream-out arguments. Using our tool, a library of different implementations for each kernel is generated. To examine whether our tool can cover different architectures, we evaluated it with VPR, a part of the academic Verilog-To-Routing project [17]. Using VPR, different size FPGAs were generated based on Altera Stratix IV, with logic cluster size $N = 10$, look-up table size $K = 6$, channel segment length $L = 4$. Then we passed each FPGA's size as an area budget to our tool. Figure 13 shows the percentage of LUTs used for implementing *Sobel* on different device sizes. As shown, our tool was able to automatically find suitable implementations for different architecture targets and fill over 95% of the chip area on average. Further, as shown in Figure 14 our tool fills the FPGA area while improving the throughput. The dots in the Figure show the LUTs per throughput results for each FPGA size. Considering results shown in this figures, we can see that the tool covers different area constraints while meeting the expected throughput for them. We also evaluated our tool with different Xilinx FPGAs. Figure 15 shows the percentage of LUTs used for implementing *Sobel* and *Harris* benchmarks on different Xilinx 7 series devices. As shown, our tool was able to automatically find suitable implementations for different architecture targets and fill over 97% of the chip area on average.

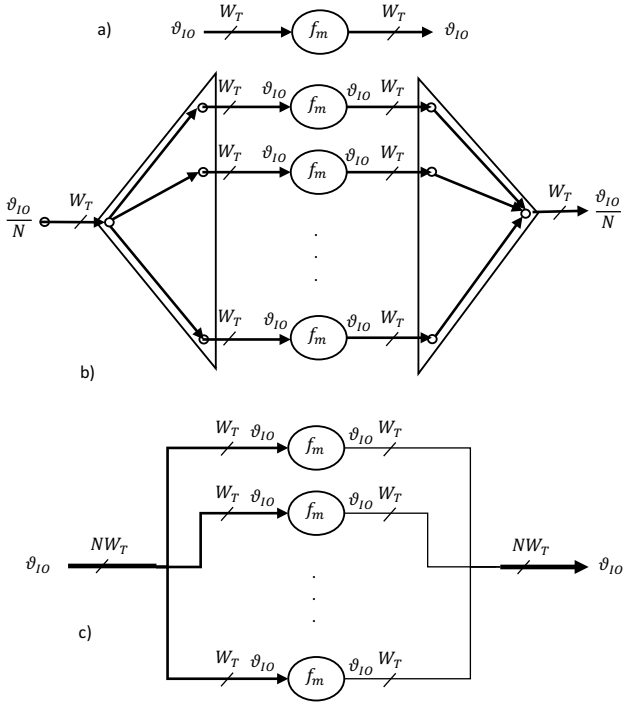


Fig. 10: Pixel2Pixel replication

Finding an optimum implementation for different throughput targets was the second goal of this study. To evaluate that, we tested our tool by setting different throughput targets for different benchmarks and compared our tool to a fully manual HLS approach. Since we implemented heavily parameterized kernels for our OpenVX approach first, we learned which parallelization strategies worked better. Using that knowledge, we generated a manual HLS version. Due to limited time (as all designers will experience), we had to choose just a handful of implementation strategies which gave similar optimal area and throughput as our tool. However, to achieve designs with throughputs in between the optimal points, these implementations were scaled in the most logical way possible and they became less efficient; they used more area as we moved further away from the optimal design points. Figure 16 shows how our tool covers a large design space and hits all targets more efficiently for the $vxMagnitude$ kernel. We compared our tool results for different CV kernels with our manual HLS and observed that our approach found better implementations for different throughput targets that saves up

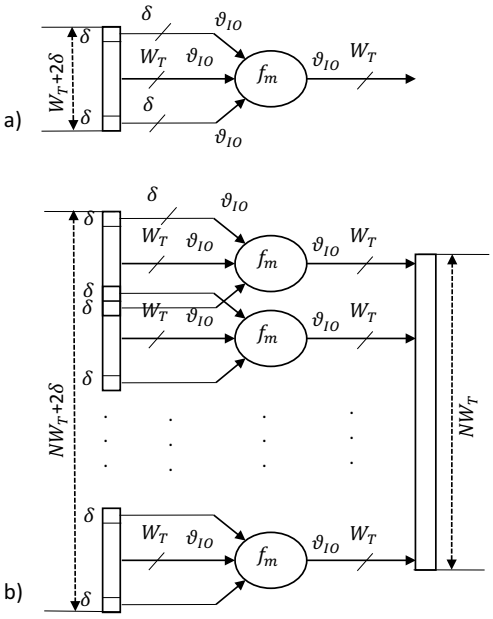


Fig. 11: Window2Pixel replication

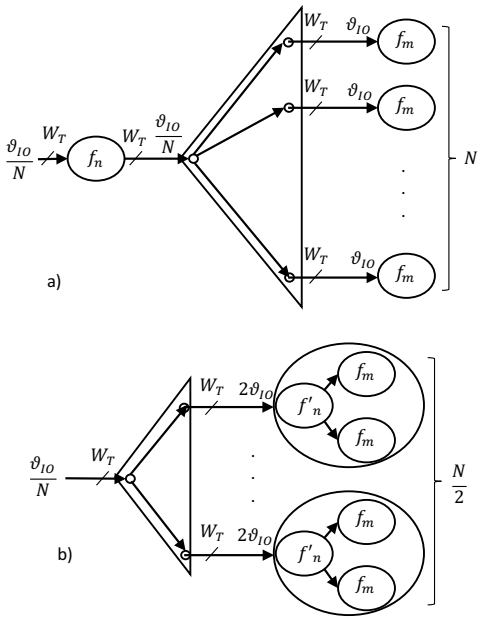


Fig. 12: Node combining

to 30% area.

Figure 17 shows area per throughout, normalized by the median value, for a range of throughput targets. As shown, for throughput targets larger than 5 Pixel/clk, our tool finds good area/throughput tradeoffs for each throughput target. For throughput targets less than 5, for some benchmarks such as $Guassian\_Filter$, the line buffer and SDA overhead became a big portion of the area cost and increased the area per throughput ratio.

To solve the trade-off finding problem, we used ILP and a heuristic approaches. Figure 18 shows the percentage of area saved by the heuristic approach compared to the ILP

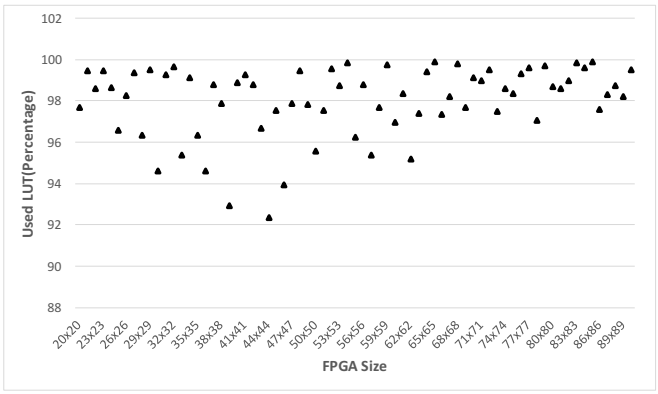


Fig. 13: LUT usage percentage for Sobel implementations on different VPR architectures

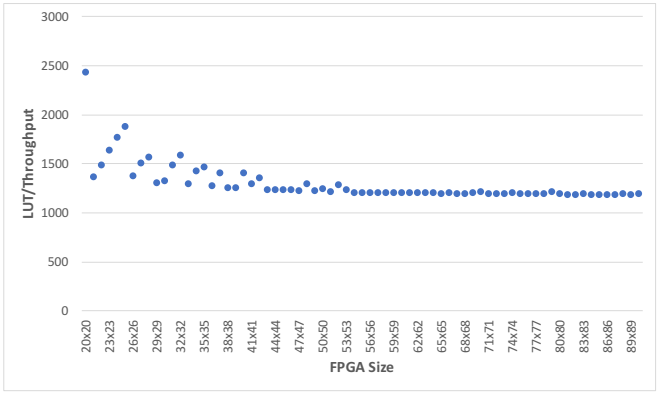


Fig. 14: Throughput achieved for Sobel on different FPGA size

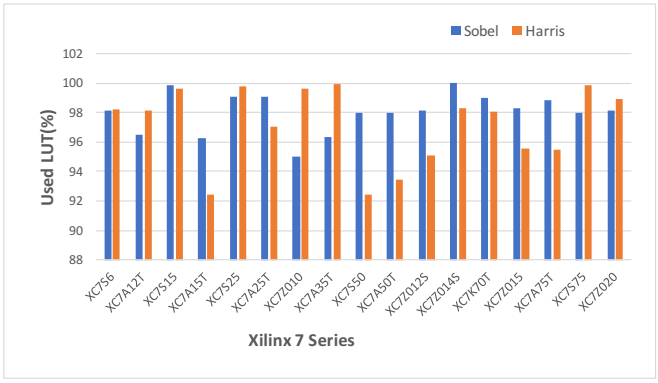


Fig. 15: Percentage of LUT usage for different Xilinx FPGAs

approach for implementing *Sobel* on different Xilinx FPGAs. The heuristic approach saves 19% area on average while decreasing the throughput only by less than 2%. The Inter-node Optimizer step in the heuristic approach is the key to the area saving, a step which is not possible to add to the ILP approach.

IV. Conclusion

In this paper, we studied the problem of automatically finding area/throughput trade-off of CV applications using OpenVX compute graphs implemented as hardware accelerators and mapped onto FPGAs. We proposed a framework on top of the Xilinx Vivado HLS tool which receives C++

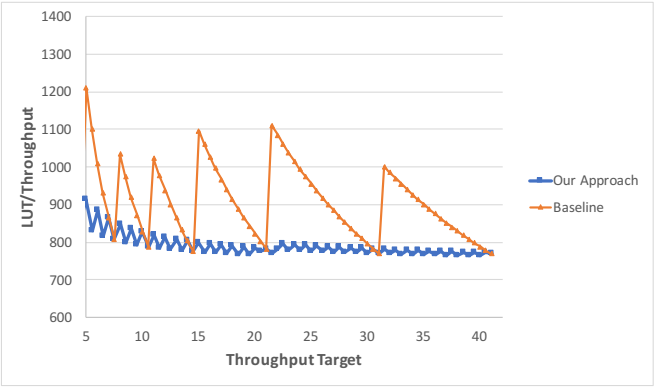


Fig. 16: $vxMagnitude$ Area/Throughput results for different throughput targets

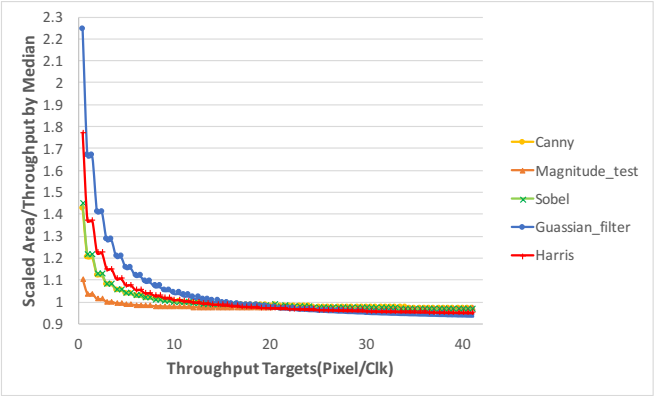


Fig. 17: Area cost results for different throughput targets

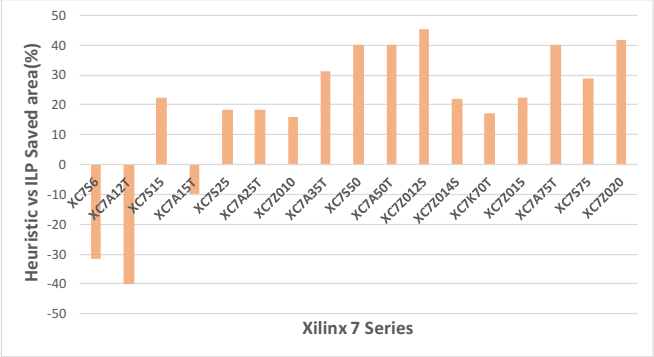


Fig. 18: Area saved percentage for Heuristic compared to ILP for Sobel benchmark due to Inter-node Optimizer step

based CV kernels and uses different approaches in order to find different implementations for each kernel. Moreover it compiles an OpenVX compute graph, analyzes it and finds a good trade-off between area and throughput. Our approach is differentiated from the existing approaches as 1) it automatically investigates finding different implementations, and 2) it combines module selection and replication methods as well as changing tile-size with node combining and splitting in order to automatically find a better area/throughput tradeoff. This approach was verified with different OpenVX benchmarks targeting several different FPGA sizes. Our tool is able to automatically achieve over 95% of the target area budget while improving the throughput. Our tool also can automatically satisfy a variety of throughput targets while minimizing the area cost. The proposed system saves up to 30% of the area cost compared to manually written and heavily parallelized implementations. Using Inter-node Optimizer step, our heuristic tradeoff finder is able to hit the same throughput targets while saving 19% area on average compared to existing ILP approaches.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 365–376.

[2] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 37–47.

[3] J. Cong, M. Huang, D. Wu, and C. H. Yu, "Invited - heterogeneous datacenters: Options and opportunities," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 16:1–16:6.

[4] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, pp. 10–22, 2015.

[5] Khronos-Group, "Portable, power-efficient vision processing," 2017. [Online]. Available: https://www.khronos.org/openvx/

[6] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97, 1997.

[7] V. Sarkar and R. Thekkath, "A general framework for iteration-reordering loop transformations," *SIGPLAN Not.*, pp. 175–187, 1992.

[8] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou, "Combining module selection and replication for throughput-driven streaming programs," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 1018–1023.

[9] H. Omidian and G. G. F. Lemieux, "Automated space/time scaling of streaming task graph," *International Workshop on Overlay Architectures for FPGA (OLAF)*, vol. abs/1606.03717, 2016.

[10] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "Adrenaline: An openvx environment to optimize embedded vision applications on many-core accelerators," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, 2015, pp. 289–296.

[11] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, pp. 144:1–144:11, 2014.

[12] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: Flexible multi-rate image processing hardware," *ACM Trans. Graph.*, pp. 85:1–85:11, 2016.

[13] Xilinx-inc, "Axi4 stream interconnect," 2017. [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi4-stream_interconnect.html

[14] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of multiple loops on fpgas using high level synthesis," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014, pp. 456–463.

[15] D. Liu and B. C. Schafer, "Efficient and reliable high-level synthesis design space explorer for fpgas," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–8.

[16] G. project, "Gnu linear programming kit," 2017. [Online]. Available: https://www.gnu.org/software/glpk/

[17] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "Vtr 7.0: Next generation architecture and cad system for fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, pp. 6:1–6:30, 2014.