Benchmarking Vision Kernels and Neural Network Inference Accelerators on Embedded Platforms

Murad Qasaimeh[a], Kristof Denolf[b], Alireza Khodamoradi[c], Michaela Blott[b], Jack Lo[b], Lisa Halder[b], Kees Vissers[b], Joseph Zambreno[a], Phillip H. Jones[a]

[a]*Iowa State University, IA, USA*
[b]*Xilinx Research Labs, USA and Ireland*
[c]*UC San Diego, CA, USA*

Abstract

Developing efficient embedded vision applications requires exploring various algorithmic optimization trade-offs and a broad spectrum of hardware architecture choices. This makes navigating the solution space and finding the design points with optimal performance trade-offs a challenge for developers. To help provide a fair baseline comparison, we conducted comprehensive benchmarks of accuracy, run-time, and energy efficiency of a wide range of vision kernels and neural networks on multiple embedded platforms: ARM57 CPU, Nvidia Jetson TX2 GPU and Xilinx ZCU102 FPGA. Each platform utilizes their optimized libraries for vision kernels (OpenCV, VisionWorks and xfOpenCV) and neural networks (OpenCV DNN, TensorRT and Xilinx DPU). For vision kernels, our results show that the GPU achieves an energy/frame reduction ratio of 1.1–3.2× compared to the others for simple kernels. However, for more complicated kernels and complete vision pipelines, the FPGA outperforms the others with energy/frame reduction ratios of 1.2–22.3×. For neural networks [Inception-v2 and ResNet-50, ResNet-18, Mobilenet-v2 and SqueezeNet], it shows that the FPGA achieves a speed up of [2.5, 2.1, 2.6, 2.9 and 2.5]× and an EDP reduction ratio of [1.5, 1.1, 1.4, 2.4 and 1.7]× compared to the GPU FP16 implementations, respectively.

Keywords: Benchmarks, CPUs, GPUs, FPGAs, Embedded Vision, Neural Networks.

1. Introduction

Computer vision empowered with the recent advances in deep learning plays a fundamental role in solving many problems that seemed impossible just a decade ago. The computational complexity and memory footprint of these algorithms keep increasing to enhance accuracy or solve more complex problems [1]. This trend is driving the development of energy-efficient processing solutions, which are especially important for energy or thermal constrained real-time embedded systems. Often their limited communication power budget or communication capabilities preclude them from streaming images to more powerful computing entities.

Both industry and academia have explored the development of acceleration engines to help meet the needs of embedded vision applications. Three common types of such accelerators are benchmarked in this case study: multicore CPUs, Graphic Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). Each of these accelerators take a different approach to accelerating embedded vision applications. Multi-core CPUs make use of SIMD instruction extensions, such as: the ARM NEON SIMD engine, Intel's family of SSE, and dedicated vision processing units (VPU), such as Myriad [2]. The multi-threading programming model has made GPUs highly popular in this domain. GPUs provide massively parallel execution resources and high memory bandwidth. However, their high performance comes at the cost of high power dissipation [3]. FPGAs offer opportunities for using low-level fine-grained parallelism by customizing processing/control units and data paths to the requirements of a specific algorithm or application [4].

Embedded vision applications can exhibit vastly different performance characteristics depending on their underlying hardware accelerator platform and compute fabrics [5]. This varying behavior fundamentally stems from differences in accelerator micro architectures, middleware support, and programming styles. This mixture of factors makes choosing the best application-to-accelerator mapping a nontrivial task for embedded vision application developers. They must take into consideration metrics, such as expected runtime performance, energy-efficiency, and programmability. Moreover, running vision pipelines on heterogeneous platforms requires partitioning them into phases that can run on the available accelerators in the most efficient and cost-effective manner.

Beside the broad spectrum of hardware architectures choices for vision applications, there are various complexity-accuracy algorithmic trade-offs that need to be explored [6]. Quantization and pruning are two examples of optimization methods that are used to reduce the computational complexity and memory requirement of vision algorithms but at the expense of accuracy loss. Another design choice that need to be explored is deciding whether neural network solutions are more suitable choice compared to the traditional vision kernels for a specific vision application. The traditional (hand-engineered) algorithms are mature, proven, and optimized for performance and power efficiency, while neural networks (learned algorithms) offer greater accuracy and versatility, but demand large amounts of computing and resources. Moreover, the modeling capacity of traditional vision kernels are limited by the fixed transformations (filters) that stay the same for different sources of data. While learned features are data-driven and adapt based on the training data. Therefore, the complexity-accuracy trade-offs between neural networks solutions and traditional vision kernels need to be taken into consideration during the development process.

In order to clearly understand how different hardware architectures may impact the performance of vision kernels and neural networks, we analyze their performance on such accelerators. In this paper, we evaluate the performance of three commonly used HW accelerators for vision applications: the ARM Cortex A57 CPU, Jetson TX2 GPU, and ZCU102 FPGA in terms of accuracy, run-time performance (latency and throughput), and energy efficiency. For vision kernel benchmarking, we propose an easily reproducible approach that only uses publicly available vision libraries: OpenCV, Nvidia VisionWorks and xfOpenCV, without adding any special platform specific code. We also evaluate the performance of neural network inference implementation of [Inception-v2 and ResNet-50, ResNet-18, Mobilenet-v2 and SqueezeNet] using OpenCV DNN module, Nvidia TensorRT and Xilinx Vitis AI frameworks running on these accelerators. All benchmark code is available at: https://github.com/isu-rcl/cvBench.

Contributions. The main contributions of this paper are: (1) Benchmark representative vision kernels and complete pipelines using OpenCV, Visionworks and xfOpenCV libraries on the ARM57 CPU, Nvidia Jetson TX2 (GPU-accelerated) and Xilinx UltraScale (FPGA-accelerated), (2) Benchmark a set of five neural networks implementations using OpenCV DNN module, Nvidia TensorRT and Xilinx DPU, (3) Provide an insight into the reasons behind the observed run-time, power, and energy consumption performance for each evaluated platform and discuss rationales for why a given underlying hardware architecture innately performs well or poorly, and (4) Provide easily reproducible open-source benchmarking templates that only use publicly available vision libraries.

Organization. The remainder of this article is structured as follows. Section 2 reviews related work. Section 3 provides insights into the architectural differences between the hardware accelerators evaluated. It also provides details on the six categories of vision algorithms and two neural networks used in our study. Section 4 presents the performance metrics used in this study and provide a detailed description of our measurement methodology. Section 5 discusses our experimental results and observations. Finally, Section 6 concludes the paper with outlooks for future work.

2. Related Work

In this section, we take a look at existing benchmarking efforts in the literature that evaluate the performance of vision kernels and neural networks on embedded platforms. Even though most prior benchmarking efforts focus solely on comparing the performance of a limited number of vision kernels or cover only subsections of the embedded design space in evaluating neural networks performance, there are a few exceptions discussed in this section.

Vision kernels benchmarks. The comparison study in [7] analyzed the performance efficiency of FPGAs and GPUs on the GPU-friendly benchmark suite (Rodinia). They ported 15 of its kernels using Vivado HLS for the FPGA and OpenCL for host programs. The platforms used were a Xilinx Virtex-7 FPGA and Nvidia Tesla K40c GPU. Although this study included some vision kernels such as: GICOV, Dilate, SRAD and MGVF, it was not mainly focused on benchmarking vision algorithms; it included other kernels for data mining, fluid dynamic, and physics simulation, etc [8].

Other comparison studies each focused on a subset of vision kernels. For example, the study in [9] and [10] evaluated the performance of sliding window applications on FPGAs, GPUs and multi-core CPUs. They compared the performance of three applications: Sum of Absolute Differences (SAD), 2D convolution, and correntropy. The platforms used in their study were an Altera Stratix IV FPGA, an Nvidia GeForce GTX 560, and an Intel Xeon Core i7. Another study in [11] focused on comparing the performance of morphological image filtering operations. The authors utilized the OpenCV library for CPU and GPU (cv::CUDA module) implementations. For the FPGA platform, they used Vivado HLS video libraries and hand-optimized implementations. The platforms used in their study were the Xilinx Zynq 7020 FPGA, Nvidia Tegra K1, and Intel core i7. The work in [12] also focused only on a subset of vision operations such as normalized cross correlation and finite impulse response (FIR) filters. This study's evaluation included development time, component cost, and power consumption.

Neural networks benchmarks. There are two types of machine learning benchmarks based on the classification of [13]: (1) Machine learning (ML) benchmarks focus mainly on achieving high test accuracy, independent of the hardware implications. Examples of this kind of benchmarks are ILSVRC ImageNet competition [14] and MLBench [15], (2) Performance benchmarks focus on measuring performance metrics such as latency, throughput and power consumption. This category of benchmarks give algorithmic modifications freedom to reach the highest performance. Examples include DeepBench [16], SPEC [17] and STREAM [18]. A more complete benchmarking suite has been proposed in QuTiBench [13]. QuTiBench is a novel multi-tiered benchmarking methodology that supports algorithmic optimizations and couple hardware performance with accuracy at the application level. It includes test suites at 4 levels of abstraction: (1) level-0 includes roofline analysis that provides insight into the memory and compute requirements, (2) level-1 focuses on the achievable compute performance for different compute patterns, (3) level-2 captures potential bottlenecks in data movements, and (4) level-3 covers the system-level performance.

Our benchmark is exhaustive and energy-efficiency focused: we evaluate the accuracy, run-time, and energy consumption of different embedded hardware platforms over a wide range of standard vision kernels, vision pipelines and neural networks. The results are easily reproducible through the use of open-source benchmarking templates that only use publicly available vision and neural network libraries.

3. Background

In this section, we first present the characteristics of the hardware accelerators evaluated in this study. Then, we briefly discuss three vision libraries and neural network inference frameworks that are widely used with these accelerators. We group the vision kernels into categories based on their characteristics to understand the implications of the underlying hardware architectures on the performance of these kernels in their respective categories. Finally, we provide details on the used neural networks models and their architectures.

3.1. Embedded Platforms

The following are the three most common platforms used in embedded vision applications:

3.1.1. Central Processing Unit (CPU):

Modern CPUs are able to perform SIMD (Single Instruction, Multiple Data) instructions using multiple ALUs. Such processing scheme exploit data level parallelism; there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. These SIMD instruction sets are useful in the context of image processing, where operations are often repetitively applied to a continuous stream of data. This is particularly true in the context of computer vision, where most operations are performed over the entire image. Examples of SIMD architectures are: ARM NEON SIMD engine [19] and Intel's streaming SIMD extensions (SSE) [20].

3.1.2. Graphic Processing Unit (GPU):

As compared to general purpose CPUs, which have developed SIMD instruction extensions to help parallelize image processing type tasks, GPUs have taken the direction of evolving into a specialized SIMD architecture. This specialization has led to GPUs having simpler processing cores than high-performance general purpose CPUs. For example, they have simpler control logic, typically no branch prediction or prefetch, and small per-core memory. Simpler computing cores allow GPUs to pack many more cores into a chip than a general purpose CPU. GPU architectures perform extremely well on workloads that have little to no branching conditions or data dependences. Additionally, GPU architectures have specialized their memory architecture to support high-speed data streaming for image processing. For example, the L2 cache in the Jetson TX2 (Pascal GPU) is 2048 KB, which can fit a 1080p grayscale image.

3.1.3. Field Programmable Gate Array (FPGA):

Instead of having a fixed processor-like design, FPGAs consist of an array of logic blocks, DSPs, on-chip BRAMs, I/O pads, and routing channels. In FPGA, custom data paths can be architected to stream pixels directly between computing units without needing to read/write from/to external memory. Moreover, the distributed on-chip BRAMs can be used to exploit data locality in vision kernels by keeping pixels on-chip (e.g Zynq UltraScale MPSoC FPGA has 32.1 Mb on-chip memory). With FPGAs, developers need to ensure that their customized designs meet timing and space requirements.

3.2. Computer Vision Libraries

A number of vision libraries have been optimized to target the hardware platforms discussed in the previous section. In this work, we focused on the most complete and commonly used libraries, as follows:

3.2.1. OpenCV:

OpenCV (Open Source Computer Vision Library) is the defacto standard C/C++ library for image and vision processing [21]. It is used by the computer vision community to create desktop and embedded vision applications. It has more than 2500 optimized vision kernels, which includes a comprehensive set of both traditional and state-of-the-art vision and machine learning algorithms. OpenCV has bindings for languages such as Python and Java. The latest version of OpenCV (at the time of writing this paper) is 4.1.1.

3.2.2. NVIDIA VisionWorks:

VisionWorks is a toolkit for computer vision and image processing released by Nvidia in 2015 [22]. It implements and extends the OpenVX standard, and is optimized for CUDA-capable GPUs. VisionWorks provides three programming models: (1) immediate mode which enables developers to easily port their applications, (2) graph mode which enables advanced optimizations such as: buffer reuse, efficient use of streaming and CUDA textures, tiling and pipelining functions at sub-frame level, and (3) CUDA API which enables developer with low-level access to manage data allocations and transfer, scheduling and pipelining. The latest version of VisionWorks is 1.6.

3.2.3. Xilinx xfOpenCV:

The xfOpenCV library is a set of OpenCV functions optimized for Zynq, Zynq Ultrascale+, and Alveo FPGAs devices by Xilinx [23]. It was first released in 2017, as part of the Xilinx reVISION stack. xfOpenCV kernels are implemented using HLS to work in their SDx development environment and provides a software interface for building vision pipelines on FPGAs. The library includes a set of 60+ vision kernels optimized to be mapped into the programmable logic. The latest version of the xfOpenCV library is 2019.1.

3.3. Neural Network Inference Frameworks

In this work, we used the following three deep learning inference frameworks to benchmark neural networks:

3.3.1. OpenCV DNN Module:

OpenCV DNN module has been promoted from the contrib repository to the main repository since the release of OpenCV 3.3 [24]. The module now supports deep learning frameworks such as Caffe, TensorFlow, and Torch/PyTorch. It only supports the forward pass by importing weights from pre-trained models. OpenCV DNN also includes a set of pre-processing functions for preparing images, such as: cropping, channel swapping, mean subtraction, etc. Examples of compatible network architectures include: GoogleLeNet, AlexNet, SqueezeNet, VGGNet and ResNet. The module supports SSE, AVX, NEON acceleration and Halide backend.

3.3.2. *NVIDIA TensorRT:*

TensorRT is a framework for implementing high-performance inference on NVIDIA GPUs [25]. TensorRT applies couple of optimizations to deep learning networks such as: (1) Weight and activation precision quantization to FP16 and INT8 to maximizes throughput while maintaining accuracy, (2) Optimizing the use of GPU memory and bandwidth by layer and tensor fusion, (3) Kernel auto-tuning to select best data layers and algorithms based on target GPU platform, (4) Dynamic tensor memory allocation to re-use memory efficiently, (5) Multi-stream execution to process multiple input streams in parallel. The integration of TensorRT with TensorFlow allows for applying TensorRT GPU optimizations within TensorFlow environment.

3.3.3. *Xilinx Vitis AI:*

Xilinx Vitis AI is a deep learning framework that provides a combination of flexibility, high performance, low latency and low power consumption for deploying deep learning inference into Xilinx FPGAs and SoCs [26]. It allows for compressing DNN models to reduce their size without loss of accuracy, and compiling DNN models into DPU instruction code before deploying them into the target DPU platform. Xilinx DPU provides a customized and scalable overlay with ISA architecture for optimized DNNs implementations.

Examples of other libraries and frameworks for embedded DNNs: (1) Intel OneDNN [27]: is a framework that supports heterogeneous execution across Intel CPUs, Graphics, FPGAs and vision accelerators VPUs. (2) ARM NN[28]: is a framework that enable efficient translation of DNNs allowing them to run efficiently across ARM Cortex-A CPUs, ARM, Mali GPUs and ARM Ethos NPUs. (3) FAST DNN[29]: is a framework that is optimized for CPUs using SIMD instructions, linear quantization, batch processing, sigmoid lookup and lazy output calculation.

3.4. *Categories of Vision Kernels*

Computer vision algorithms can be grouped into six categories based on their functionality, as shown in Figure 1. The complexity of these kernels grows over the first five categories. The last category includes composite kernels, which are composed of kernels from other categories. The following discusses each category in more detail:

3.4.1. *Input Processing:*

The kernels in this group are usually used as pre-processing steps. They include simple arithmetic operations to change the input format or number of channels into a desired format. Some examples of these kernels are: channel combine, channel extract, color conversion, and bitdepth conversion.

3.4.2. *Image Arithmetic:*

Image arithmetic applies standard arithmetic/logic operations to one or more images. Because of the multi-dimensional nature of these pixel based operations, these kernels can benefit from highly parallel hardware architectures, such as GPUs and FPGAs. Furthermore, the data being processed is very localized; the algorithms can be distributed among different processing units without concerns of data dependencies. These operations include: thresholding, absolute difference, addition/subtraction, bitwise and/or/xor/not, multiplication, accumulate, accumulate squared, and accumulate weighted.

3.4.3. *Image Filters:*

These algorithms compute the correlation between an input image and a kernel (small matrix of fixed-size). The data in these algorithms are local to the size of the kernel which is different from the arithmetic case where the operations were performed on a pixel basis. When the underlying hardware has enough local memory to accommodate the kernel size, the algorithm is still easily distributed among parallel processing units. On the other hand, nonlinear filters are more irregular as they have branching conditions. This impedes their decomposition into parallel blocks. These kernels include: filter2D, box filter, erode, dilate, median, pyramid up, and pyramid down.

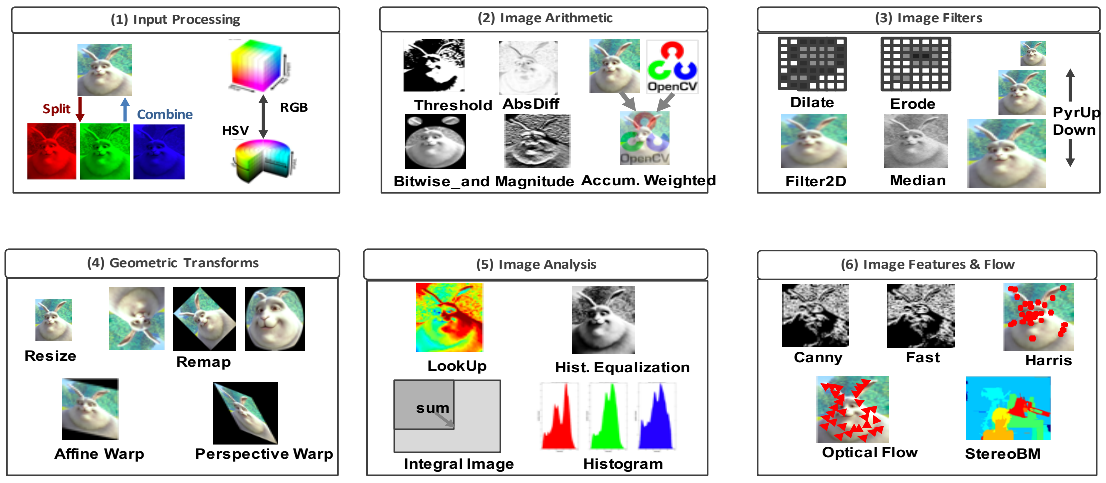

Figure 1: Examples of Vision Kernels from the Six Categories.

3.4.4. *Image Analysis:*

Analytic kernels are typically used to understand characteristics of an image, such as color distribution, mean, maximum and minimum pixel value, etc. Also, they are usually placed at the end of vision pipelines to reduce the image into a decision variable (min/max locations). These kernels are filled with branching conditions and complex memory access patterns that negatively impact their performance on CPUs and GPUs. These operations include: histogram, mean/std, min/max location, table lookup, histogram equalization, and integral image.

3.4.5. *Geometric Transformation:*

Transformations in geometric space are essential to understanding the 3D world through the lens of a 2D image sensor. These kernels include matrix multiplication that map effectively into highly parallel architectures composed of simple computing blocks (e.g. GPU). While these kernels are simple, their performance is negatively affected by irregular memory access patterns. These kernels include: remap, resize, affine warp, and perspective warp.

3.4.6. *Composite Kernels:*

The kernels in this category are composed in part of kernels from the previously described categories. Examples of these composite kernels are: feature extraction, stereo block matching, and optical flow. *Feature extraction* is used to find interesting pixels in an image. Once features are extracted, they are no longer stored as a continuous block of adjacent pixels in memory. This forces other kernels to load non-continuous memory addresses, which may hinder parallelism performance. *Stereo block matching* uses two cameras, with known position and characteristics, to compute disparity by comparing overlapped regions, leading to a high computational load. *Optical flow* is used to estimate the apparent motion of objects between two consecutive images. Optical flow can be computed for each pixel (dense) or a subset of pixels (sparse).

3.5. *Neural Networks*

Convolutional Neural Network (CNN) is a special class of multi-layer neural networks, designed to recognize and analyze visual patterns directly from pixel images. They are usually comprised of a sequence of convolutional layers, activation functions, pooling layers, fully connected layers and normalization layers. In this study, we focused on five CNNs: (1) Inception-v2: a 22 layers network that introduces a special 1x1 convolution, and using global average pooling instead of using fully connected layers [30]. (2) ResNet-50, short for Residual Network. It introduces the idea of (identity shortcut connection) that skips one or more layers to address the vanishing gradient problem [31]. It is a deep residual network of 50 layers. (3) ResNet-18: one of the residual network variants. It consists of 18 layers. (4) MobileNet: is a small model built upon the idea of using depthwise separable convolutions as efficient building blocks [32]. (5) SqueezeNet: is a family of models that achieve AlexNet-level accuracy on ImageNet with 50x fewer parameters [33].

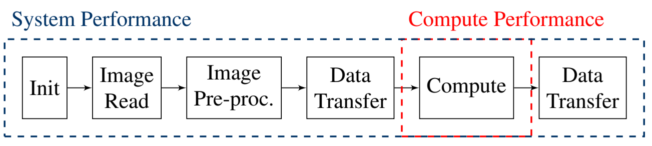

Figure 2: Steps involved in compute and system performance measurements.

4. Experimental Methodology

This section describes the performance metrics, hardware and software environments used in our benchmarking setup. It also describes measurement techniques, and introduces our benchmarking approach.

4.1. *Performance Metrics*

In this work, we evaluate the efficiency of vision kernels and neural networks using four performance metrics: (1) accuracy, (2) run-time, (3) energy per frame, and (4) energy delay product (EDP). These metrics provide a fair way of comparison between different design points and a meaningful interpretation to make design choices. In this subsection, we discuss the performance metrics, as follows:

4.1.1. *Accuracy:*

To evaluate the accuracy of vision kernels, we compute the L1-norm error between the results generated by the GPU and FPGA implementations and compare them to the CPU implementation. Table 8 in the appendix shows the L1-norm errors. For neural networks, classification accuracy (test error rate) is used to evaluate the CPU, GPU and FPGA implementations by comparing their results to the ground truth in the whole Imagenet validation set with 50k images. The top-1 and top-5 classification accuracy are reported in this paper for Inception-v2, ResNet-50, ResNet-18, Mobilenet-v2 and SqueezeNe neural networks.

4.1.2. *Run-time:*

There are two different types of run-time performance measurements: (1) *Compute performance* which measures only the compute part of vision kernel or neural network excluding potential bottlenecks for moving data from/to the external memory. Even though it doesn't capture the application level performance, it reflects the efficiency in preforming various compute patterns, (2) *System performance* which measures the performance of the complete pipeline, including initialization, image reading, pre-processing, post-processing and data transfer. It measures the un-optimized performance of the platform by capturing data movement bottlenecks. Figure (2) shows steps involved in each of the compute and systems performance measurement.

4.1.3. *Energy:*

Energy consumption per frame quantifies the amount of electrical energy dissipated by hardware accelerators to perform a kernel's operations on one frame. It is measured as the power consumed during the delay time to process a frame. Device power can be divided in two parts: (1) Static power: represents the amount of power consumed when no active computation is taking place (system is idle), (2) Dynamic power: represents the amount of power consumed above the static power level when the system is computing.

4.1.4. *Energy-Delay Product (EDP):*

Run-time or energy per frame alone do not show the entire picture. A hardware platform can be extremely low power while being too slow to be of practical use. The Energy Delay Product (EDP) [34] metric takes into account the throughput of the algorithm measured in (ms/frame) along with the energy consumed per frame (mJ/frame). EDP is the product of energy/frame and delay time. This way, a fair comparison can be made when deciding which hardware architecture is better suited for specific computation. Lower EDP is better which means that the hardware architecture can finish specific computation tasks using less power in less time.

4.2. *Measurement Techniques and Platforms:*

In this study, we evaluated two popular platforms for deploying embedded vision applications: Nvidia Jetson TX2 and Xilinx ZCU102. These platforms come equipped with an on-board power measuring IC that can measure multiple power rails such as: CPU cores and GPU cores on the Jetson, and programmable logic, full power CPU cores and low power CPU cores on the FPGA platform. On the Jetson TX2, shell scripts (running on its ARM CPU) sample power rails and log their values along with the system's timestamp into text files. The act of measuring power consumes power, thus consequently affects the results. The presented data in this paper has been corrected for this. On the ZCU102, a Python script is used to sample power rails that are accessible through the INA226 and are mapped to PS readable virtual files in sysfs.

For every benchmark, we first processed 1000 frames on the CPU core of the platform and then 1000 frames on the hardware accelerated part of the platform. This can be seen in Figure (3), where the first two vertical lines mark the first 1000 frames on the CPU and the following two lines mark the last 1000 frames on the hardware. We computed the average frame rate by measuring the time between vertical lines and divided it by 1000. The x-axis represents the number of power samples taken for each platform. Note that the ZCU102 has a different sampling rate than the TX2. For vision kernels and pipelines, input frames were in gray-scale with 1080p resolution. For neural networks, frames were in RGB with 224×224 resolution. We also used 1024 frames instead 1000 frames to have multiple of batch sizes.

4.2.1. *Hardware environments.*

In this work, we used Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board. It has a 16nm XCZU9EG FPGA, and an on-board 4GB 64bit DDR4 RAM with a peak bandwidth of 136Gb/s. For GPU board, we used the Nvidia Jetson TX2 (Pascal 256 CUDA cores (16nm)) has 8GBs of 128bit DDR4 RAM with a peak bandwidth of 477.6 Gb/s. Both the FPGA and GPU have on-chip ARM CPU cores with NEON SIMD optimization.

The TX2 GPU board supports three operating modes with different clock frequencies and power consumptions, as follow: (1) Max-Q: (maximum energy efficiency) in this mode

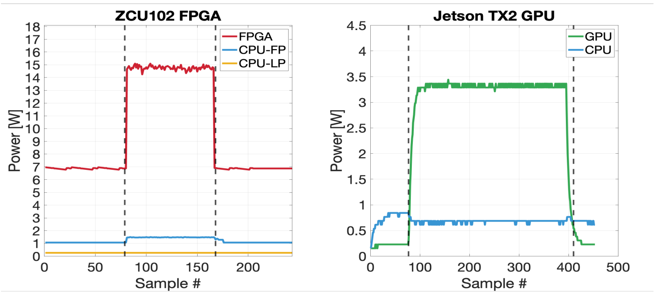

Figure 3: Measuring power samples on the platform's CPU cores (first 1000 frames), and its FPGA or GPU (second 1000 frames).

all components on the TX2 are configured to achieve the best power-throughput tradeoff. (GPU @ 0.85GHz). (2) Max-P: this mode increases the GPU's clock frequencies to increase the performance sacrificing the power (GPU @ 1.12GHz), and (2) Max-N: (maximum clock) is the maximum performance mode allowing the TX2 to hit higher performance at the cost of some energy efficiency (GPU @ 1.30GHz).

On the ZCU102 FPGA baord, the xfOpenCV FPGA kernels are clocked at 300 MHz, and the Xilinx DPU overlay at configuration mode (3xB4096) runs at 333 MHz. The ARM-A57 is clocked at 1.7 GHz. Table 1 shows the theoretical peak performance of the platform used in this paper.

Table 1: Theoretical Peak Performance of Hardware Platforms [13]

Platform	Configuration	Data types	Perf [TOPS]
ARM Cortex A57	-	FP32, FP64	0.41
Xilinx ZCU102	-	INT8	6.71
NVIDIA Jetson TX2	MaxN	FP16, FP32	1.33, 0.67
NVIDIA Jetson TX2	MaxP	FP16, FP32	1.15, 0.57
NVIDIA Jetson TX2	MaxQ	FP16, FP32	0.87, 0.44

4.2.2. *Software environments.*

We used three publicly available vision libraries: (1) OpenCV 3.4, (2) Nvidia's VisionWorks 1.6, and (3) Xilinx's xfOpenCV 2018.3. While the OpenCV code base already comes with some GPU accelerated code, it does not come with FPGA support. For this purpose, we used OpenCV compatible C++ wrappers for xfOpenCV kernels [35]. With this wrapped functionality we were able to compile the same OpenCV code for both GPU and FGPA. Both OpenCV and VisionWorks support full IEEE FP precision, while xfOpenCV supports 8 bit precision.

In neural network benchmarking, NVIDIA TensorRT and Xilinx Vitis AI are used, since both are hardware-specific frameworks that are optimized for neural network inference on embedded GPUs and FPGAs, respectively. We used OpenCV 3.4 DNN module to evaluate the performance of ARM57 CPU. On NVIDIA side we used Jetpack 3.3 on TX2 and Jetpack 4.1.1 with corresponding TensorRT versions 1. For Xilinx platforms, the Xilinx Vitis AI framework version 1.1 is used.

4.3. *Benchmarking Approach*

In this study, we intentionally focused on evaluating the performance of out-of-the-box kernels from publicly available libraries (without writing special platform specific code around kernel calls) to give a fair comparison in terms of development efforts.

For this reason, we first ran single kernel calls from OpenCV and VisionWorks libraries on the CPU and GPU, respectively, and instantiated a single kernel from xfOpenCV in FPGA fabric (even though small kernels utilize few FPGA resources). We then measured the efficiency of representative vision pipelines on the three HW accelerators to quantify their speed and energy efficiency on these more complete vision applications.

For single kernel evaluation, we compared the efficiency of the HW accelerators in terms of their energy consumption per frame. We measured a vision kernel's dynamic power while excluding the static power required to power the rest of the platform. This better reflects the actual workload that is being deployed to the system since certainly for small kernels, the *compute energy* [4] (energy consumed for computation only) and data transfer energy are usually dominated by the static power. In the vision pipeline evaluation, we compared the performance of HW accelerators in terms of their energy delay products (EDP). We used the total power consumption (static + dynamic), because it represents the actual power consumption when a complete system is deployed. We also measured the maximum frame rate achieved on the three HW accelerators. The theoretical frame rate on the FPGA is fixed for vision kernels that perform a single pass over the input image. Equation (1) shows an FPGA's frame rate when it is clocked at 300MHz for 1080p images.

$$FPS = \frac{300MHz}{1080 \times 1920 \times 1 cycle/pixel} = 144 \qquad (1)$$

In our experiments, we measure run-time as follow:

- *Vision Kernels*: We measure the compute performance and the time to transfer data from/to these kernels.
- *Complete Vision Pipelines*: We measure the compute performance and the time to transfer data from/to the input and output of these pipelines. Communication between the pipeline's kernels is local on the FPGA (through FIFOs) and depends on the caching on the GPU.

Table 2: Data Movers Energy Consumption Measurements

platform	Time/frame (ms)	Energy/frame (mJ/f)
FPGA	6.945	0.41
GPU	1.298	0.19

- *Neural Networks*: Due to library limitations, we could not measure the isolated compute only performance in neural networks. The weights and activation maps need to be streamed from/to off-chip memory between layers. Therefore, only the system performance of neural networks is measured and reported in this work.

In our experiments, we measure the power as follows:

- *Vision Kernels*: dynamic power only using the following power rails: (VCCINT) in the FPGA, (VDD_SYS_GPU) in the GPU, and (VDD_SYS_CPU) in CPU.
- *Complete Vision Pipelines*: total power (static + dynamic) using the following power rails: (VCCINT) power rail in the FPGA, (VDD_SYS_GPU) in the GPU, and (VDD_SYS_CPU) in the CPU.
- *Neural Networks*: total power (static + dynamic) using the following power rails: (VCCINT) power rail in the FPGA, (VDD_SYS_GPU) in the GPU, and (VDD_SYS_CPU) in the CPU.

In order to have a sense of the amount of energy consumed for computation only, we measured the energy consumption of data movers in the FPGA and GPU. We implemented passthrough kernels which copy pixels from one memory location to another without applying any arithmetic/logical operations. In the FPGA implementation, Xilinx's SDx tool instantiates data movers [36] for each input or output port to transfer data between the memory mapped domain and the stream domain. Table 2 shows that FPGA takes 6.945 ms to copy an entire image (1080p) with 0.41 mJ/frame, while GPU takes 1.298 ms with 0.19 mJ/frame. These values can be used to give a sense of the ratio of energy consumed for computation to data transfer in each kernel.

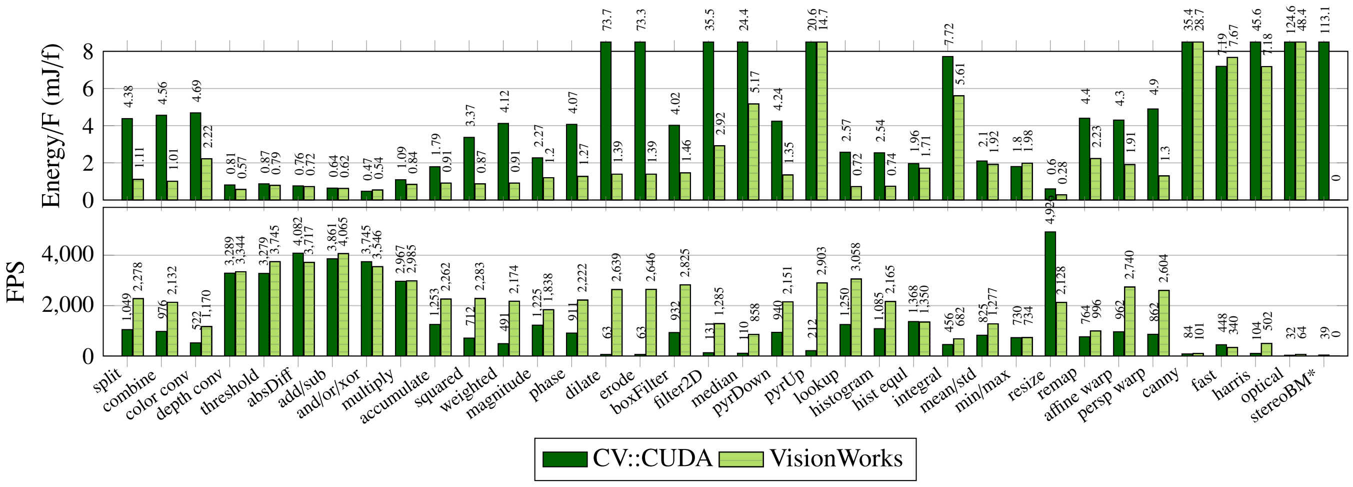

Figure 4: VisionWorks outperforms OpenCV CUDA module in terms of frame rate and energy/frame. *VisionWorks's implementation of the stereoBM kernel is not publicly available.

5. Experimental Results

This section first presents the benchmarking results of single vision kernels and a set of representative vision pipelines are evaluated. Then, it shows the results of a set of neural networks.

5.1. Single Kernel Performance:

Before evaluating the run-time performance and energy/frame consumption of single kernels on the HW accelerators, we first compare two available GPU implementations: OpenCV CUDA module and Nvidia's VisionWorks toolkit. The OpenCV GPU module is written using CUDA and as a result benefits from the CUDA ecosystem. The Visionworks library applies many optimization techniques to boost performance, such as buffer reuse, kernel fusion, efficient use of streaming and CUDA textures, automatic scheduling across processing units, tiling and pipelining vision functions at the sub-frame level. Figure (4) shows the frame rate (bottom) and energy per frame (top) achieved by running vision kernels on the Jetson TX2. The dark color represents OpenCV CUDA module, and the light color represents VisionWorks. We can observe that the VisionWorks implementation outperforms the OpenCV module in frame rate over all kernels. It achieved up to a 9.7× speedup compared to the OpenCV module. It also consumes less energy per frame over all kernels. It achieved up to a 6.3× reduction in energy consumption per frame. For this reason, in the rest of the paper, we will use only the VisionWorks implementation for the GPU.

Next, we measured the energy per frame consumption of vision kernels from the following six categories: (1) input processing, (2) arithmetic operations, (3) filter operations, (4) image analysis, (5) geometric transformation, and (6) composite kernels.

Input processing: The energy/frame of input processing kernels is shown in Figure (5). These kernels mapped well to the GPU and FPGA compared to the CPU because of their significant data parallelism, low complexity, and no data dependency. The GPU and FPGA achieved an average reduction ratio of 1.79× and 1.41× in energy/frame compared to the CPU. It also shows that GPU's implementation of bit-depth conversion achieved a 2.4× reduction compared to FPGA, because of the efficient use of streaming and CUDA textures in the VisionWorks kernel's implementation. In all kernels, the GPU has an order of magnitude speed up compared to CPU and FPGA in terms of frame rate.

Image Arithmetic: The performance of arithmetic/logic operations is shown in Figure (6). It shows that simple operations such as: threshold, absDiff, add/sub, and bitwise and/or/xor can be efficiently implemented by the CPU. However, the CPU starts to perform poorly in kernels with multiplication operations, such as: multiply, accumulate squared, weighted, magnitude and phase. The GPU has the lowest energy/frame compared to the CPU and FPGA. The GPU's implementations achieved an average reduction ratio in energy/frame of 4.6× and 7.2× compared to CPU and FPGA, respectively. An expected result, as these algorithms can be granulated into many pieces that execute the same operation (SIMT).

Image Filters: In Figure (7), the results of filtering operations show that the FPGA performs better than the GPU and CPU for these kernels. The FPGA's implementation achieved an average reduction ratio of 1.8× and 7.4× in energy/frame compared to the GPU and CPU, respectively. The memory access patterns and mathematical complexity of linear filters (filter2D, box filter, pyramid up and pyramid down) maps well to the parallel processing of the GPU and FPGA. Median filters, however, are unlike linear filters. They do not use sequential data access and multiply-and-accumulate operations, but sort input elements and select the median of them, which makes them less straightforward to implement efficiently on a GPU. The morphological operations (dilate and erode) use hit and miss functions over a structuring element. These functions are more difficult to implement than filtering functions due to comparison and branching. This explains the low frame rate (as shown in Figure 2) and high energy/frame consumption of VisionWorks's implementations of small (3×3) filter kernels.

Image Analysis: The results of the image analysis kernels are shown in Figure (8). For kernels such as lookup table, histogram, and histogram equalization, the energy/frame consumption of the FPGA achieves an average reduction of 1.2× compared to the GPU. While for kernels with more branching conditions and complex memory access patterns, such as integral image, mean/std, and min/max locations, the FPGA's implementation achieved an average reduction ratio of 3.5× compared to the GPU.

Geometric Transformation: The results of the geometric transformation kernels are shown in Figure (9). The CPU performs poorly for these kind of operations compared to the GPU and FPGA. Also, the FPGA was more energy efficient compared to

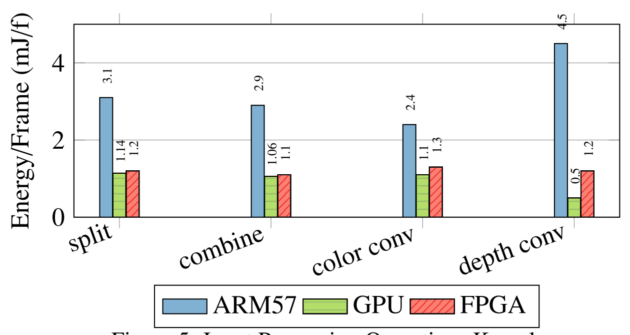

Figure 5: Input Processing Operations Kernels

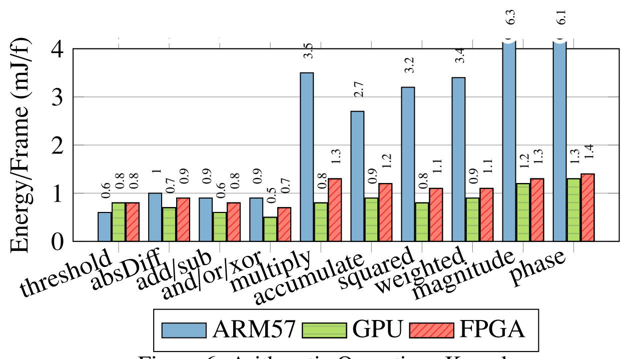

Figure 6: Arithmetic Operations Kernels

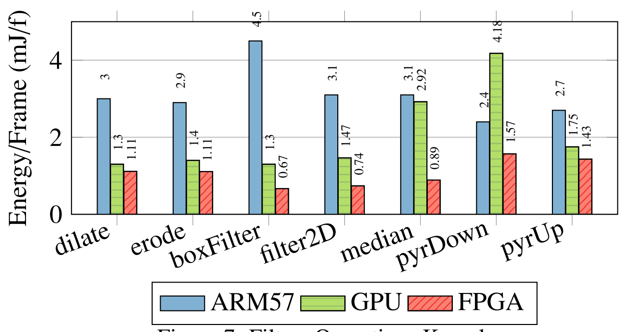

Figure 7: Filters Operations Kernels

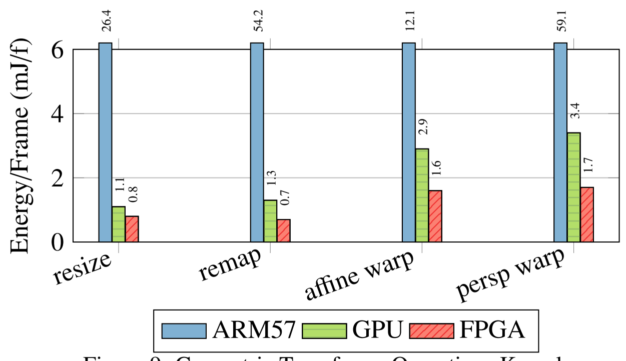

Figure 9: Geometric Transforms Operations Kernels

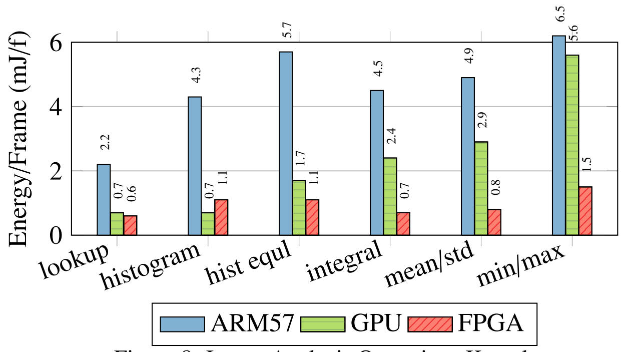

Figure 8: Image Analysis Operations Kernels

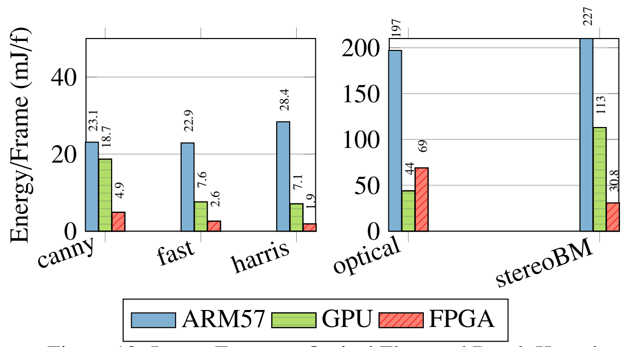

Figure 10: Image Features, Optical Flow and Depth Kernels

the GPU. It achieved a reduction of 1.6× in energy/frame for the resize and remap kernels, and 2× for affine warp and perspective warp kernels. The computations in the warp operations are more complex compared to resize and remap as mapping addresses need to be generated from 2×3 or 3×3 matrices before starting the mapping operation. The mapping process in these kernels is done from destination to source in order to avoid sampling artifacts and visiting every pixel in the destination image multiple times.

Composite Kernels: The last category in our study includes kernels for: (1) detecting image features (canny, fast and harris), (2) computing optical flow, and (3) computing disparity using stereo block matching. Figure (10) shows that the FPGA implementation of feature extraction kernels (canny, fast and harris) were more energy-efficient compared to the CPU and GPU by an average reduction of 7.7× and 3.5×, respectively. The steps to calculate sparse optical flow using the pyramid Lucas-Kanade algorithm includes extracting feature points from one frame and tracking them in the next frame. The FPGA implementation was able to detect 488 Harris corners compared to 94 for VisionWorks for the same input frame and parameters. Also, it was able to keep track of these points in the next frame. This explains the high energy/frame consumption in the FPGA implementation. Moreover, the VisionWorks's implementations of StereoBM is not open sourced yet, so the number reported in this paper is for the GPU implementation using OpenCV's CUDA module instead.

The average energy/frame reduction for the GPU and FPGA is shown in Table 3. The ratio is with respect to CPU consumption (higher is better). We can observe a trend from simple kernels (top) to more complex kernels (bottom). The trend demonstrates that the performance of the GPU and FPGA compared to the CPU improves as kernels' complexity increases. For simple kernels (input processing and image arithmetic), the GPU shows the highest performance/energy efficiency, while for more complicated kernels (image filters, image analysis and geometric transform), the FPGA shows the highest performance/energy efficiency. Moreover, as the complexity of kernels increase, the FPGA shows higher energy-efficiency compared to the GPU and CPU. This occurs due to the fact that more complex algorithms naturally occupy more resources on the programmable logic, as well as the fact that GPUs do not scale well for problems that are not easily divisible (data locality) or have many conditions or complex memory access patterns.

Table 3: Ratios of Energy/Frame Reduction (Reference CPU)

	CPU	**GPU**	**FPGA**
Input Processing	1	**1.79**×	1.41×
Image Arithmetic	1	**3.19**×	2.93×
Image Filters	1	3.17×	**3.89**×
Image Analysis	1	2.34×	**5.67**×
Geometric Transform	1	10.3×	**16.6**×
Features/ OF/ StereoBM	1	7.44×	**22.3**×

For completeness, we did a frame rate comparison between the ARM57 CPU OpenCV and GPU VisionWorks implementations. Our result shows that VisionWorks implementations outperform OpenCV implementation by an average speedup of 2.9×, 4.2×, 6.3×, 3.9×, 42× and 4.5× for the six categories of vision kernels. The FPGA's frame rate met the theoretical rate of Equation (1) for kernels performing a single pass over the input image (144 fps @300MHz for 1080p). To validate kernels' accuracy, we used OpenCV's output image as our reference and computed pixel-wise subtraction with the VisionWorks and xfOpenCV outputs to measure differences. We had no differences for all reported vision kernels.

5.2. Complete Vision Pipeline Performance:

In this section, we evaluated the performance of the HW accelerators for four representative pipelines. Common steps in many computer vision pipelines include: pre-processing, feature extraction, and post-processing. The pipelines used in our study follow this structure: (1) background subtraction, (2) color segmentation, (3) stereo block matching, and (4) Harris corner tracking. These pipelines are implemented on the GPU using VisionWorks OpenVX graph mode to enable its advanced optimization techniques (buffer reuse, kernel fusion, etc.). We also pipelined the execution of kernels on the FPGA at pixel/frame level using xfOpenCV modules. In this way, the FPGA can leverage the fact that image pixels stays within the programmable fabric and avoids going back and forth to read/write from external memory. In terms of CV pipelines accuracy, the results in OpenCV matches the GPU and FPGA. The pipelines evaluated in this paper are:

5.2.1. Background Subtraction:

The background subtraction pipeline is used to detect changes in image sequences [37]. It is mainly used when regions of interest are foreground objects. The pipeline components include: subtraction, Gaussian filtering, threshold, erode and dilate, as shown in Figure (11).

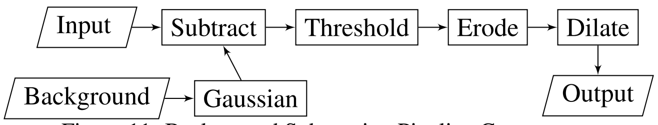

Figure 11: Background Subtraction Pipeline Components

5.2.2. Color Segmentation:

This pipeline is used to partition an image into multiple segments based on a specific range of colors. It converts the color format from RGB to HSV, then applies range thresholding to its three channels, and applies erode and dilate operations, as shown in Figure (12).

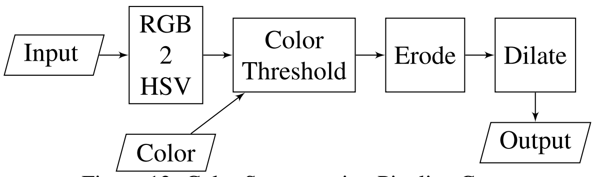

Figure 12: Color Segmentation Pipeline Components

Table 4: FPGA's Reduction Ratios with respect to GPU

Pipeline	**Energy/frame (mJ/f)**	**EDP (mJ.s/f2)**
Background Subtraction	1.74×	1.32×
Color Segmentation	1.86×	1.41×
Harris Corners Tracking	3.94×	2.65×
Stereo Block Matching	8.83×	107.7×

5.2.3. Harris Corners Tracking:

This pipeline is used to detect and track feature points in a set of successive frames of a video. It takes in the current and next frame as inputs. It computes Harris corners from the current frame and outputs a list of tracked corners in the next frame. The pipeline uses five kernels: Gaussian pyramid, Harris corner detection, Optical flow and update corners kernels, as shown in Figure (13).

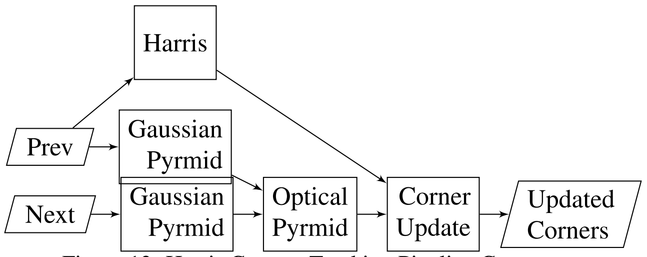

Figure 13: Harris Corners Tracking Pipeline Components

5.2.4. Stereo Block Matching:

This pipeline is used to generate a disparity map given the camera parameters and inputs from a stereo camera setup. It is used as a first step in creating a three dimensional map of an environment. The main components involved in the pipeline are shown in Figure (14). It consists of stereo rectification, remapping, and disparity estimation using a local block matching method.

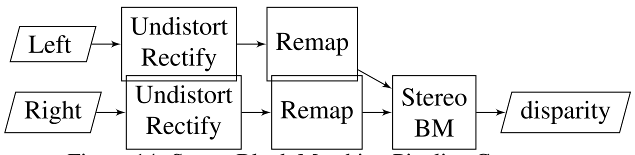

Figure 14: Stereo Block Matching Pipeline Components

Figure (15) plots the Energy/frame and EDP comparison of the four pipelines, and shows the FPGA implementations consume less energy/frame compared to the CPU and GPU for all pipelines. The FPGA is also more efficient in terms of EDP (lower EDP is better). The FPGA's Energy/frame and EDP reduction ratio with respect to the GPU is listed in Table 4. As the complexity of the pipeline grows, the energy/frame and EDP reduction ratio increases. More complex vision pipelines can use more of the FPGA programmable logic, reducing the relative impact of static power consumption. Additionally, data communicated between modules of the pipeline are kept on-chip in the streaming FPGA implementation.

Table 5: Top-5 (Top-1) Classification Accuracy

Library	Inception-v2	ResNet-50	ResNet-18	MobileNet-v2	SqueezeNet
OpenCV DNN (FP32)	91.1 (72.75)	91.85 (74.44)	88.48 (68.32)	86.1 (64.75)	78.13 (54.38)
TensorRT (FP16)	90.8 (72.01)	91.15 (72.86)	89.30 (69.93)	86.4 (65.40)	76.30 (52.29)
Xilinx DPU (INT8)	90.30 (71.68)	91.31 (73.34)	88.25 (66.94)	85.06 (63.54)	76.58 (50.26)

5.3. *Neural Network Inference Performance*

In this section, we measure the accuracy and performance of five different neural networks: Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet. We benchmark their implementation using OpenCV DNN, TensorRT, and Vitis AI frameworks on an ARM-57 CPU, Jetson TX2 GPU, and ZCU102 FPGA, respectively. We also evaluate the performance of hardware optimizations: reduced precision implementations (GPU and FPGA), multiple batch sizes and different operating modes (GPU) and different threads counts (FPGA). These implementations are evaluated using the following performance metrics: test accuracy and system performance (frame rate, energy/frame and EDP).

5.3.1. *Accuracy:*

The Top-1 and Top-5 classification accuracy achieved by Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet implemented using OpenCV DNN, TensorRT and DPU is shown in Table 5. The accuracy is measured on the ImageNet-1K validation set (ILSVRC-2012). These networks have different computational complexity [GFOPs] and parameters size [MBs]. We listed these networks in the Table 5 based on their GOPs/MBs ratio: Inception-v2 (6 GOPs/ 91MB), Resnet-50 (4 GOPs/ 98MB), Resnet18 (2 GOPs/ 45MB), MobileNet-v2 (0.3 GOPs/ 14MB) and SqueezeNet (0.36 GOPs/ 5MB). The results show that reducing the bit precision from FP32 in OpenCV DNN to FP16 in TensorRT and INT8 in DPU keeps the Top-5 (Top-1) accuracy loss within $\sim 2\%(\sim 4\%)$. This suggests that these models can be used with precisions as low as INT8 which will reduce model complexity by 4x while maintaining an acceptable accuracy loss.

5.3.2. *System Performance:*

System performance measures the efficiency of the complete inference pipeline including its pre-processing, computation, data movement, and post-processing stages which gives insight into the actual performance achieved after deployment. It captures potential memory bandwidth bottlenecks in data copying between off-chip and on-chip memories.

In our experiment, we measure the performance of CPU, GPU and FPGA implementations of Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet networks. We measure their performance at multiple batch sizes (b=1,b=2, ..., b=128) and thread counts (t=1, t=2, ..., t=8) to evaluate the effect of increasing batch size on data reuse and data movements reduction. Figure 16 and 17 show the experimental results of Inception-v2 and ResNet-50. The blue, green and red bars represents ARM-57, GPU and FPGA, respectively. In these figures, we selected the GPU's most power efficient operating mode (MaxQ) and its reduced precision (FP16). We measured the frame rate and EDP at different batch sizes and thread counts. Then, we selected the highest frame rate and lowest EDP among them for the side-to-side comparison.

The results show that even with the limited memory bandwidth in the FPGA, it was able to achieve higher frame rates

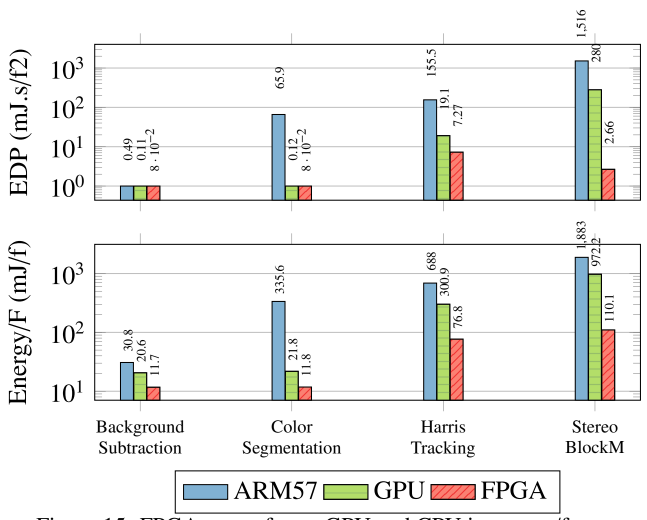

Figure 15: FPGA outperforms GPU and CPU in energy/frame consumption and EDP

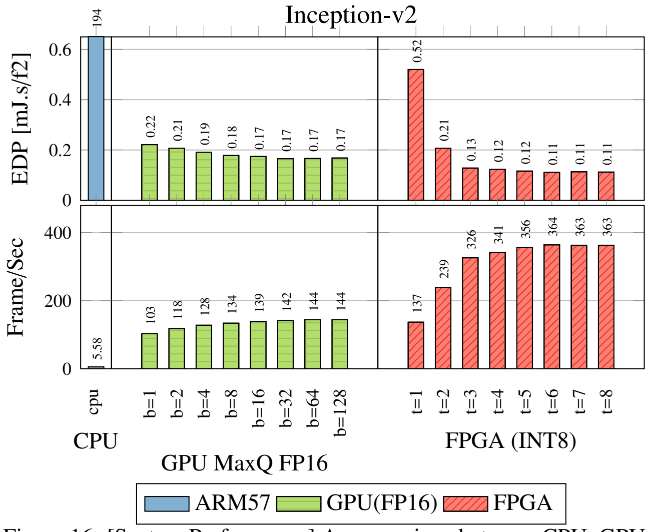

Figure 16: [System Performance] A comparison between CPU, GPU and FPGA in terms of frame Rate (fps) and energy delay product (EDP) for Inception-v2 Network.

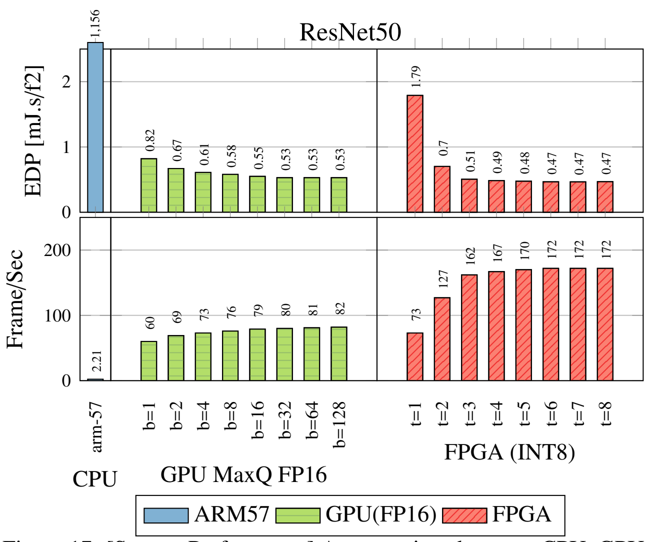

Figure 17: [System Performance] A comparison between CPU, GPU and FPGA in terms of frame Rate (fps) and energy delay product (EDP) for ResNet50 Network.

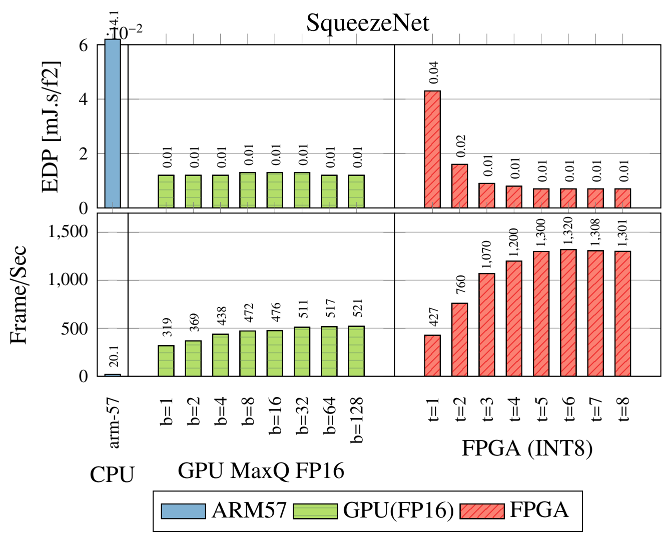

Figure 19: [System Performance] A comparison between CPU, GPU and FPGA in terms of frame Rate (fps) and energy delay product (EDP) for SqueezeNetV2 Network.

compared to CPU and GPU. For Inception-v2, the FPGA (t=6) achieves a speed up of 2.5× and 65× compared to the GPU FP16 (b=128) and CPU. For ResNet50, the FPGA (t=8) also achieves a speed up of 2.1×, and 77× compared to the GPU FP16 (b=128) and CPU, respectively. This speed-up comes from the low numerical precision (INT8) used in FPGA compared to the (FP32 and FP16) in CPU and GPU, as well as multiple optimizations supported by Xilinx Vitis AI framework such as: (1) memory allocation, scheduling and reusing, (2) node fusion/decomposition and (3) data stream optimization. Moreover, it is noticed that the improvement in frame rate starts to saturate as batch size and thread count increases due to reaching the maximum chip capacity. Another observations is that the FPGA implementations of Inception-v2 and ResNet50 are 4.72× and 3.8× more energy efficient when number of threads equals (t=8) compared to (t=1).

In terms of EDP, the FPGA implementations have lower EDP values compared to the CPU and GPU FP16 implementations. Figure 16 shows that FPGA implementation (t=8) of Inception-v2 has an EDP reduction ratios of 1.5× compared to the GPU FP16 (b=128). Figure 17 shows that FPGA implementation (t=8) of ResNet50 has 1.1× EDP reduction ratios compared to the GPU FP16 (b=128).

Figure 18, 20 and 19 show the experimental results for the small networks: ResNet-18, Mobilenetv2 and SqueezeNet. For

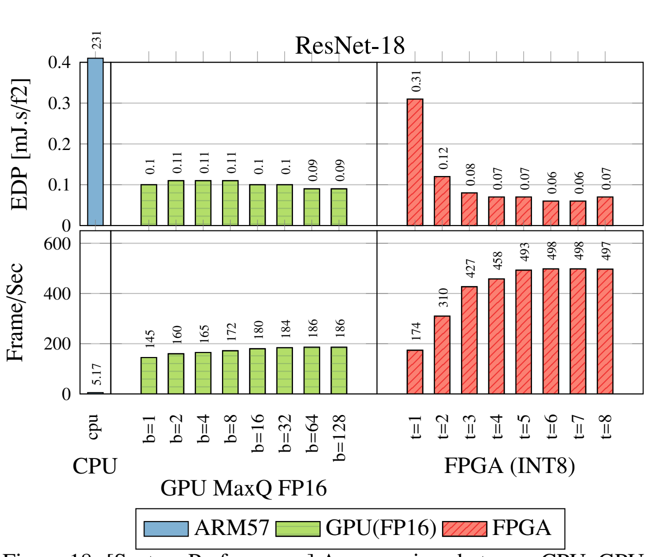

Figure 18: [System Performance] A comparison between CPU, GPU and FPGA in terms of frame Rate (fps) and energy delay product (EDP) for ResNet-18 Network.

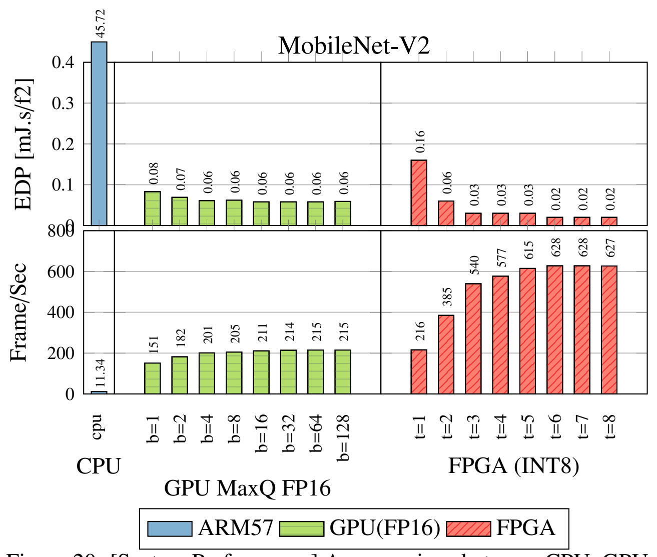

Figure 20: [System Performance] A comparison between CPU, GPU and FPGA in terms of frame Rate (fps) and energy delay product (EDP) for MobileNet Network.

ResNet-18, the FPGA (t=6) achieves a speed up of 2.6× and 96× compared to the GPU FP16 (b=128) and CPU. For Mobilenetv2, the FPGA (t=7) also achieves a speed up of 2.9×, and 55× compared to the GPU FP16 (b=128) and CPU. For SqueezeNet, the FPGA (t=6) achieves a speed up of 2.5×, and 65.6× compared to the GPU FP16 (b=128) and CPU.

Table 6 summarizes the FPGA's frame rate and EDP reduction ratio compared to the GPU FP16 implementations. The FPGA is 2.1–2.9× faster and 1.1–2.4× more energy efficient than GPU F16 implementations when running Inception-v2, ResNet-50, ResNet-18, Mobilenetv2 and SqueezeNet.

6. Conclusion

The development of cost-efficient embedded vision applications is challenged in its initial design phase by the variety of hardware solutions and software libraries. This paper performs an in-depth benchmark analysis of three embedded platforms, CPU, GPU- and FPGA-accelerated, evaluating the efficiency of their different hardware architectures towards vision kernels, complete vision pipelines and neural networks [Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet]. To support reproducibility, the benchmark only relies on publically available libraries and frameworks. Given the energy-efficiency focus, three key metrics are collected in the benchmarks: energy per frame, frame rate and energy delay product (EDP).

The experimental results show that many simple and easy-to-parallelize vision kernels perform well on GPUs (1.1–3.2× energy/frame reduction), but for more complete vision pipelines, FPGAs outperform GPUs and CPUs (1.2–22.3× energy/frame reduction). Moreover, FPGAs perform increasingly better as the complexity of vision pipelines grow. This is evident by the energy-delay product, a metric that not only takes into account the energy/frame, but also the throughput. The FPGA is 2.1–2.9× faster and 1.1–2.4× more energy efficient than GPU F16 implementations when running Inception-v2, ResNet-50, ResNet-18, Mobilenetv2 and SqueezeNet.

Our future work will update this analysis to the latest platform generations, like Nvidia's recently released AGX board, and will include more vision kernels and neural networks. We will also investigate instantiating multiple instances of single vision kernels using xfOpenCV and compare it with GPUs. Additionally, we will extend this benchmarking analysis to include popular neural processing units (NPUs) in mobile processors such as iPhone A13 Bionic, Samsung Exynos, Qualcomm Snapdragon, etc.

Table 6: FPGA's Speedup and EDP Reduction Ratios with Respect to GPU FP16 when [b=128 and t=8]

Model	Frame Rate (fps)	EDP (mJ.s/f2)
Inception-v2	2.5×	1.5×
ResNet-50	2.1×	1.1×
ResNet-18	2.6×	1.4×
Mobilenet-v2	2.9×	2.4×
SqueezeNet	2.5×	1.7×

References

[1] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, S. Iyengar, A survey on deep learning: Algorithms, techniques, and applications, ACM Computing Surveys (CSUR) 51 (5) (2018) 92.

[2] M. H. Ionica, D. Gregg, The movidius myriad architecture's potential for scientific computing, IEEE Micro 35 (1) (2015) 6–14.

[3] S. Collange, D. Defour, A. Tisserand, Power consumption of gpus from a software perspective, in: International Conference on Computational Science, Springer, 2009, pp. 914–923.

[4] H. Giefers, P. Staar, C. Bekas, C. Hagleitner, Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga, in: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2016, pp. 46–56.

[5] S. Che, J. Li, J. W. Sheaffer, K. Skadron, J. Lach, Accelerating compute-intensive applications with gpus and fpgas, in: Application Specific Processors, 2008. SASP 2008. Symposium on, IEEE, 2008, pp. 101–107.

[6] N. O'Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan, J. Walsh, Deep learning vs. traditional computer vision, in: K. Arai, S. Kapoor (Eds.), Advances in Computer Vision, Springer International Publishing, Cham, 2020, pp. 128–144.

[7] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, S. Zhang, Understanding performance differences of fpgas and gpus, in: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2018, pp. 93–96.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, Ieee, 2009, pp. 44–54.

[9] P. Cooke, J. Fowers, G. Brown, G. Stitt, A tradeoff analysis of fpgas, gpus, and multicores for sliding-window applications, ACM Transactions on Reconfigurable Technology and Systems (TRETS) 8 (1) (2015) 2.

[10] J. Fowers, G. Brown, P. Cooke, G. Stitt, A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications, in: Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, ACM, 2012, pp. 47–56.

[11] C. Brugger, L. Dal'Aqua, J. A. Varela, C. De Schryver, M. Sadri, N. Wehn, M. Klein, M. Siegrist, A quantitative cross-architecture study of morphological image processing on cpus, gpus, and fpgas, in: Computer Applications & Industrial Electronics (ISCAIE), 2015 IEEE Symposium on, IEEE, 2015, pp. 201–206.

[12] E. Fykse, Performance comparison of gpu, dsp and fpga implementations of image processing and computer vision algorithms in embedded systems, Master's thesis, Institutt for elektronikk og telekommunikasjon (2013).

[13] M. Blott, L. Halder, M. Lesser, L. Doyle, Qutibench: Benchmarking neural networks on heterogeneous hardware, ACCEPTED to be published (2019).

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, ImageNet: A Large-Scale Hierarchical Image Database, in: CVPR09, 2009.

[15] Y. Liu, H. Zhang, L. Zeng, W. Wu, C. Zhang, Mlbench: Benchmarking machine learning services against human experts, in: Proceedings of the VLDB Endowment (VLDB 2018), proceedings of the vldb endowment (vldb 2018) Edition, 2018.

[16] Deepbench: Benchmarking deep learning operations on different hardware, https://svail.github.io/DeepBench/, accessed: 2019-08-17.

[17] Spec 2018. standard performance evaluation corporation, http://spec.org, accessed: 2019-08-17.

[18] J. D. McCalpin, Stream: Sustainable memory bandwidth in high performance computers. (2018), https://www.cs.virginia.edu/stream/, accessed: 2019-08-17.

[19] Arm neon technology, https://developer.arm.com/architectures/instruction-sets/simd-isas/neon, accessed: 2019-08-17.

[20] Intel instruction set extensions technology, https://www.intel.com/content/www/us/en/support/articles/000005779/processors.htmln, accessed: 2019-08-17.

[21] G. Bradski, Open source computer vision library, https://opencv.org/opencv-4-0-0.html (2018).

[22] NVIDIA., Nvidia visionworks toolkit, https://developer.nvidia.com/embedded/visionworks (2018).

[23] Xilinx., xfopencv library functions., https://www.xilinx.

com/support/documentation/sw_manuals/xilinx2018_3/ug1233-xilinx-opencv-user-guide.pdf (2018).

[24] Deep neural networks (dnn module), https://docs.opencv.org/master/d2/d58/tutorial_table_of_content_dnn.html, accessed: 2019-08-17.

[25] NVIDIA., Nvidia tensorrt: Programmable inference accelerator, https://developer.nvidia.com/tensorrt (2019).

[26] Xilinx vitis ai, https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, accessed: 2020-04-17.

[27] Openvino toolkit documentation, https://docs.openvinotoolkit.org/latest/index.html, accessed: 2020-04-17.

[28] Software developer kit (sdk), https://developer.arm.com/ip-products/processors/machine-learning/arm-nn, accessed: 2020-04-17.

[29] fast-dnn, https://github.com/ahmetaa/fast-dnn, accessed: 2020-04-17.

[30] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 1–9.

[31] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[32] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L.-C. Chen, Mobilenetv2: Inverted residuals and linear bottlenecks, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 4510–4520.

[33] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, K. Keutzer, Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size, arXiv preprint arXiv:1602.07360 (2016).

[34] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, K. Bernstein, Scaling, power, and the future of cmos, in: Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International, IEEE, 2005.

[35] Computer vision overlays on pynq, https://github.com/Xilinx/PYNQ-ComputerVision, accessed: 2018-12-17.

[36] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, R. Wittig, Ultrascale+ mpsoc and fpga families, in: Hot Chips 27 Symposium (HCS), 2015 IEEE, IEEE, 2015, pp. 1–37.

[37] Background subtraction, https://docs.opencv.org/3.4/db/d5c/tutorial_py_bg_subtraction.html, accessed: 2018-12-17.

Table 7: Inference Results ResNet50

ResNet50		Accuracy [%]	Latency [ms]		Throughput [fps]		Power [W]	Energy/Frame (mJ/f)		EDP (mJ.s/f2)	
Platform	**Parameters**	**Top-5 (Top-1)**	**system**	**compute**	**system**	**compute**		**system**	**compute**	**system**	**compute**
ZCU102	INT8, t=1	90.85 (72.53)	17.78	14.82	43.75	67.64	9.3	212.6	137.5	4.86	2.03
ZCU102	INT8, t=2	90.85 (72.53)	18.59	15.57	107.37	127.66	12.1	112.7	94.8	1.05	0.74
ZCU102	INT8, t=3	90.85 (72.53)	20.69	17.61	120.67	169.74	14.05	116.4	82.8	0.96	0.49
ZCU102	INT8, t=4	90.85 (72.53)	24.62	21.52	161.76	183.81	15.19	93.9	82.6	0.58	0.45
ZCU102	INT8, t=5	90.85 (72.53)	29.96	27.05	165.99	193.02	15.13	91.2	78.4	0.55	0.41
ZCU102	INT8, t=6	90.85 (72.53)	35.5	32.48	167.64	193.79	15.59	93.0	80.4	0.55	0.41
ZCU102	INT8, t=7	90.85 (72.53)	41.59	38.55	168.13	190.79	15.56	92.5	81.6	0.55	0.43
ZCU102	INT8, t=8	90.85 (72.53)	47.61	44.41	167.29	191.91	15.53	92.8	80.9	0.55	0.42
TX2, MaxN	FP16, b=1	92.12 (75.11)	13.99	10.68	70.96	93.94	13.62	191.9	145	2.7	1.54
TX2, MaxN	FP16, b=2	92.12 (75.11)	23.65	18.25	84.38	110.83	14.0	165.9	126.3	1.97	1.14
TX2, MaxN	FP16, b=4	92.12 (75.11)	43.79	34.23	91.16	118.09	14.08	154.5	119.2	1.69	1.01
TX2, MaxN	FP16, b=8	92.12 (75.11)	84.79	65.86	94.33	121.41	14.23	150.9	117.2	1.6	0.97
TX2, MaxN	FP16, b=16	92.12 (75.11)	162.58	126.23	98.41	126.5	14.65	148.9	115.8	1.51	0.92
TX2, MaxN	FP16, b=32	92.12 (75.11)	317.87	247.34	100.91	129.51	14.95	148.2	115.4	1.47	0.89
TX2, MaxN	FP16, b=64	92.12 (75.11)	620.08	490.37	103.53	130.42	15.18	146.6	116.4	1.42	0.89
TX2, MaxN	FP16, b=128	92.12 (75.11)	1211.85	975.98	104.85	131.08	15.72	149.9	119.9	1.43	0.91
TX2, MaxN	FP32, b=1	92.11 (75.15)	22.32	18.97	44.67	52.79	14.8	331.3	280.4	7.42	5.31
TX2, MaxN	FP32, b=2	92.11 (75.15)	38.46	32.96	51.96	60.99	15.22	292.9	249.5	5.64	4.09
TX2, MaxN	FP32, b=4	92.11 (75.15)	72.96	62.96	54.8	63.68	15.49	282.7	243.2	5.16	3.82
TX2, MaxN	FP32, b=8	92.11 (75.15)	141.13	122.18	56.67	65.55	15.49	273.3	236.3	4.82	3.60
TX2, MaxN	FP32, b=16	92.11 (75.15)	272.41	235.85	58.74	67.8	15.82	269.3	233.3	4.58	3.44
TX2, MaxN	FP32, b=32	92.11 (75.15)	531.12	460.67	60.29	69.46	16.24	269.4	233.8	4.47	3.37
TX2, MaxN	FP32, b=64	92.11 (75.15)	1042.67	913.42	61.3	69.93	17.15	279.8	245.2	4.56	3.51
TX2, MaxN	FP32, b=128	92.11 (75.15)	2115.54	1810.9	59.95	70.53	17.15	286.1	243.2	4.77	3.45
TX2, MaxQ	FP16, b=1	92.12 (75.11)	20.46	15.86	48.71	64.29	6.96	142.9	108.3	2.93	1.68
TX2, MaxQ	FP16, b=2	92.12 (75.11)	34.65	26.69	57.57	75.47	6.88	119.5	91.2	2.08	1.21
TX2, MaxQ	FP16, b=4	92.12 (75.11)	64.53	50.01	61.88	80.15	6.77	109.4	84.5	1.77	1.05
TX2, MaxQ	FP16, b=8	92.12 (75.11)	124.75	96.69	64.08	82.73	7.23	112.8	87.4	1.76	1.06
TX2, MaxQ	FP16, b=16	92.12 (75.11)	239	185.17	66.86	86.3	7.3	109.2	84.6	1.63	0.98
TX2, MaxQ	FP16, b=32	92.12 (75.11)	466.49	362.02	68.56	88.37	7.61	111.0	86.1	1.62	0.97
TX2, MaxQ	FP16, b=64	92.12 (75.11)	924.53	717.12	69.19	89.11	8.1	117.1	90.9	1.69	1.02
TX2, MaxQ	FP16, b=128	92.12 (75.11)	1838.48	1429	69.47	89.51	8.45	121.6	94.4	1.75	1.05
TX2, MaxQ	FP32, b=1	92.11 (75.15)	32.66	27.94	30.49	36.17	7.61	249.6	210.4	8.19	5.82
TX2, MaxQ	FP32, b=2	92.11 (75.15)	56.36	48.32	35.41	41.56	7.95	224.5	191.3	6.34	4.60
TX2, MaxQ	FP32, b=4	92.11 (75.15)	106.84	92.09	37.41	43.48	7.8	208.5	179.4	5.57	4.13
TX2, MaxQ	FP32, b=8	92.11 (75.15)	207.45	179.45	38.56	44.61	7.76	201.2	174	5.22	3.90
TX2, MaxQ	FP32, b=16	92.11 (75.15)	398.74	344.35	40.1	46.48	8.03	200.2	172.8	4.99	3.72
TX2, MaxQ	FP32, b=32	92.11 (75.15)	779.69	673.93	41.01	47.46	8.1	197.5	170.7	4.82	3.60
TX2, MaxQ	FP32, b=64	92.11 (75.15)	1540.33	1333.2	41.52	47.97	8.29	199.7	172.8	4.81	3.60
TX2, MaxQ	FP32, b=128	92.11 (75.15)	3118.09	2650.9	40.85	48.2	9.02	220.8	187.1	5.41	3.88
TX2, MaxP	FP16, b=1	92.12 (75.11)	16.52	12.46	60.12	81.87	9.36	155.7	114.3	2.59	1.40
TX2, MaxP	FP16, b=2	92.12 (75.11)	28.1	20.92	70.93	96.51	9.59	135.2	99.4	1.91	1.03
TX2, MaxP	FP16, b=4	92.12 (75.11)	52.22	39.14	76.53	102.41	9.59	125.3	93.6	1.64	0.91
TX2, MaxP	FP16, b=8	92.12 (75.11)	100.21	75.54	79.74	105.97	10.09	126.5	95.2	1.59	0.90
TX2, MaxP	FP16, b=16	92.12 (75.11)	193.89	145.36	82.51	110.08	10.13	122.8	92	1.49	0.84
TX2, MaxP	FP16, b=32	92.12 (75.11)	375.86	283.47	85.21	112.82	10.39	121.9	92.1	1.43	0.82
TX2, MaxP	FP16, b=64	92.12 (75.11)	733.74	562.74	87.2	113.88	11.34	130	99.6	1.49	0.87
TX2, MaxP	FP16, b=128	92.12 (75.11)	1453.87	1121.2	87.78	113.94	11.64	132.6	102.2	1.51	0.90
TX2, MaxP	FP32, b=1	92.11 (75.15)	26.16	22	38.03	46.07	10.66	280.3	231.4	7.37	5.02
TX2, MaxP	FP32, b=2	92.11 (75.15)	45.07	37.86	44.26	53.12	10.88	245.8	204.8	5.55	3.86
TX2, MaxP	FP32, b=4	92.11 (75.15)	85.31	72.25	46.88	55.54	11.19	238.7	201.5	5.09	3.63
TX2, MaxP	FP32, b=8	92.11 (75.15)	165.16	140.36	48.43	57.1	11.23	231.9	196.7	4.79	3.44
TX2, MaxP	FP32, b=16	92.11 (75.15)	318.38	270.54	50.22	59.16	11.42	227.4	193	4.53	3.26
TX2, MaxP	FP32, b=32	92.11 (75.15)	621.3	528.54	51.51	60.52	11.8	229.1	195	4.45	3.22
TX2, MaxP	FP32, b=64	92.11 (75.15)	1219.1	1046.8	52.38	61.08	12.52	239	205	4.56	3.36
TX2, MaxP	FP32, b=128	92.11 (75.15)	2495.53	2076.5	50.94	61.52	12.9	253.2	209.7	4.97	3.41

Table 8: Frame Rate (fps) of Different Vision Kernels on CPU, GPU and FPGA

Category	**Kernel**	**Frame Rate (fps)**			**L1-norm Error**	
		ARM	**GPU**	**FPGA**	**GPU**	**FPGA**
Image Arithmetic	absolute diff	904	3717	133	0.0	0.0
	accumulate	301	2262	127	0.0	0.0
	accumulate squared	281	2283	128	0.0	0.0
	accumulate weighted	286	3846	128	0.0	0.6
	arithmetic add	913	4065	128	0.0	0.0
	arithmetic subtract	884	4048	133	0.0	0.0
	arithmetic multiply	259	2985	132	0.0	0.55
	bitwise and, or, xor	879	3546	133	0.0	0.0
	bitwise not	1612	4424	128	0.0	0.0
	magnitude	227	1838	108	1.96	0.22
	phase	150	2222	108	1.23	0.23
	threshold	1996	3745	135	1.35	0.0
Input Processing	channel combine	881	2132	135	0.0	0.0
	channel split	443	2277	138	0.0	0.0
	color conversion	53	2147	132	0.0	0.0
	bit depth conversion	1141	3344	137	0.0	0.0
	table lookup	961	3058	32	0.0	0.0
Geometric Transforms	affine warp	69	2739	365	0.11	0.0
	perspective warp	36	2604	384	0.11	0.0
	resize	690	2857	350	-	0.35
	remap	33	996	1004	0.11	0.0
Filters	filter 2D	26	1285	134	0.0	0.0
	box filter	253	2824	135	0.0	0.36
	dilate	356	2638	134	6.4	0.0
	erode	359	2645	134	6.8	0.0
	median	323	858	121	0.0	0.0
	pyr Down	377	2150	108	0.45	0.0
	pyr Up	80	1375	104	0.40	0.43
Analysis	histogram	354	2164	139	0.0	0.0
	hist equalization	308	1349	69	0.0	0.05
	integral image	235	682	122	0.0	0.0
	mean std deviation	336	1277	141	0.0	0.0
	min max locations	128	734	140	0.0	0.0
Features	canny	64	101	99	N/A	N/A
	fast corner	169	339	49	N/A	N/A
	harris corner	11	501	113	0.0	0.0
	optical flow	18	64	5	0.0	0.0
	stereoBM	18	54	7	-	0.02