Energy-Efficient Acceleration of OpenCV Saliency Computation using Soft Vector Processors

Gopalakrishna Hegde
School of Computer Engineering
Nanyang Technological University
Singapore, 639798
hegd0001@e.ntu.edu.sg

Nachiket Kapre
School of Computer Engineering
Nanyang Technological University
Singapore, 639798
nachiket@ieee.org

***Abstract*—Soft vector processors in embedded FPGA platforms such as the VectorBlox MXP engine can match the performance and exceed the energy-efficiency of commercial off-the-shelf embedded SoCs with SIMD or GPU accelerators for OpenCV applications such as Saliency detection. We are also able to beat spatial hardware designs built from high-level synthesis while requiring significantly lower programming effort. These improvements are possible through careful scheduling of DMA operations to the vector engine, extensive use of line-buffering to enhance data reuse on the FPGA and limited use of scalar fallback for non-vectorizable code. The driving principle is to keep data and computation on the FPGA for as long as possible to exploit parallelism, data locality and lower the energy requirements of communication. Using our approach, we outperform all platforms in our architecture comparison while needing less energy. At 640×480 image resolution, our implementation of MXP soft vector processor on the Xilinx Zedboard exceeds the performance of the Jetson TK1-GPU by 1.5× while needing 1.6× less energy, Beaglebone Black by 4.7× at 2.3× less energy, Raspberry Pi by 9× at 4× less energy, and Intel Galileo by 28× at 16× less energy. Our vector implementation also outperforms Vivado HLS generated OpenCV library implementation by 1.5×.**

I. Introduction

The availability of cheap embedded hardware and cheap cameras are enabling a range of innovative applications in embedded vision. Novel implementation domains such as remote monitoring with drones, industrial vision, and home automation along with medical image processing are a burgeoning market for these platforms where cost, power (energy) and performance are simultaneously critical. Commercial off-the-shelf embedded platforms are often based on ARM SoCs but now ship with high-performance NEON SIMD engines and CUDA-programmable GPU accelerators that can, in principle, handle the challenging demands of vision processing. Computations in computer vision are characterized by data-parallel operations on image pixels that naturally fit the SIMD organization of the NEON engines as well as the SIMT style of GPUs. However, in practice, the limited DRAM bandwidths, fixed ISA-based CPU organizations, limited cache capacities and inefficiencies in accelerators prevent these systems from achieving their raw potential. When latency and power considerations are strict, FPGA-based heterogeneous SoCs based on Zynq are an attractive alternative. However, despite performance and power advantages, FPGA programming is beyond the capability of most embedded developers. To make these systems truly viable to the broader embedded developer community, we need to lower the development cost of spatial hardware designs.

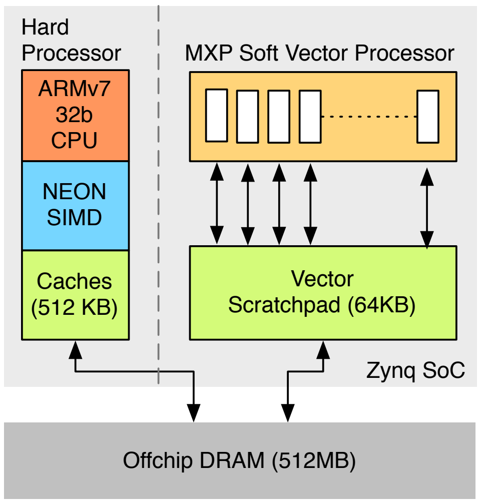

Fig. 1: High-Level Block Diagram of the Zynq SoC with the VectorBlox MXP [9] Soft Vector Processor

Recent attention to high-level synthesis (HLS) from Xilinx (Vivado HLS, SDAccel) and Altera (OpenCL) suggests a way for FPGA-based designs to bridge the productivity gap with competing hardware platforms. Vivado HLS flow even supports OpenCV library [2], [12] (an open-source computer vision library) for simpler high-level specification of image processing pipelines. However, this veneer of simplicity masks the underlying system-level design challenges such as design partitioning, hardware reuse, and data movement. The developer must manually select the buffering strategies to optimize data movement subject to FPGA on-chip memory constraints and consider limited FPGA logic capacity to carefully partition the design for maximum performance. The long FPGA simulation and compilation time slows down the classic edit-compile-debug design cycle, further discouraging prospective developers. Additionally, when newer, more capable FPGAs become available, the system-level design process needs to be repeated all over again. While this is still preferred to low-level Verilog-based flows, a productivity gap nevertheless exists over embedded developers targeting SIMD and GPU-based platforms.

Soft vector processors such as the VectorBlox MXP [9] (shown in Figure 1) are a superior alternative to high-level synthesis for OpenCV-based image processing computations. Similar to NEON and GPU hardware, OpenCV pixel operations are a natural match to the data-parallelism available on soft vector processors. Unlike NEON and GPU accelerators, soft vector processors can be tailored to economically support needed operations [6]. Similar to high-level synthesis, the soft vector processors can be programmed directly in C/C++ albeit using vector APIs. Unlike high-level synthesis, vector code written once can run with little modification on other supported FPGA boards. While performance tuning and memory optimizations are still needed, the degree of effort required is substantially lower as all optimizations are possible using conventional software programming flows.

The key contributions of this paper include:

- Development and functional verification of MXP vector code for Saliency detection while including optimizations for DMA memory transfers, line-buffering and compute optimizations.
- Quantification of performance and power usage of the MXP soft vector processor against a range of commercial off-the-shelf SoC platforms for various image resolutions.
- Comparison of HLS-generated OpenCV spatial hardware against MXP soft vector processor performance on the Zedboard platform.

II. Background

A. Saliency Detection

In this paper, we use Saliency detection as a case study to compare and contrast different SoC architectures and programming flows using high-level synthesis and vector processing. Visual Saliency detection is a bottom-up model for visual attention introduced in [5]. It is used to mimic the neuromorphic pathways in humans and other primates to help direct the focus of attention and understand complex scenes in the natural environment. It is a computationally demanding, but highly parallelizable application that struggles to achieve real-time execution on modern hardware. Broadly, Saliency detection operates on an input image in three compute pipelines as shown in Figure 2 – (1) intensity, (2) color, and (3) orientation, before a final saliency map is constructed. Within each pipeline, the image is processed by a pyramid filter, a center surround operation followed by a normalization. A Pyramid Filter is a multi-level multi-resolution operation that takes an input image, runs a 2D filter on it, subsamples the resulting image and repeats this process several times (5 levels in our case). A Center Surround operation combines two input images at different resolutions, upsamples the lower resolution image to equalize resolution of both inputs and performs an absolute difference on these images. Normalization adjusts all pixel values in the image by scaling them with a factor computed from the largest pixel value

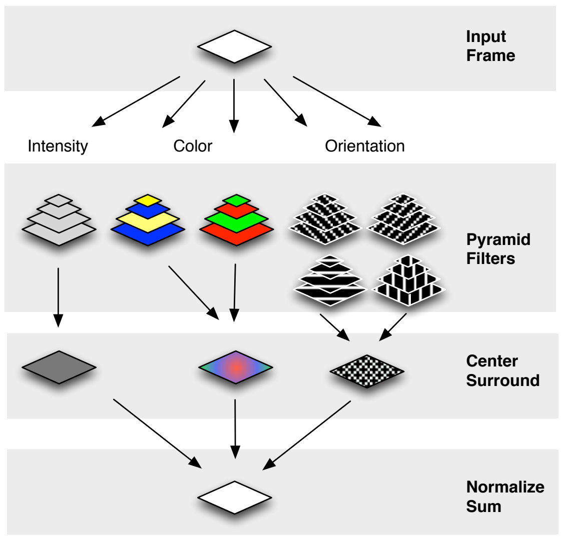

Fig. 2: Saliency Processing Compute Stack (One computation consists of 50 `FILTER2D`, 70 `RESIZE`, 64 `ADDWEIGHTED` and 5 `NORMALIZE` calls)

in that image. For obtaining the saliency map, we need to compute a single intensity map, two color maps (Red-Green and Blue-Yellow) and four orientation maps (four angles). This application is a composition of multiple OpenCV functions such as `FILTER2D`, `RESIZE`, `NORMALIZE`, `ABSDIFF` and `ADDWEIGHTED`.

B. Zynq Embedded SoCs

We choose the Zynq-based Zedboard platform for our experiments since we target applicability in the embedded vision domain. The Zynq Z7020 SoC on the Zedboard contains a dual-core 667 MHz ARMv7 32b CPU with a NEON SIMD unit capable of 8-wide processing of 8b data along with a 53.2 K LUT, 220 DSP-block, 560 kB BlockRAM FPGA connected over AXI and ACP ports rated at 800 MB/s–2 GB/s. The system-level power consumption of the Zedboard under various configurations lies between 4–6 W. Hardware accelerators can be integrated with the host ARM CPU through streaming communication channels mapped to the AXI or ACP ports. In particular, for OpenCV applications, Xilinx provides an AXI-VDMA engine which we use to move data rapidly to/from the host DRAM to FPGA logic.

C. MXP Soft Vector Processor

The MXP soft processor, shown earlier in Figure 1, is a multi-vendor, multi-platform configurable vector overlay architecture inheriting from a long line of vector FCCM designs stretching back to [11]. For the Zedboard, we can accommodate a 100 MHz 16-lane 32b (or 64-lane 8b) MXP implementation with a 64 KB scratchpad that interfaces with the host ARM CPUs over the 880 MB/s AXI ports. MXP runs a custom vector ISA and comes with an assembler that can convert vector APIs written in C into appropriate low-level

	Zedboard	Jetson TK1	Intel Galileo	Raspberry Pi	Beaglebone Black
Technology	28nm	28nm	32nm	40nm	45nm
SoC	Xilinx	NVIDIA	Intel	Broadcom	TI
	Zynq 7020	Tegra K1	Quark X1000	BCM2835	AM3359
Processor	ARMv7	ARMv7	i586	ARMv6	ARMv7
Accelerator	NEON, FPGA	NEON, CUDA cores	Nil	Nil	NEON
Clock Freq.	667 MHz CPU	2.3 GHz CPU	400 MHz	700 MHz	1 GHz
	250 MHz FPGA	852 MHz GPU			
On-chip	32 KB L1	32 KB L1	16 KB L1	16 KB L1	32 KB L1
Memory	512 KB L2	2048 KB L2		128 KB L2	256 KB L2
	560 KB FPGA	64 KB L1 + CSM		-	
Off-chip	512 MB	2048 MB	256 MB	512 MB	512 MB
Memory	32b DDR3-1066	64b DDR3-933	32b DDR3-800	32b DDR2-400	16b DDR3-606
Steady-state Power	5.1 W	3.6 W	3.6 W	2.1 W	2 W

TABLE I: Datasheet specifications of various COTS embedded SoC platforms

vector instrinsics. The scratchpad memory is banked for fast concurrent access to the various lanes and is fully managed by software through the vector DMA API. For optimum performance on image data, line buffers can be copied to the scratchpad using efficient image row-sized DMA transfers over the AXI ports. The scratchpad can be additionally tuned by the programmer to support double-buffered DMA transfers to hide memory loading costs. Within the context of chained OpenCV evaluation, it is further possible to carefully pipeline multiple function calls while keeping data resident in the scratchpad. Additionally, as most OpenCV operations are on 8b data items (few are 16b), the type specified in the instruction is decoded by the soft processor to automatically switch from 16-lane 32b design to 32-lane 16b design or 64-lane 8b design for higher throughput pixel operations.

D. *Embedded Platforms*

Ultimately we are interested in choosing the most power efficient platform for implementing embedded vision computations. To support this quest, we compare performance and power measurements across a set of low-power COTS embedded systems such as the Jetson TK1, Beaglebone Black, and the Raspberry Pi boards. In particular, we are interested in comparing the performance and power efficiency of SIMD and GPU acceleration wherever they are available. We tabulate the raw operating specifications and micro-benchmarking results of these platforms in Table I. From these raw specifications, the faster 4-core ARMv7 CPU on the Jetson TK1 along with its 192 CUDA cores and faster DRAM interface makes it a formidable entry into this architecture comparison. While aimed at applications with floating-point performance needs, the Jetson TK1 ships with a GPU-optimized OpenCV library that we use in our comparison. The Intel Galileo with its older-generation CPU and small caches, low operating frequency appears weakest in our set. The Beaglebone Black and the Raspberry Pi and both mid-range platforms that have been previously deployed in embedded computer vision applications.

III. OpenCV Implementation on FPGAs

In this section, we outline the computational requirements of the Saliency stack, identify opportunities for parallelism and describe our HLS and MXP programming methodologies.

A. *Compute Requirements*

Embedded vision computations typically operate on a stream of image frames delivered to the processor from the camera sensor. Saliency computations are notoriously challenging due to the sheer complexity of the underlying computations. A first-cut implementation of Saliency detection running on the ARMv7 CPU without NEON optimizations on 640×480 VGA frame requires 5.3s to process a single image (≈300M 8b integer operations) while consuming 4.6 W of power on the Xilinx Zedboard. A NEON optimized implementation of the same code shows a substantial performance improvement over vanilla CPU version to deliver a compute time of 0.49s (10× saving) at 5.7 W. This is not unexpected as bulk of the calculations are arithmetic operations in the 2D filter resulting in high SIMD potential. In Table II, we list the arithmetic and memory complexity of the various image processing tasks required to assemble a complete Saliency stack. While `FILTER2D` has large arithmetic intensity (several multiply-accumulate operations per memory access), `NORMALIZE` and `ABSDIFF` are memory bound, while `SUBSAMPLE` and `INTERPOLATE` have irregular, staggered memory accesses. In the final column of Table II, we show the relative distribution of runtime for the various OpenCV functions running on ARMv7 CPU with NEON acceleration.

OpenCV	**Arithmetic Operations**			**Memory Operations**		**Time**
Function	+	×	/	Rd	Wr	(ms)
`FILTER2D`	$8{\cdot}W{\cdot}H$	$9{\cdot}W{\cdot}H$	$W{\cdot}H$	$9{\cdot}W{\cdot}H$	$W{\cdot}H$	18.2
`SUBSAMPLE`	$2{\cdot}W{\cdot}H$	$\frac{9}{4}{\cdot}W{\cdot}H$	$\frac{1}{4}{\cdot}W{\cdot}H$	$\frac{5}{2}{\cdot}W{\cdot}H$	$\frac{1}{2}{\cdot}W{\cdot}H$	4.8
`INTERPOLATE`	$8{\cdot}W{\cdot}H$	$9{\cdot}W{\cdot}H$	$W{\cdot}H$	$9\frac{1}{4}{\cdot}W{\cdot}H$	$2{\cdot}W{\cdot}H$	18.7
`ABSDIFF`	$W{\cdot}H$	0	0	$2{\cdot}W{\cdot}H$	$W{\cdot}H$	1.6
`ADDWEIGHTED`	$W{\cdot}H$	0	0	$2{\cdot}W{\cdot}H$	$W{\cdot}H$	1.6
`NORMALIZE`	0	$W{\cdot}H$	$W{\cdot}H$	$2{\cdot}W{\cdot}H$	$W{\cdot}H$	1.4

TABLE II: Arithmetic and memory complexity of leaf-level OpenCV functions for W×H *input* image resolution along with NEON runtime for 640×480 resolution images

```
void cpu_filter2D(uint8_t *in, uint8_t *out)
{
  // loop over row/col
  for (int row = 0; row < M; row++) {
    for (int col = 0; col < N; col++) {
      int sop = 0;
      // loop over 3x3 filter
      for (int krow = 0; krow < 3; krow ++) {
        for (int kcol = 0; kcol < 3; kcol++) {
          sop += kernel[krow][kcol] *
                 in[row+krow][col+kcol];
        }
      }
      out[row][col] = sop/scale;
    }
  }
}
```

Fig. 3: C implementation of 2D filter

B. Optimizing Soft Vector Performance

We first show a scalar representation of a 2D filter code snippet in Figure 3 with doubly-nested for loops to process each pixel and an additional doubly-nested for loops within to process each kernel coefficient. The vectorized MXP version of this code using the vector APIs is shown in Figure 4. You can observe that the outer loop over rows is still retained and controlled from the scalar CPU core, while the inner loop over columns has been vectorized. Additionally, we also unroll the kernel loops into separate vector API calls. This separation allows a straightforward decoupling of memory and compute operations, and encourages overlapping of compute and DMA transfers. Note that it is possible to use an alternative form of vectorization where the internal 2D kernel operation is vectorized, but we observed faster performance when using the row-level parallelization strategy shown in Figure 4.

Based on this example, we can generalize the MXP programming strategy as consisting of three key components which we illustrate in Figure 5:

- The scalar ARMv7 CPU allocates the input image buffer in the memory for transfer to the MXP.
- The CPU then sends DMA API commands to the MXP DMA engine to load the scratchpad with appropriate data.
- Finally, the vector instructions are sent to the MXP decoder based on the vector API calls used in CPU code.

Using this flow, a straight-forward functionally-correct MXP implementation of saliency can be easily developed. However, there are opportunities for optimization that can improve performance by as much as 40–50%. The choice of buffering strategy is crucial to enable high-performance composition of multiple OpenCV functions in a sequence. This fact has long been recognized when designing high-performance streaming spatial hardware and is not unique to vision applications. However, choosing which mix of buffering strategies to use and to what extent is still tricky. While we perform the obvious double-buffering of inputs and outputs to overlap vector computation with DMA memory transfers to/from the DRAM, largest wins come from line buffering [13].

```
void vbx_filter2D(uint8_t *in, uint8_t *out)
{
  // Allocate scratch pad memory DMA not shown
  uint8_t top = vbx_sp_malloc(N);
  uint8_t mid = vbx_sp_malloc(N);
  uint8_t bot = vbx_sp_malloc(N);
  // Allocated 9 buffers for 3x3 temp results
  // top_11, top_12, top_13
  // top_21, top_22, top_23
  // top_31, top_32, top_33

  for (row = 1; row < M; row++) // row loop
  {
    // load new row
    vbx_dma_to_vector(bot, in+N*(row+2), N);

    // multiply rows in unrolled fashion
    vbx(SVBHU, VSHL, top_11, k_11, top);
    vbx(SVBHU, VSHL, top_12, k_12, top);
    vbx(SVBHU, VSHL, top_13, k_13, top);
    // repeat for mid and bot rows

    // add rows in unrolled fashion
    vbx(SVBHU, VSHL, top_11, k_11, top);
    vbx(VVHU, VADD, top_sum, top_11, top_12+1);
    vbx(VVHU, VADD, top_sum, top_sum, top_13+2);
    // repeat for other rows

    // scale result and copy row back
    vbx_dma_to_host(out+(N*row), out, N);

    // update pointers
    temp=top; top=mid; mid=bot; bot=temp;
  }
  vbx_sync();
}
```

Fig. 4: MXP implementation of 2D filter (significantly simplified)

Line Buffering From the perspective of performance and power constrains, keeping data in on-chip memories for as long as possible is important for performance as well as power. However, when processing large image frames on limited hardware, this may not always be possible. This leads to two strategies (1) limited use of line buffering in on-chip BRAM wherever possible, and (2) relegation to frame buffering in off-chip DRAM wherever necessary, for composing OpenCV compute pipelines in hardware. We show an example OpenCV function sequence $f \rightarrow g$ in Figure 6 such that OpenCV compute time is dwarfed by DMA transfer time. In this situation, keeping data on-chip is great for performance as it permits f and g to be scheduled back-to-back. Line-buffering enables overlapped evaluation of downstream OpenCV functions that can directly consume data from the scratchpad thereby avoiding a round-trip to the DRAM. Even if OpenCV compute times are longer than DMA times, avoiding DRAM access is good for lowering IO power.

As an example, consider the PYRAMID FILTER calculation which contains an alternating series of FILTER2D and SUBSAMPLE stages (up to 5 levels deep in our implementation). When implemented in spatial hardware, these can be implemented in dataflow streaming style where both functions operate concurrently. For soft vector processors, this

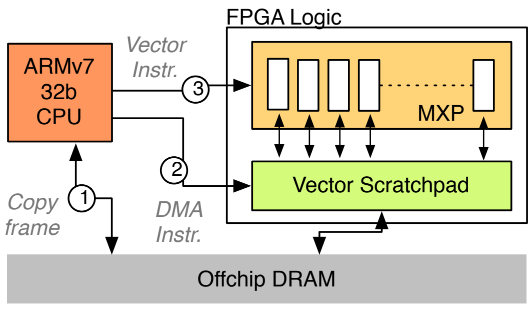

Fig. 5: MXP Flow showing different stages of execution

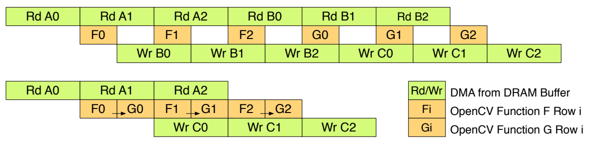

Fig. 6: Comparing Line-Buffering (bottom) and Frame-Buffering (top)

opportunity allows us to *chain* the OpenCV functions at the granularity of rows (or lines) instead of entire frames by scheduling consecutive row operations from the dependent OpenCV functions together. While this form of parallelism is natural for image processing pipelines, we must analyze function dependencies and memory access patterns across function boundaries to enable this global optimization. The careful reader may observe that this cannot be applied throughout the compute pipeline – the Normalize operator breaks the link by requiring a complete frame buffer accumulation before applying a suitable scaling factor to all pixels. Additionally, the 64KB on-chip scratchpad capacity further limits the number of rows that can be buffered before we must spill out into the DRAMs. Line buffering is a critical optimization that helps MXP performance approach that of fully spatial hardware designs with streaming on-chip buffers.

Beyond buffering, there are various computation-related optimizations necessary for efficiency:

- **Variable-Resolution Pipelining** Saliency is characterized by variations in image resolutions through the execution flow due to `SUBSAMPLE` and `INTERPOLATE` operations. This means there is often a mismatch between input and output data rates to the vector blocks resulting in a potential for wasted cycles. A naïve approach would offload the data rearrangement functions back to the CPU, but that would result in needless round-trip DMA transfers of data. Instead, we use type conversion vector operations to repack 8b data into 16b boundaries. While this is admittedly a slight waste of parallel lanes, (1) this avoids DMA overheads, by rearranging data in-place, and (2) keeps the hardware permutation fabric simple by avoiding the need for an expensive interconnect between the vector lanes that can reduce operating frequency.
- **Dead-pixel Removal**: If we analyze certain OpenCV function sequences, we realize that in some instances pixel results are unused by the downstream operators. These pixels are invisible to the rest of the compute stages and can be considered *dead pixels*. We can safely remove the computation associated with these pixels without affecting correctness. An example of this is the `FILTER2D` and `SUBSAMPLE` sequence where odd rows and columns can be removed.

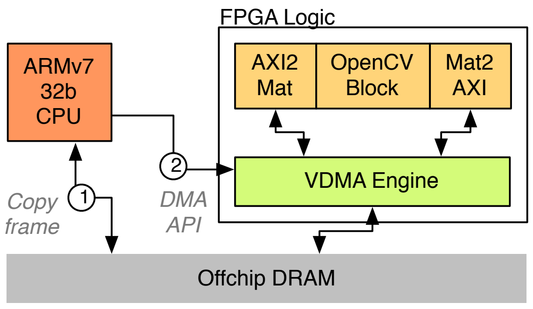

Fig. 7: HLS Operation

C. Optimizing High-Level Synthesis Performance

Traditional high-level synthesis tools compile high-level C/C++ code blocks, typically organized as for loops, into pipelined RTL pipelines with suitable streaming interfaces. While its possible to directly compile the C code shown in Figure 3 with synthesis directives and compiler hints, Xilinx provides an alternative method that directly uses an optimized OpenCV library. In Figure 8, we show the 2D filter being described using the Vivado HLS OpenCV library approach. The HLS compiler translates this description into pipelined OpenCV RTL datapaths from their internal pre-synthesized library. When integrating with the Zynq SoC, VDMA engines are needed for connecting the AXI streaming interfaces of the OpenCV datapaths to the DRAM where the frame buffers are allocated. Such a streaming design where *multiple* OpenCV datapaths are chained together with adequate internal buffering is the key source of performance benefits of the FPGA fabric.

```
void hls_filter2D(AXI_STREAM& in,
                  AXI_STREAM& out)
{
        GRAY_IMAGE in_img(ROWS, COLS);
        GRAY_IMAGE out_img(ROWS, COLS);
        // Convert in image stream to Mat
        hls::AXIvideo2Mat(in, in_img);
        hls::GaussianBlur<3,3>(in_img, out_img);
        hls::Mat2AXIvideo(out_img, out);
}
```

Fig. 8: HLS implementation of 2D filter

For spatial pipelines generated by high-level synthesis, a key consideration is whether the design will fit the intended FPGA platform. In our case, preliminary experiments for a 640×480 design suggests that, for the Zedboard, we may be able to barely fit an Intensity Map calculation on the FPGA fabric. A fully spatial implementation of the complete design is estimated to require ≈11 Zedboards worth of logic. For our application scenario, we are forced to perform logic reuse within the allocated resource constraints. HLS-based directives for resource sharing can help generate smaller footprints for

Platform	**OS**	**gcc g++**	**Total Power (W)** Active	Δ
Zedboard	Xillinux 1.3, MXP is bare metal	4.6.3	5.7 (NEON) 6.4 (MXP)	1.3
Jetson TK1	Ubuntu 14.04 LTS v3.10.24	4.8.2	6.5 (NEON) 6.6 (GPU)	3
Beaglebone	Ubuntu 14.04 LTS v3.8.13	4.8.2	3.2	1.2
Raspberry Pi	Raspbian Wheezy v3.12.28	4.6.3	2.9	0.8
Intel Galileo	Debian Wheezy v3.8.7	4.7.2	3.7	0.1

TABLE III: Platform Configurations across embedded platorms, Δ is additional power over steady state power reported in Table I.

the C/C++-based operators, but they sacrifice performance by keeping large portions of hardware inactive taking away the fundamental benefits of streaming spatial processing. In any case, these pragmas have no effect on the OpenCV Vivado HLS libraries as they come with pre-bundled synthesizable cores that do not expose these optimization hooks. Yet another acknowledged limitation of the OpenCV Vivado HLS design flow, at present, is the need to instantiate multiple VDMA blocks (or to multiplex the different streams carefully while avoiding deadlock) to allow frame-buffering of intermediate image frames when on-chip streaming implementation is not possible. This limitation is particularly severe in the Saliency compute stack as any partitioning of the flow shown in Figure 2 produces dozens of streams that must be buffered in the DRAM. Constrained by these limitations, we (1) instantiate one copy of each of the leaf-level OpenCV functions in hardware, (2) compose the larger application by frame-buffering through the off-chip DRAM, and (3) instantiating multiple VDMA engines for each image stream. We show a high-level picture of our composition flow in Figure 7 where the CPU helps load the input and intermediate images in the DRAM through low-level control of the VDMA engine.

IV. Methodology

The goal of our experiments is to perform an architecture comparison across various embedded SoC platforms and quantify the gap between MXP and HLS-based FPGA programming flows. For our OpenCV experiments, we install and configure OpenCV v3.0.0 alpha across all the embedded SoC platforms as it includes optimizations for NEON SIMD parallelism. The OpenCV library is multi-platform and contains various special cases and checks that often slow down performance especially for small problem sizes. We also write our own functionally equivalent but vanilla C implementations in cases where better performance can be achieved. We rigorously stress test the various platforms at multiple resolutions from 240×160 upto 1920×1080 to identify trends and isolate bandwidth bottlenecks. We compile Saliency OpenCV code across all CPU/GPU platforms with the −O3 switch that enables NEON vectorization. For the GPU implementation of Saliency, we rewrite our reference implementation with slight modifications to use `GpuMat` structures for holding the 2D images and invoking the GPU-accelerated versions of the OpenCV functions. For FPGA setup, we use the MXP

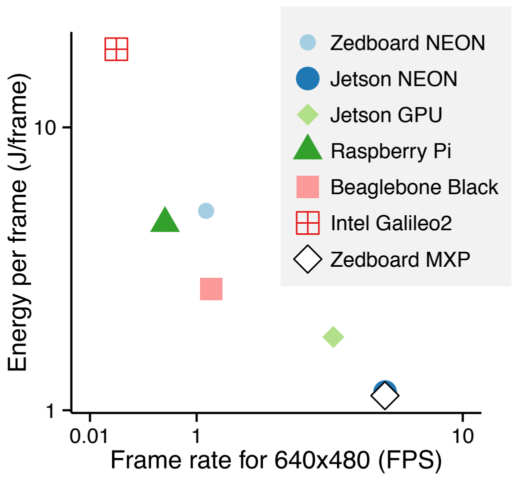

Fig. 9: Energy-FPS Trends in Embedded SoCs

soft vector processor release v0.0.6 (Xilinx Zedboard). When using high-level synthesis, we use Vivado HLS 2013.4 along with Vivado 2013.4 to run backend CAD. We execute MXP vector processor while running the ARM CPU in bare metal mode without any host operating system. For the ARMv7 experiments on the Zedboard, we use Xillinux-1.3 OS. We tabulate the OS and compiler versions along with power usage information across our experiments in Table III.

V. Results

In this section, we describe our experimental results across various embedded SoC platforms, provide a performance comparison of MXP and HLS hardware while finally showing the benefit of MXP tuning on overall MXP performance.

A. *Comparison of Embedded SoC Platforms*

We are primarily interested in uncovering the energy efficiency for the FPGA implementation when compared to other commodity SoCs. In Figure 9, we report the overall runtime vs. energy per input frame required by the various SoC platforms for an identical processing of a VGA 640×480 image. At this resolution, the MXP and Jetson (NEON) outperform all other systems requiring 176 ms of runtime (5.6fps) with the MXP needing a marginally lower energy requirement of 1.12 J over Jetson NEON's 1.16 J. However, the Jetson GPU implementation is not only slower, running at 263 ms (3.8fps), but also

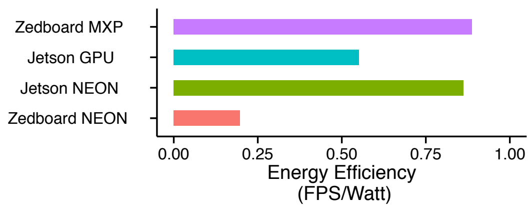

Fig. 10: Measuring FPS/Watt for Jetson and Zedboard

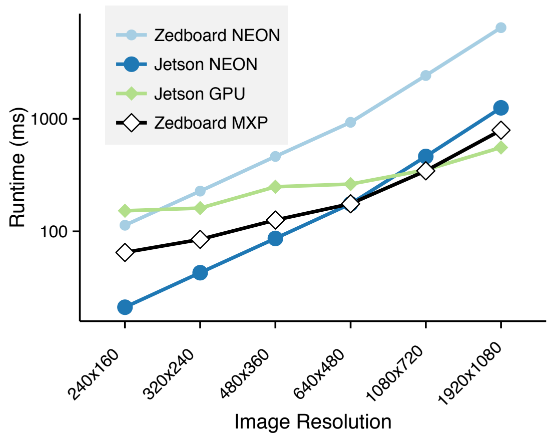

Fig. 11: Impact of Image Resolution on total compute time (Zedboard vs. Jetson TK1)

less energy efficient consuming 1.8 J of energy. The rest of the platforms are neither fast nor energy efficient compared to the Zedboard MXP and Jetson NEON/GPU implementations. We also show a breakdown total and dynamic power usage of the different platforms in Table III. We plot the FPS/Watt figures for the Jetson and Zedboard platforms in Figure 10 and observe that the MXP (FPGA) has better energy efficiency compared to all other platforms at ≈0.8 FPS/W.

We further quantify performance scaling trends at various image resolutions in Figure 11 for the high-performing Zedboard and Jetson TK1 platforms. At resolutions below 640×480, the 2.3 GHz Jetson ARMv7 CPU easily outperforms all other platforms by a wide margin. And at the largest 1920×1080 resolution, the Jetson 192-CUDA core GPU beats the MXP implementation by 40%. Across all resolutions, the MXP is either second best or the best running implementation (at 640×480 and 1080×720).

B. Optimizing MXP Performance

We consider various optimizations for improving the performance and efficiency of MXP-based processing as quantified in Figure 12 for the 640×480 case. We show constituent, leaf-level OpenCV functions as well as a select few composed higher-order OpenCV operations. Double buffering provides the first 10–12% improvements in performance for the `ADDWEIGHTED`, `NORMALIZE` and `FILTER2D` stages. Our initial implementation of `SUBSAMPLE`, and `INTERPOLATE` would fall back to the scalar CPU due to the complexity of data shuffling, but we rewrote the scalar code in a vector form that admittedly underutilized the vector lanes. However, this still allowed us to avoid the larger penalty of needless DMA traffic yielding a significant additional 20% reduction in runtime. When composing multiple higher-order OpenCV functions such as `PYRAMID FILTER`, and `CENTER SURROUND`, we are able to exploit data reuse and function overlap to reduce runtime by an additional 20–25%. Finally, for the complete Saliency computation, when combining all these optimizations together, we observe a 40% reduction (almost 2×) in runtime.

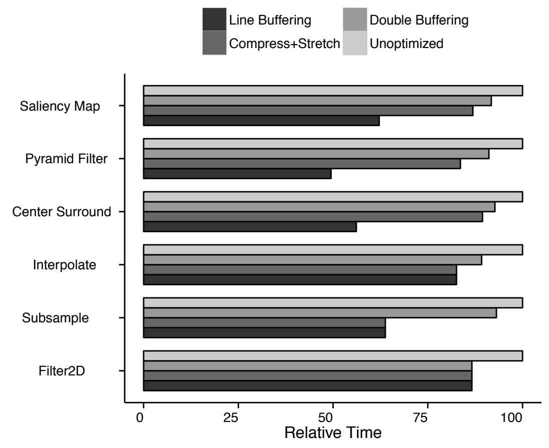

Fig. 12: Quantifying the impact of MXP Optimizations

OpenCV Function	**LUTs**	**FFs**	**DSPs**	**BRAMs** (18kB)	**Time (ms)** 640×480
FILTER2D	812	533	4	3	3.18
(% of FPGA)	1.5%	0.5%	1.8%	1%	
RESIZE	6K	6.8K	16	2	3.33[1]
	11.5%	6.5%	7.2%	0.7%	6.4[2]
NORMALIZE	2.2K	1.9K	4	3	3.12
	4.3%	1.8%	1.8%	1%	
ADDWEIGHTED	3.9K	2.6K	28	0	4.4
	7.5%	2.5%	12.7%	0%	
ABSDIFF	361	302	0	0	4.5
	0.7%	0.3%	0%	0%	
PYRAMID	26.3K	28.6K	68	11	8.9
FILTER	50%	27%	31%	4%	
CENTER	5.9K	6.9K	16	2	1.7–6.4[3]
SURROUND	11%	6.5%	7.2%	0.7%	
INTENSITY	72.6K	70.2K	232	21	36.8
MAP	136%	66%	105%	7.5%	
COLOR	100K	98.8K	300	32	87.5
MAP	189%	93%	136%	11.5%	
ORIENTATION	406K	331K	1.1K	104	140
MAP	763%	311%	525%	37%	
SALIENCY	578K	500K	1.6K	107	270
MAP	1088%	470%	766%	56%	

TABLE IV: OpenCV HLS Utilization and Performance (% values reflect utilization of the Zedboard capacity)
[1]subsampling [2]interpolation [3]depends on resolution

C. Comparison of MXP with High-Level Synthesis

Finally, we compare the performance of fully-optimized MXP implementation against an HLS-based approach that is forced to rely on frame-buffering to compose larger computations. Due to their frame-buffered implementation, overall frame processing time for a 640×480 frame is 270 ms for the HLS-based flow as compared to the 176 ms runtime of the line-buffered MXP design. This performance is 1.5× slower and ultimately limited by DRAM access bandwidth. We show a complete breakdown of resource utilization and runtime across constituent OpenCV functions in Table IV. Resource numbers are obtained from post place-and-route reports when design fits the chip, and post-synthesis estimates otherwise.

VI. Discussion

A. Related Work

FPGA accelerators for saliency detection have previously been demonstrated in [7] (Berkeley Calinx board) and [1] (ML605 board). In these designs, the complete compute stack fits on the FPGAs through manual time-multiplexing of stencil hardware with intermediate results stored on offchip DRAM when needed. The ML605 implementation runs at 4Mpixels/s for a modified implementation of Saliency while the Calinx version reports no results. In [3], the authors make a case for FPGAs in mobile vision processing applications using Haar detection as a case study. Oddly, they choose to compare a Google Nexus One with an FPGA implementation on a large Stratix IV GX EP4SGX530 chip (DE4-530 board) instead of using a lower-power Cyclone series. With the large chip, an array of SIMD units were used to implement the entire cascade of Haar classifiers, with each SIMD unit configured as a streaming hardware block. They were able to classify 320×240 images in 20 ms on the large FPGA which is 60× faster than the mobile SoC. An embedded implementation of vision processing tasks on the Zynq platform is presented in [4], where multiple pre-compiled Acadia Vision IP cores are mapped to the FPGA fabric and interface with the ARMv7 host via VDMA blocks. For certain vision tasks like contrast normalization, image stabilization and moving target indication, they report processing rates of 15fps for 640×480 resolutions on the ZC702 and ZC706 boards . IPPro [10] is a soft processor core optimized for image processing on the Zedboard platform and has been demonstrated to deliver 2.3fps for traffic sign detection algorithm on images of 600×400 resolution when using a 32 core design. Only color filtering and morphological operations ran on the soft cores with the rest running on the ARMv7 CPU. Our MXP implementation targets a far more complex vision task than the ones reported in [4], [10] on the Zedboard while running at 5fps for 640×480 images. An alternative to Vivado HLS OpenCV library is presented in [8], where few OpenCV functions such as Gaussian Filter and Sobel Filter are implemented as optimizable nested loops. While they deliver slightly faster performance compared to Vivado OpenCV, their key contribution is that they enable design space exploration by exposing optimization hooks (HLS directives) unlike the black-box approach taken by the Vivado OpenCV library. Compared to MXP implementations, these still need a full place-and-route flow.

B. Notes on Soft Vector Portability

A key advantage of the MXP is ease of design portability across platforms from the perspective of the embedded developer. We effortlessly setup the DE2-115 board with a 16-lane 100 MHz 64KB scratchpad design that ran as fast the the Zedboard NEON implementation. Similarly, we also recompiled exact same saliency vector code to target a 32-lane, 185 MHz, 128 KB scratchpad design on the DE4-230 to run about 3-4× faster. Despite all the advantages, there are several areas for improvement to the soft vector processor design for pixel processing. To efficiently support variations in resolution and data movement within the image frame, the vector lanes should support a configurable routing pattern. Another key constraint that limits the filtering operations is the absence of fused multiply-accumulation support (3 reads, 1 write).

VII. Conclusions

Soft-vector processors such as the MXP deliver a competitive energy-efficient solution for embedded vision applications such as Saliency detection. For moderate image resolutions such as 640×480 and 1280×1024, the Zedboard-MXP implementation is the fastest and the most energy-efficient solution compared to the Jetson TK1, Beaglebone Black, Raspberry Pi and the Intel Galileo platforms. We are able to deliver these improvements by careful scheduling of DMA operations and exploiting line-buffering techniques to enhance data reuse. In capacity-constrained systems for embedded visions application, we expect soft vector processors to offer a high-performance, low-energy solution for demanding applications.

References

[1] S. Bae, Y. C. P. Cho, S. Park, K. M. Irick, Y. Jin, and V. Narayanan. An FPGA implementation of information theoretic visual-saliency system and its optimization. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 41–48. IEEE, 2011.

[2] G. Bradski. The OpenCV Library. *Doctor Dobbs Journal*, 2000.

[3] B. Brousseau and J. Rose. An energy-efficient, fast FPGA hardware architecture for OpenCV-Compatible object detection. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 166–173, 2012.

[4] E. Gudis, P. Lu, D. Berends, K. Kaighn, G. van der Wal, G. Buchanan, S. Chai, and M. Piacentino. An embedded vision services framework for heterogeneous accelerators. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2013 IEEE Conference on*, pages 598–603, 2013.

[5] L. Itti and C. Koch. A model of saliency-based visual attention for rapid scene analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(11), Jan. 2002.

[6] S. J. Jie and N. Kapre. Comparing soft and hard vector processing in fpga-based embedded systems. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–7, 2014.

[7] N. Kapre, D. Walther, C. Koch, and A. DeHon. Saliency on a chip: a digital approach with an FPGA . *The Neuromorphic Engineer*, pages 1–2, Nov. 2004.

[8] M. Schmid, N. Apelt, F. Hannig, and J. Teich. An image processing library for C-based high-level synthesis. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4, 2014.

[9] A. Severance and G. G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–10. IEEE, 2013.

[10] F. M. Siddiqui, M. Russell, B. Bardak, R. Woods, and K. Rafferty. IPPro: FPGA based image processing processor. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6, 2014.

[11] M. Weinhardt and W. Luk. Pipeline vectorization for reconfigurable systems. In *Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. 7th Annual IEEE Symposium on*, pages 52–62, 1999.

[12] Xilinx, Inc. Xilinx XAPP1167 Accelerating OpenCV Applications withZynq-7000 All Programmable SoC usingVivado HLS Video Libraries. Technical report, Feb. 2009.

[13] H. Yu and M. Leeser. Automatic Sliding Window Operation Optimization for FPGA-Based Computing Boards. *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 76–88, 2006.