Optimizing Deep Learning Accelerators for Real-Time Triggering in the Cherenkov Telescope Array: A Comparative Study of FPGA, GPU, and CPU Performance

Author:
Tamer VAN TEEFFELEN

Supervisors:
Dr. Carlos ABELLAN
Dr. Iaroslava BEZSHYIKO

29/08/2024

Contents

1 Introduction

The Cherenkov Telescope Array (CTA) represents a groundbreaking international endeavor in the field of very high-energy gamma-ray astronomy, aiming to establish the world's most sophisticated ground-based observatory [1]. This ambitious project comprises an extensive network of Imaging Atmospheric Cherenkov Telescopes (IACTs) strategically distributed across multiple sites. The primary objective of these IACTs is to detect and analyze the faint Cherenkov light flashes generated when high-energy gamma rays interact with the Earth's atmosphere. Through this advanced detection system, the CTA seeks to provide unprecedented insights into some of the most energetic phenomena in the cosmos, including supernovae, black holes, and dark matter [1].

A critical component of the CTA's operational framework is its multi-tiered triggering system, which is responsible for discriminating valid Cherenkov events and initiating data recording processes. The triggering mechanism operates on two levels: local and global. At the local level, individual telescopes employ sophisticated algorithms to identify concentrations of Cherenkov light in their cameras. This is achieved either through thresholding pixel signals and detecting coincidences in adjacent pixels or by summing signals from pixel clusters and comparing the aggregate to a predetermined threshold [2, 3]. Events that satisfy the local trigger criteria are temporarily stored, and a notification is transmitted to a global array trigger operating at the software level. The global trigger then analyzes potential coincidences across multiple telescopes, and upon detection, initiates the transmission of event information for further processing [4]. This intricate triggering scheme enables the CTA to operate with lower telescope trigger thresholds while mitigating the risk of readout system saturation [2–4].

Recent advancements in the field of deep learning have presented novel opportunities for enhancing the CTA's triggering system. Convolutional Neural Networks (CNNs) have demonstrated significant potential in improving event selection through more sophisticated and accurate data filtering techniques [5]. Deep learning models, such as those developed within the CTLearn framework, are trained on simulated event reconstructions and have been successfully employed to autonomously extract features from raw Cherenkov images, thereby improving the accuracy of event detection [5]. The integration of these deep learning techniques into the triggering process holds promise for reducing false positives and enhancing the precision of gamma-ray event detection [6].

Traditionally, Graphics Processing Units (GPUs) have been the preferred hardware platform for deep learning inference due to their capacity to efficiently handle the parallel computations required by neural networks [7]. Their architecture is particularly well-suited for the matrix operations that form the core of CNNs, making them ideal for the real-time processing demands of projects such as the CTA [8]. However, despite the widespread adoption of GPUs, Field-Programmable Gate Arrays (FPGAs) have emerged as a compelling alternative, offering advantages such as lower power consumption, customizable architecture, and potential for real-time processing [9]. FPGAs present a promising solution for deploying CNNs within the CTA's triggering system, particularly in resource-constrained environments where energy efficiency and low latency are paramount.

This study aims to compare the performance of CNNs implemented on FPGAs, GPUs, and CPUs within the context of the CTA's final triggering system. The performance metrics of interest are throughput—the rate at which data can be processed—and latency—the delay between input and processing output. By evaluating these architectures based on these key metrics, this research seeks to identify the most effective hardware platform for deploying deep learning models in the CTA's real-time, high-stakes environment.

2 Methods

This section outlines the tools and methodologies used to evaluate the performance of deep neural networks on FPGA and GPU platforms, specifically focusing on throughput and latency metrics. The corresponding code as well as commands are also given.

2.1 FPGA Inference Using Intel's Deep Learning Accelerator

For FPGA inference, Intel's deep learning accelerator (DLA) plugin for the FPGA AI Suite optimizer was used to deploy neural networks and measure performance. Two tools within this plugin were utilized:

- The `dla_compiler` tool was used to estimate throughput on various FPGA architectures using randomly generated data. This provided a theoretical performance benchmark for each architecture.
- The `dla_benchmark` tool was employed to measure real-time throughput on the FPGA architectures, also using randomly generated data, offering empirical validation of the compiler's estimates.

2.2 FPGA Compilation and Optimization with Quartus

Intel's Quartus software was employed for the bitstream compilation and optimization of different FPGA architectures using Intel's FPGA AI Suite. The architectures used and their corresponding FPGA boards were:

- **Arria 10 (A10)**:
  - A10_Generic.arch
  - A10_Performance.arch
  - A10_FP16_Generic.arch
  - A10_FP16_Performance.arch
- **Agilex 7 (AGX7)**:
  - AGX7_Generic.arch
  - AGX7_Performance.arch
  - AGX7_FP16_Generic.arch
  - AGX7_FP16_Performance.arch

It should be noted that the architectures without the FP16 specification indicate an FP11 precision instead and are thus expected to yield higher throughput. For the Arria 10 FPGA, both compilation and real-time inference were performed. The Agilex 7 FPGA, being in its testing phase and not yet available for industrial use, was only subjected to compilation for performance estimation.

Furthermore, optimization for each FPGA was performed in accordance to the maximum resources available. For Arria 10, the maximum available resources are 427,200 Adaptive Logic Modules (ALMs), 2,713 M20K memory blocks, and 1,518 DSP blocks. For the Agilex 7 FPGA, the AFG014 variant has 487,200 ALMs, 7,110 M20K blocks, and 4,510 DSP blocks, while the AGM039 variant offers 1,305,600 ALMs, 18,960 M20K blocks, and 12,300 DSP blocks. The assumed maximum core frequencies were set at 300 MHz for Arria 10 and 600 MHz for Agilex 7.

2.3 GPU and CPU Inference

Inference was conducted using an NVIDIA L40S GPU and an AMD EPYC 9254 CPU. To ensure the reliability of the results, latency was measured by averaging the processing time of over 10,000 samples, while throughput was calculated by averaging the rate at which these samples were processed.

2.4 Commands

2.4.1 Configuration

To execute the FPGA scripts effectively, it is necessary to configure the software environment by initializing the required tools and setting up the appropriate environment variables. The following commands must be run sequentially to ensure the proper configuration:

```
source /opt/openvino/openvino_2022.3.1_env/bin/activate
source /opt/intel/inteldevstack/init_env_user.sh
unset python_version
source /opt/intel/openvino_2022/setupvars.sh
source /opt/intel/fpga_ai_suite_2024.1/dla/bin/init_env.sh
source $HOME/coredla_work/coredla_work.sh
```

2.4.2 Model conversion

Before deploying the models on the FPGA, it is necessary to convert them to a format compatible with OpenVINO. The following Python script accomplishes this by converting the models and setting the appropriate tensor layouts:

```
import openvino as ov

dirname = "ctlearn_model"

# Specify that batch size is 1 because dynamic tensor sizes are not supported on
    FPGA
ov_model = ov.convert_model(dirname, input=[1,30,30,5])

input_name = list(ov_model.inputs[0].names)[0]
prep = ov.preprocess.PrePostProcessor(ov_model)

```

```
# Specify channel layout---batch (N) is 1, (H)eight is 30, (W)idth is 30, (C)
    hannels is 5
prep.input(input_name).tensor().set_layout(ov.Layout("NHWC"))

ov_model = prep.build()

# OpenVINO 2023.3 default behavior is to compress weights to fp16. We specify
    that we do not want that
ov.save_model(ov_model, 'ctlearn_model.xml', compress_to_fp16=False)
```

2.4.3 Compilation

To estimate the throughput and area usage of the FPGA, the `dla_compiler` tool is employed. This tool provides a comprehensive analysis of the performance and resource utilization based on the network model specified. The basic command to perform this estimation is as follows:

```
dla_compiler --march "${ARCH_PATH}/${arch}" --network-file "$network_file" \
--foutput-format=open_vino_hetero --o "$output_file" --batch-size=1 \
--fanalyze-area --fanalyze-performance $opt_params
```

In this command:

- `--march "$ARCH_PATH/$arch"` specifies the architecture file that defines the FPGA hardware configuration.
- `--network-file "$network_file"` indicates the path to the network model file that will be analyzed.
- `--foutput-format=open_vino_hetero` sets the output format to be compatible with OpenVINO for heterogeneous execution.
- `--o "$output_file"` defines the output file where the compiled model will be saved.
- `--batch-size=1` configures the batch size for the model, which is set to 1 in this case.
- `--fanalyze-area` enables the analysis of area utilization on the FPGA.
- `--fanalyze-performance` activates the performance analysis to estimate the throughput of the FPGA.
- `$opt_params` includes any additional optional parameters such as optimizations.

In scenarios where one desires to fully utilize the maximum resources available on the FPGA, optional parameters can be added to the command. These parameters specify the allocation of Adaptive Logic Modules (ALMs), M20K memory blocks, and DSP blocks, as well as additional optimization settings. The command with these optimizations is given below:

```
--gen-arch --mmax-resources=$ALMs,$M20Ks,$DSPs --gen-min-sb=2048 \
--mmax-resources-alm-util=75 --fassumed-fmax-core=300
```

Including these parameters not only optimizes the performance but also generates an optimized architecture that is tailored to the specific resource constraints and performance goals of the FPGA deployment.

2.4.4 Programming the FPGAs

In order to perform inference on the FPGA, it is necessary to program the bitstream onto the FPGA device first. This ensures that the FPGA is configured correctly for the specific design example. Below are the detailed steps to program the FPGA for different devices.

To program the DE10-Agilex-B2E2 board required for the PCIe-based design example for Agilex™ 7 devices, one must:

1. Build the runtime with the following commands:

```
cd $COREDLA_WORK/runtime
rm -rf build_Release
./build_runtime.sh -de10_agilex
```

2. Program the bitstream onto the FPGA device with the following commands:

```
jtagdir=$COREDLA_WORK/runtime/build_Release/fpga_jtag_reprogram
bitsdir=$COREDLA_WORK/demo/bitstreams
$jtagdir/fpga_jtag_reprogram $bitsdir/AGX7_Performance.sof
curarch=$COREDLA_ROOT/example_architectures/AGX7_Performance.arch
```

To program the Intel® PAC with Arria® 10 GX FPGA hardware required for the PCIe-based design example for Arria® 10 devices, we use the following steps:

1. Build the runtime with the following commands:

```
cd $COREDLA_WORK/runtime
rm -rf build_Release
./build_runtime.sh
```

2. Program the bitstream onto the FPGA device with the following commands:

```
fpgaconf -v $COREDLA_WORK/demo/bitstreams/A10_Performance.gbs
curarch=$COREDLA_ROOT/example_architectures/A10_Performance.arch
```

2.4.5 Inference

Once the FPGA has been programmed with the appropriate bitstream, inference can be performed using the `dla_benchmark` tool. This tool allows users to benchmark the performance of neural network models on the FPGA by specifying various parameters related to batch size, model path, and device architecture. The command to run inference is as follows:

```
$COREDLA_WORK/runtime/build_Release/dla_benchmark/dla_benchmark \
-b=1 -m "$model_path" -d=HETERO:FPGA,CPU -niter=8 \
-plugins_xml_file "$COREDLA_WORK/runtime/plugins.xml" \
-arch_file "${ARCH_PATH}/${arch}.arch" -api=async -perf_est -nireq=4 -bgr
```

In this command:

- `-b=1` specifies the batch size for inference.
- `-m "$model_path"` indicates the path to the model file to be used for inference.

- `-d=HETERO:FPGA,CPU` designates the heterogeneous execution across FPGA and CPU devices.
- `-niter=8` sets the number of iterations for the benchmark.
- `-plugins_xml_file "$COREDLA_WORK/runtime/plugins.xml"` specifies the plugins XML file required for the FPGA runtime.
- `-arch_file "$ARCH_PATH/$arch.arch"` points to the architecture file that matches the FPGA configuration.
- `-api=async` runs the benchmark in asynchronous mode for improved performance.
- `-perf_est` enables performance estimation during the benchmarking process.
- `-nireq=4` sets the number of inference requests to be processed simultaneously.
- `-bgr` specifies that the input data format is BGR (blue, green, red).

2.4.6 Compiling Bitstream for Optimized Architectures

When one is interested in compiling a bitstream after generating an optimized architecture, it is necessary to build the FPGA bitstream to deploy the PCIe-based design example on Arria 10 devices. This process ensures that the FPGA is configured according to the specified architecture. The following command is used to build the bitstream:

```
cd $COREDLA_WORK/demo
dla_build_example_design.py \
-a $COREDLA_ROOT/example_architectures/A10_Performance.arch \
--build -ed 1 \
--build-dir build_A10_Performance \
--output-dir $COREDLA_WORK/demo/my_bitstreams \
--n 1
```

In this command:

- `cd $COREDLA_WORK/demo` navigates to the demo directory where the bitstream will be built.
- `dla_build_example_design.py` is the script used to compile the design example.
- `-a $COREDLA_ROOT/example_architectures/A10_Performance.arch` specifies the path to the optimized architecture file generated for the Arria® 10 device.
- `--build -ed 1` initiates the build process for the specified design example.
- `--build-dir build_A10_Performance` sets the directory where the build files will be stored.
- `--output-dir $COREDLA_WORK/demo/my_bitstreams` designates the output directory where the compiled bitstream will be saved.
- `--n 1` ensures that a single instance of the architecture is built, which is crucial to avoid errors related to insufficient DSP resources.

It is important to include the `-n 1` option to build a single instance of the architecture. Without this option, one is likely to encounter an error due to insufficient DSP resources. If this error persists, it may be necessary to adjust the compilation process by lowering the `mmax-resources-alm-util` to a lower value (sometimes as low as 20 or 10) or by reducing the `mmax-resources` number for the DSPs during the generation of the optimized architecture.

2.4.7 Power usage

To monitor and evaluate the power consumption of the FPGA and GPU during inference, one must run the following commands: (1) For FPGA devices, the power usage can be checked by using the `fpgainfo` command. (2) For GPU devices, the power usage can be monitored using the `nvidia-smi` tool.

2.5 FPGA Throughput, Latency and Area Estimate Scripts

The following is a script that finds the throughput and area estimates of all architectures across all models including and excluding optimizations of Arria 10 and Agilex 7. The results are stored in a file titled “summary.csv”.

```
#!/bin/bash

# Define paths
BASE_PATH="/home/tvanteef/coredla_work/LST-AI-trigger-FPGA"
ARCH_PATH="/opt/intel/fpga_ai_suite_2024.1/dla/example_architectures"
MODELS=("finaltrigger_cnn_8_16_fch_32_GlobalAvgPool_IB" "
    testing_finaltrigger_cnn_8_16_fch_64_16" "
    finaltrigger_cnn_8_16_fch_32_GlobalMaxPool" "
    testing_finaltrigger_cnn_16_32_fch_128_32" "
    finaltrigger_cnn_8_16_fch_32_GlobalMaxPool_noNSB" "
    testing_finaltrigger_cnn_32_32_64_fch_256_64")

# Architecture arrays
A10_ARCHS=("A10_FP16_Generic.arch" "A10_FP16_Performance.arch" "A10_Generic.arch
    " "A10_Performance.arch")
AGX7_ARCHS=("AGX7_FP16_Generic.arch" "AGX7_FP16_Performance.arch" "AGX7_Generic.
    arch" "AGX7_Performance.arch")

# Optimized commands for A10
A10_OPTIMIZED="--gen-arch --mmax-resources=427200,2713,1518 --gen-min-sb=2048 --
    mmax-resources-alm-util=75 --fassumed-fmax-core=300"

# Optimized commands for AGX7
AGX7_OPTIMIZED_1="--gen-arch --mmax-resources=487200,7110,4510 --gen-min-sb=2048
     --mmax-resources-alm-util=75 --fassumed-fmax-core=600"
AGX7_OPTIMIZED_2="--gen-arch --mmax-resources=1305600,18960,12300 --gen-min-sb
    =2048 --mmax-resources-alm-util=75 --fassumed-fmax-core=600"

# Output CSV file
CSV_FILE="$(pwd)/summary.csv"
echo "Model,Architecture,Optimization,Final Throughput (fps),ALMs,ALUTs,
    Registers,DSPs,M20Ks,Memory ALMs" > $CSV_FILE

# Function to compile models
compile_model() {
  local model=$1
  local arch=$2
  local opt_params=$3
  local opt_label=$4

  local network_file="${BASE_PATH}/${model}/ctlearn_model/Conversion/
      ctlearn_model.xml"
```

```
  local output_dir="${BASE_PATH}/Performance/${model}/${arch}"
  mkdir -p $output_dir

  local output_file="${output_dir}/ctlearn_model_${opt_label}.out.bin"
  local log_file="${output_dir}/output_${opt_label}.txt"

  dla_compiler --march "${ARCH_PATH}/${arch}" --network-file "$network_file" --
      foutput-format=open_vino_hetero --o "$output_file" --batch-size=1 --
      fanalyze-area --fanalyze-performance $opt_params > $log_file

  # Parse and store the results
  parse_and_store_results $log_file $model $arch $opt_label
}

# Function to parse log file and store results
parse_and_store_results() {
  local log_file=$1
  local model=$2
  local arch=$3
  local opt_label=$4

  local throughput=$(grep -m 1 "FINAL THROUGHPUT =" $log_file | awk '{print $4}'
      )
  local ALMs=$(grep -m 1 "ALMs:" $log_file | awk '{print $2}')
  local ALUTs=$(grep -m 1 "ALUTs:" $log_file | awk '{print $2}')
  local Registers=$(grep -m 1 "Registers:" $log_file | awk '{print $2}')
  local DSPs=$(grep -m 1 "DSPs:" $log_file | awk '{print $2}')
  local M20Ks=$(grep -m 1 "M20Ks:" $log_file | awk '{print $2}')
  local Memory_ALMs=$(grep -m 1 "Memory ALMs:" $log_file | awk '{print $3}')

  echo "$model,$arch,$opt_label,$throughput,$ALMs,$ALUTs,$Registers,$DSPs,$M20Ks
      ,$Memory_ALMs" >> $CSV_FILE
}

# Iterate over models and architectures
for model in "${MODELS[@]}"; do
  # A10 architectures
  for arch in "${A10_ARCHS[@]}"; do
    compile_model $model $arch "" "regular"
    compile_model $model $arch "$A10_OPTIMIZED" "optimized"
  done

  # AGX7 architectures
  for arch in "${AGX7_ARCHS[@]}"; do
    compile_model $model $arch "" "regular"
    compile_model $model $arch "$AGX7_OPTIMIZED_1" "optimized1"
    compile_model $model $arch "$AGX7_OPTIMIZED_2" "optimized2"
  done
done

echo "Compilation completed. Summary stored in $CSV_FILE."
```

Listing 1: Throughput and Area Estimates of Arria 10 and Agilex 7 for all models and architectures.

The next script does the same as the above but for the ResNet-50 model for comparison.

```
#!/bin/bash

```

```
# Define paths
ARCH_PATH="/opt/intel/fpga_ai_suite_2024.1/dla/example_architectures"
OUTPUT_BASE_PATH="."  # Use current directory for output
MODEL="resnet50_model"  # Use only ResNet-50 model

# Architecture arrays
A10_ARCHS=("A10_FP16_Generic.arch" "A10_FP16_Performance.arch" "A10_Generic.arch" "A10_Performance.arch")
AGX7_ARCHS=("AGX7_FP16_Generic.arch" "AGX7_FP16_Performance.arch" "AGX7_Generic.arch" "AGX7_Performance.arch")

# Optimized commands for A10
A10_OPTIMIZED="--gen-arch --mmax-resources=427200,2713,1518 --gen-min-sb=2048 --mmax-resources-alm-util=75 --fassumed-fmax-core=300"

# Optimized commands for AGX7
AGX7_OPTIMIZED_1="--gen-arch --mmax-resources=487200,7110,4510 --gen-min-sb=2048 --mmax-resources-alm-util=75 --fassumed-fmax-core=600"
AGX7_OPTIMIZED_2="--gen-arch --mmax-resources=1305600,18960,12300 --gen-min-sb=2048 --mmax-resources-alm-util=75 --fassumed-fmax-core=600"

# Output CSV file
CSV_FILE="$(pwd)/resnet_summary.csv"
echo "Model,Architecture,Optimization,Final Throughput (fps),ALMs,ALUTs,Registers,DSPs,M20Ks,Memory ALMs" > $CSV_FILE

# Function to compile the model
compile_model() {
  local model=$1
  local arch=$2
  local opt_params=$3
  local opt_label=$4
  local network_file="/home/tvanteef/model_performance/graph.xml"  # Path to ResNet-50 model
  local output_dir="${OUTPUT_BASE_PATH}/${model}/${arch}"
  mkdir -p $output_dir

  local output_file="${output_dir}/resnet50_${opt_label}.out.bin"
  local log_file="${output_dir}/output_resnet_${opt_label}.txt"

  # Run the compilation and capture output
  dla_compiler --march "${ARCH_PATH}/${arch}" --network-file "$network_file" --foutput-format=open_vino_hetero --o "$output_file" --batch-size=1 --fanalyze-area --fanalyze-performance $opt_params > $log_file

  # Extract data from the log file
  local throughput=$(grep -m 1 "FINAL THROUGHPUT =" $log_file | awk '{print $4}')
  local ALMs=$(grep -m 1 "ALMs:" $log_file | awk '{print $2}')
  local ALUTs=$(grep -m 1 "ALUTs:" $log_file | awk '{print $2}')
  local Registers=$(grep -m 1 "Registers:" $log_file | awk '{print $2}')
  local DSPs=$(grep -m 1 "DSPs:" $log_file | awk '{print $2}')
  local M20Ks=$(grep -m 1 "M20Ks:" $log_file | awk '{print $2}')
  local Memory_ALMs=$(grep -m 1 "Memory ALMs:" $log_file | awk '{print $3}')

  # Append data to CSV file
  echo "$model,$arch,$opt_label,$throughput,$ALMs,$ALUTs,$Registers,$DSPs,$M20Ks
```

```
      ,$Memory_ALMs" >> $CSV_FILE
}

# Create the Output directory
mkdir -p "$OUTPUT_BASE_PATH"

# A10 architectures
for arch in "${A10_ARCHS[@]}"; do
  compile_model $MODEL $arch "" "regular"
  compile_model $MODEL $arch "$A10_OPTIMIZED" "optimized"
done

# AGX7 architectures
for arch in "${AGX7_ARCHS[@]}"; do
  compile_model $MODEL $arch "" "regular"
  compile_model $MODEL $arch "$AGX7_OPTIMIZED_1" "optimized1"
  compile_model $MODEL $arch "$AGX7_OPTIMIZED_2" "optimized2"
done

echo "Compilation completed. Results stored in $CSV_FILE."
```

Listing 2: Throughput and Area Estimates of Arria 10 and Agilex 7 for all architectures on ResNet-50.

The following script evaluates the realtime throughput and latency performance of the Barvinok model across different architectures and saves them to a file. This is done using `dla_benchmark`.

```
#!/bin/bash

# Define paths
BASE_PATH="/home/tvanteef/coredla_work/LST-AI-trigger-FPGA"
COREDLA_WORK="/home/tvanteef/coredla_work"
ARCH_PATH="/opt/intel/fpga_ai_suite_2024.1/dla/example_architectures"
MODEL="finaltrigger_cnn_8_16_fch_32_GlobalAvgPool_IB"
A10_ARCHS=("A10_FP16_Generic" "A10_FP16_Performance" "A10_Generic" "
    A10_Performance")
CSV_FILE="${BASE_PATH}/FPGA_benchmark_A10_smallmodel_Performance_results.csv"

# Function to run dla_benchmark and parse throughput and latency
run_and_parse_benchmark() {
  local model=$1
  local arch=$2
  local model_path="${BASE_PATH}/${model}/ctlearn_model/Conversion/ctlearn_model
      .xml"
  local log_file="${BASE_PATH}/Performance/${model}/${arch}/benchmark_output.txt
      "
  mkdir -p "$(dirname "$log_file")"

  fpgaconf -v $COREDLA_WORK/demo/bitstreams/${arch}.gbs
  curarch=$COREDLA_ROOT/example_architectures/${arch}.arch

  $COREDLA_WORK/runtime/build_Release/dla_benchmark/dla_benchmark -b=1 -m "
      $model_path" -d=HETERO:FPGA,CPU -niter=8 -plugins_xml_file "$COREDLA_WORK/
      runtime/plugins.xml" -arch_file "${ARCH_PATH}/${arch}.arch" -api=async -
      perf_est -nireq=4 -bgr > $log_file

  # Parse the first instance of throughput and latency from the log file
  local throughput=$(grep -m 1 'FINAL THROUGHPUT =' $log_file | awk '{print $4}'
      )
```

```
  local latency=$(grep -m 1 'latency' $log_file | awk '{print $2}')
  echo "$model,$arch,$throughput,$latency" >> $CSV_FILE
}

# Initialize CSV file
echo "Model,Architecture,Throughput (fps),Latency (ms)" > $CSV_FILE

# Iterate over A10 architectures
for arch in "${A10_ARCHS[@]}"; do
  run_and_parse_benchmark $MODEL $arch
done

echo "Benchmarking and parsing completed. Results saved to $CSV_FILE."
```

Listing 3: Realtime throughput and latency estimates for the Barvinok model across the Arria 10 architectures.

2.6 GPU and CPU Scripts

The following Python code calculates the throughput and latency estimates of the GPU through statistical means. One should set the sample size to a large value (1̃00,000) get an accurate throughput or power reading.

```
import tensorflow as tf
import numpy as np
import time
import statistics
import os
import csv
import logging as log
import sys

# Setup logging
log.basicConfig(format='[ %(levelname)s ] %(message)s', level=log.INFO, stream=
    sys.stdout)

# Limit TensorFlow to only use GPU2
gpus = tf.config.list_physical_devices('GPU')
if gpus and len(gpus) > 2:
    try:
        # Set only GPU2 (index 2) as visible
        tf.config.set_visible_devices(gpus[2], 'GPU')
        # Optionally, configure GPU memory growth
        tf.config.experimental.set_memory_growth(gpus[2], True)
        log.info("Using GPU2 only")
    except RuntimeError as e:
        # Visible devices must be set before GPUs have been initialized
        log.error(e)
else:
    log.error("GPU2 is not available")

# Define the input shape (30, 30, 5)
input_shape = (30, 30, 5)

# Generate random data according to the specified input shape
```

```
num_samples = 500000  # You can vary this number as needed
data = np.random.random((num_samples, *input_shape))

def eval_throughput(model, data):
    # Measure throughput
    start_time = time.time()
    model.predict(data)
    end_time = time.time()
    run_time = end_time - start_time
    num_samples = data.shape[0]
    throughput = num_samples / run_time

    return throughput

def eval_latency(model, data, seconds_to_run=10, niter=10):
    latencies = []
    start_time = time.time()
    time_point_to_finish = start_time + seconds_to_run
    sample_index = 0

    while time.time() < time_point_to_finish or len(latencies) < niter:
        sample = data[sample_index:sample_index+1]
        sample_start_time = time.time()
        model.predict(sample)
        sample_end_time = time.time()

        latencies.append((sample_end_time - sample_start_time) * 1000)  # Convert to ms

        sample_index = (sample_index + 1) % len(data)

    median_latency = statistics.median(latencies)
    average_latency = sum(latencies) / len(latencies)
    min_latency = min(latencies)
    max_latency = max(latencies)

    return median_latency, average_latency, min_latency, max_latency

def main():
    models = [
        'finaltrigger_cnn_8_16_fch_32_GlobalMaxPool',
        'finaltrigger_cnn_8_16_fch_32_GlobalMaxPool_noNSB',
        'finaltrigger_cnn_8_16_fch_32_GlobalAvgPool_IB',
        'testing_finaltrigger_cnn_16_32_fch_128_32',
        'testing_finaltrigger_cnn_32_32_64_fch_256_64',
        'testing_finaltrigger_cnn_8_16_fch_64_16'
    ]

    base_path = '/home/hep/tvanteef/LST-AI-trigger-FPGA/'
    results = []

    for model_name in models:
        model_path = os.path.join(base_path, model_name, 'ctlearn_model')
        if not os.path.exists(model_path):
            log.warning(f'Model file not found: {model_path}')
            continue

```

```
        log.info(f'Benchmarking model: {model_name}')

        # Load the model
        model = tf.keras.models.load_model(model_path)

        # Evaluate throughput
        throughput = eval_throughput(model, data)

        # Evaluate latency
        median_latency, avg_latency, min_latency, max_latency = eval_latency(
            model, data)

        results.append({
            'model': model_name,
            'throughput_fps': throughput,
            'median_latency_ms': median_latency,
            'average_latency_ms': avg_latency,
            'min_latency_ms': min_latency,
            'max_latency_ms': max_latency
        })

    # Save results to CSV
    output_file = 'benchmark_GPU_TensorFlow_results.csv'
    with open(output_file, 'w', newline='') as csvfile:
        fieldnames = ['model', 'throughput_fps', 'median_latency_ms', '
            average_latency_ms', 'min_latency_ms', 'max_latency_ms']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(results)

    log.info(f'Benchmark results saved to {output_file}')

if __name__ == '__main__':
    main()
```

Listing 4: Throughput and latency estimates of the GPU.

The analogue of the above code for the CPU can be found below.

```
import tensorflow as tf
import numpy as np
import time
import statistics
import os
import csv
import logging as log
import sys

# Setup logging
log.basicConfig(format='[ %(levelname)s ] %(message)s', level=log.INFO, stream=
    sys.stdout)

# Load the data
data = np.load('/home/hep/tvanteef/input_triggerpatch_waveforms_run73.npy')

# Ensure data has the correct shape
if data.shape[-1] > 5:
    data = data[..., :5]
```

```
elif data.shape[-1] < 5:
    raise ValueError("Data must have at least 5 channels for the models")

# Resize images to 30x30 if necessary
data_resized = tf.image.resize(data, [30, 30])

def eval_throughput(model, data):
    # Measure throughput
    start_time = time.time()
    model.predict(data)
    end_time = time.time()
    run_time = end_time - start_time
    num_samples = data.shape[0]
    throughput = num_samples / run_time

    return throughput

def eval_latency(model, data, seconds_to_run=10, niter=10):
    latencies = []
    start_time = time.time()
    time_point_to_finish = start_time + seconds_to_run
    sample_index = 0

    while time.time() < time_point_to_finish or len(latencies) < niter:
        sample = data[sample_index:sample_index+1]
        sample_start_time = time.time()
        model.predict(sample)
        sample_end_time = time.time()

        latencies.append((sample_end_time - sample_start_time) * 1000)  # Convert to ms

        sample_index = (sample_index + 1) % len(data)

    median_latency = statistics.median(latencies)
    average_latency = sum(latencies) / len(latencies)
    min_latency = min(latencies)
    max_latency = max(latencies)

    return median_latency, average_latency, min_latency, max_latency

def main():
    models = [
        'finaltrigger_cnn_8_16_fch_32_GlobalMaxPool',
        'finaltrigger_cnn_8_16_fch_32_GlobalMaxPool_noNSB',
        'finaltrigger_cnn_8_16_fch_32_GlobalAvgPool_IB',
        'testing_finaltrigger_cnn_16_32_fch_128_32',
        'testing_finaltrigger_cnn_32_32_64_fch_256_64',
        'testing_finaltrigger_cnn_8_16_fch_64_16'
    ]

    base_path = '/home/hep/tvanteef/LST-AI-trigger-FPGA/'
    results = []

    for model_name in models:
        model_path = os.path.join(base_path, model_name, 'ctlearn_model')
        if not os.path.exists(model_path):
```

```
            log.warning(f'Model file not found: {model_path}')
            continue

        log.info(f'Benchmarking model: {model_name}')

        # Load the model
        model = tf.keras.models.load_model(model_path)

        # Ensure that the model runs on CPU
        with tf.device('/CPU:0'):
            # Evaluate throughput
            throughput = eval_throughput(model, data_resized)

            # Evaluate latency
            median_latency, avg_latency, min_latency, max_latency = eval_latency
                (model, data_resized)

        results.append({
            'model': model_name,
            'throughput_fps': throughput,
            'median_latency_ms': median_latency,
            'average_latency_ms': avg_latency,
            'min_latency_ms': min_latency,
            'max_latency_ms': max_latency
        })

    # Save results to CSV
    output_file = 'benchmark_CPU_TensorFlow_results.csv'
    with open(output_file, 'w', newline='') as csvfile:
        fieldnames = ['model', 'throughput_fps', 'median_latency_ms', '
            average_latency_ms', 'min_latency_ms', 'max_latency_ms']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(results)

    log.info(f'Benchmark results saved to {output_file}')

if __name__ == '__main__':
    main()
```

Listing 5: Throughput and latency estimates of the CPU.

The code below runs the ResNet-50 model on the GPU. Once again, a higher sample number yields a more accurate throughput reading.

```
import tensorflow as tf
import numpy as np
import time
import statistics
import csv

# Limit TensorFlow to only use GPU2
gpus = tf.config.list_physical_devices('GPU')
if gpus and len(gpus) > 2:
    try:
        # Set only GPU2 (index 2) as visible
        tf.config.set_visible_devices(gpus[2], 'GPU')
```

```
        # Optionally, configure GPU memory growth
        tf.config.experimental.set_memory_growth(gpus[2], True)
        print("Using GPU2 only")
    except RuntimeError as e:
        # Visible devices must be set before GPUs have been initialized
        print(e)
else:
    print("GPU2 is not available or does not exist. Using CPU.")

# Define the input shape (224, 224, 3) as required for ResNet-50
input_shape = (224, 224, 3)

# Generate random data according to the specified input shape
num_samples = 50000  # You can vary this number as needed
data = np.random.random((num_samples, *input_shape))

# Preprocess the images to match ResNet-50 input requirements
data_preprocessed = tf.keras.applications.resnet50.preprocess_input(data)

# Load ResNet-50 model from keras applications
model = tf.keras.applications.ResNet50(
    include_top=True,
    weights='imagenet',
    input_tensor=None,
    input_shape=input_shape,
    pooling=None,
    classes=1000
)

def eval_throughput(model, data):
    # Measure throughput
    start_time = time.time()
    model.predict(data)
    end_time = time.time()
    run_time = end_time - start_time
    num_samples = data.shape[0]
    throughput = num_samples / run_time

    return throughput

def eval_latency(model, data, seconds_to_run=10, niter=10):
    latencies = []
    start_time = time.time()
    time_point_to_finish = start_time + seconds_to_run
    sample_index = 0

    while time.time() < time_point_to_finish or len(latencies) < niter:
        sample = data[sample_index:sample_index+1]
        sample_start_time = time.time()
        model.predict(sample)
        sample_end_time = time.time()

        latencies.append((sample_end_time - sample_start_time) * 1000)  # Convert to ms

        sample_index = (sample_index + 1) % len(data)

```

```
    median_latency = statistics.median(latencies)
    average_latency = sum(latencies) / len(latencies)
    min_latency = min(latencies)
    max_latency = max(latencies)

    return median_latency, average_latency, min_latency, max_latency

# Function to reset TensorFlow device placement
def reset_device():
    tf.keras.backend.clear_session()
    if gpus and len(gpus) > 2:
        tf.config.set_visible_devices(gpus[2], 'GPU')  # Use GPU2
    else:
        tf.config.set_visible_devices([], 'GPU')

# Evaluate on GPU2 if available
if gpus and len(gpus) > 2:
    reset_device()
    with tf.device('/GPU:2'):
        print("\nEvaluating on GPU2...")
        model_gpu = tf.keras.applications.ResNet50(
            include_top=True,
            weights='imagenet',
            input_tensor=None,
            input_shape=input_shape,
            pooling=None,
            classes=1000
        )
        throughput_gpu = eval_throughput(model_gpu, data_preprocessed)
        median_latency_gpu, avg_latency_gpu, min_latency_gpu, max_latency_gpu =
            eval_latency(model_gpu, data_preprocessed)
        print(f"Throughput on GPU2: {throughput_gpu:.2f} samples/sec")
        print(f"Median Latency on GPU2: {median_latency_gpu:.4f} ms/sample")
        print(f"Average Latency on GPU2: {avg_latency_gpu:.4f} ms/sample")
        print(f"Min Latency on GPU2: {min_latency_gpu:.4f} ms/sample")
        print(f"Max Latency on GPU2: {max_latency_gpu:.4f} ms/sample")
else:
    throughput_gpu, median_latency_gpu, avg_latency_gpu, min_latency_gpu,
        max_latency_gpu = None, None, None, None, None

# Save the results to a file
output_file = 'benchmark_ResNet50_results.csv'
results = []

if throughput_gpu is not None:
    results.append({
        'device': 'GPU2',
        'throughput_samples_sec': throughput_gpu,
        'median_latency_ms': median_latency_gpu,
        'average_latency_ms': avg_latency_gpu,
        'min_latency_ms': min_latency_gpu,
        'max_latency_ms': max_latency_gpu
    })

# Write results to a CSV file
with open(output_file, 'w', newline='') as csvfile:
    fieldnames = ['device', 'throughput_samples_sec', 'median_latency_ms', '
```

```
        average_latency_ms', 'min_latency_ms', 'max_latency_ms']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(results)

print(f'\nBenchmark results saved to {output_file}')
```

Listing 6: Throughput and latency estimates for the ResNet-50 model on the GPU.

3 Results and Discussion

The neural networks evaluated in this study, listed in order of their number of parameters, are as follows: Barvinok 1 (2,179 parameters), Verba (52,867 parameters), Barvinok 2 (2,179 parameters), Malva (210,435 parameters), Barvinok 3 (2,146 parameters), and Topolia (193,571 parameters).

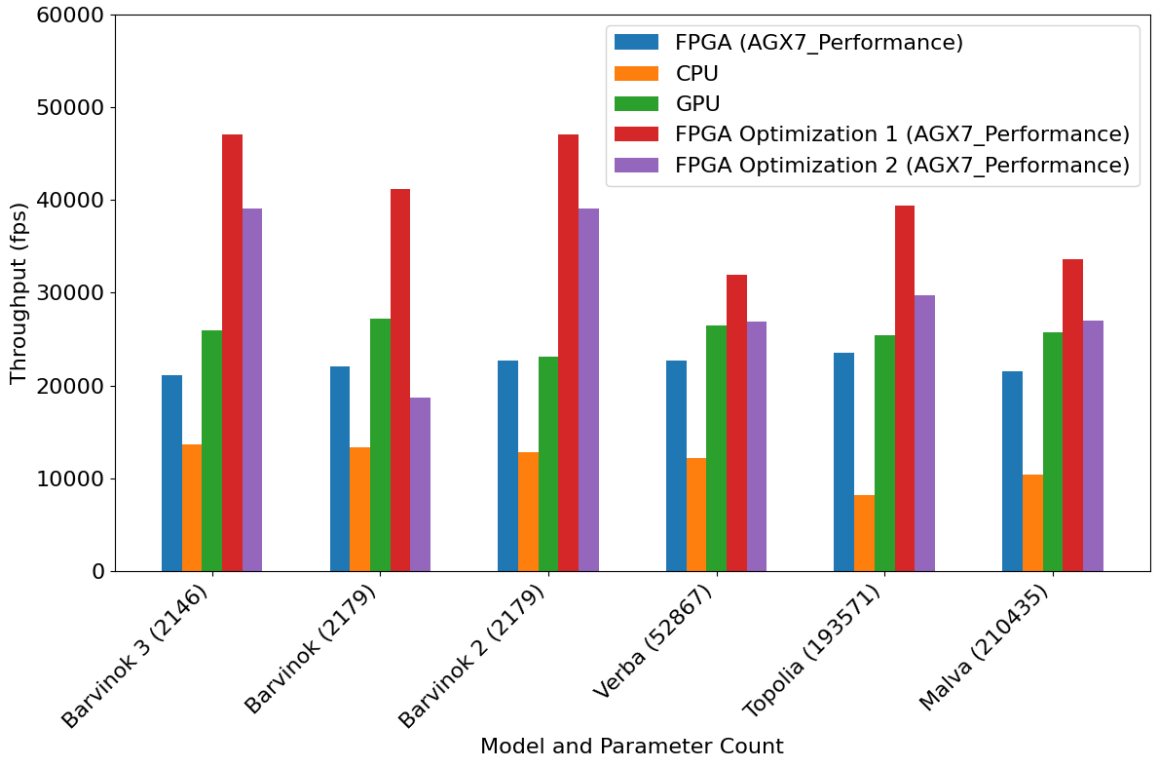
3.1 Model dependence

In this section, we analyze the dependence of throughput and latency on different neural network models across the evaluated hardware platforms, including CPUs, GPUs, and FPGAs. The results are illustrated in Figures 1 and 2.

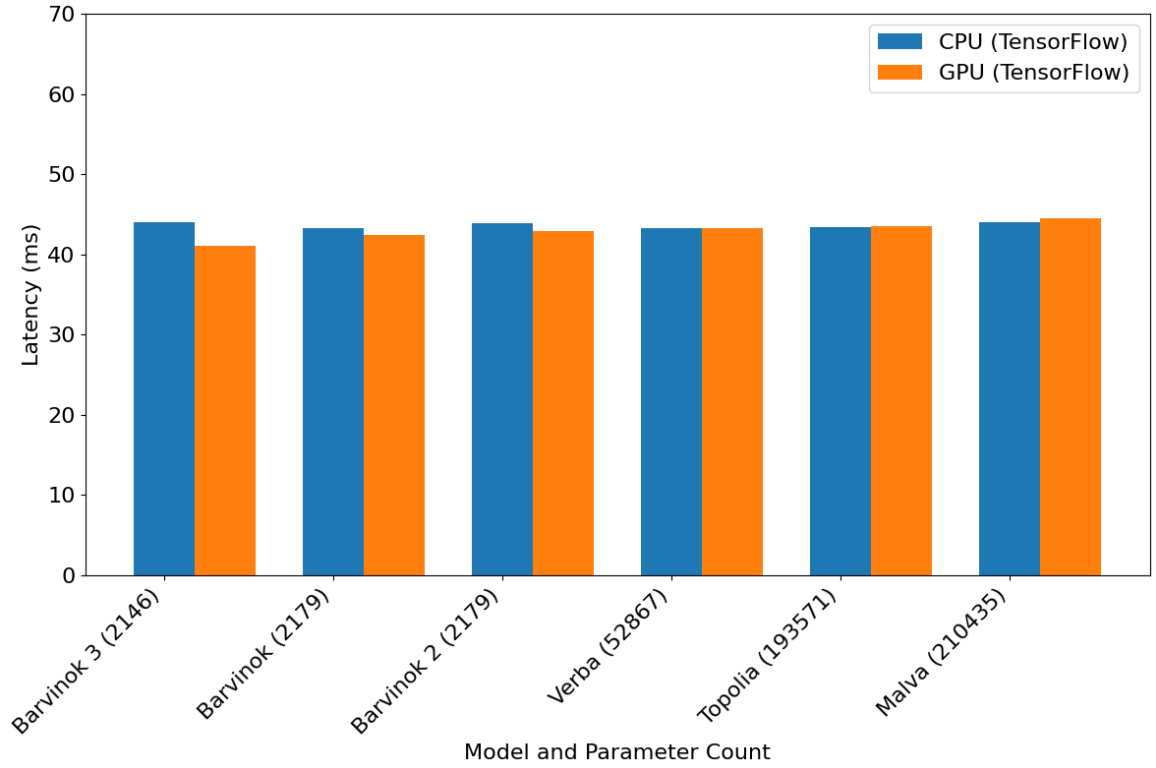
The results in Figure 1(a) indicate that the **Agilex 7 FPGA** exhibits the highest throughput among all platforms, particularly when optimized. This is especially evident in the significant performance gains observed with the Agilex 7 when compared to the GPU. Notably, Optimization 1 outperforms Optimization 2, despite Optimization 2 having access to a greater number of resources. This suggests that the larger configuration of the AGM039 variant may encounter scaling inefficiencies that could reduce performance. As a result, utilizing multiple cores or more sophisticated optimization strategies may be necessary to fully exploit the available resources in larger FPGA configurations.

The **Arria 10 FPGA**, as shown in Figure 2(a), demonstrates competitive performance, especially when optimized. In its unoptimized state, the Arria 10's throughput is on par with that of the GPU. However, after optimization, the Arria 10 is capable of matching or even surpassing GPU performance for certain models.

Latency measurements, depicted in Figures 1(b) and 2(b), reveal a significant advantage for the **Arria 10 FPGA**, which consistently outperforms both the CPU and GPU in terms of processing speed. In contrast, the latency figures for the CPU and GPU are quite similar.

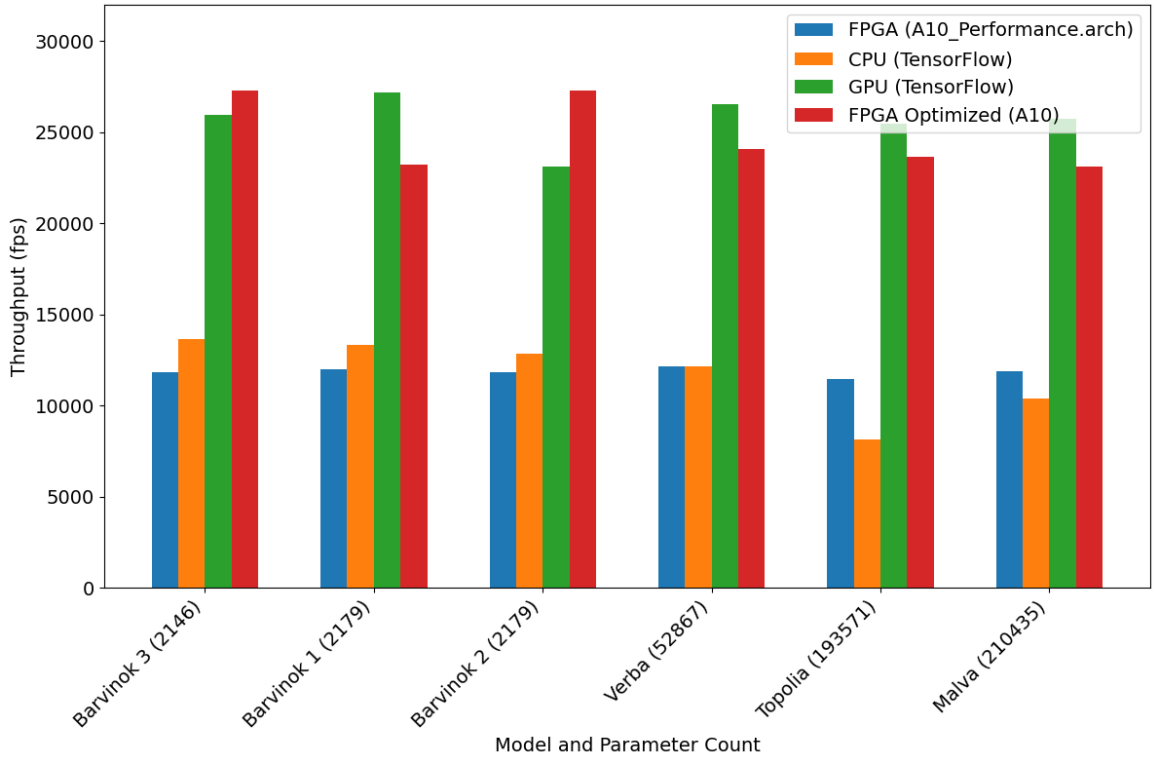


(a) Throughput vs. Model

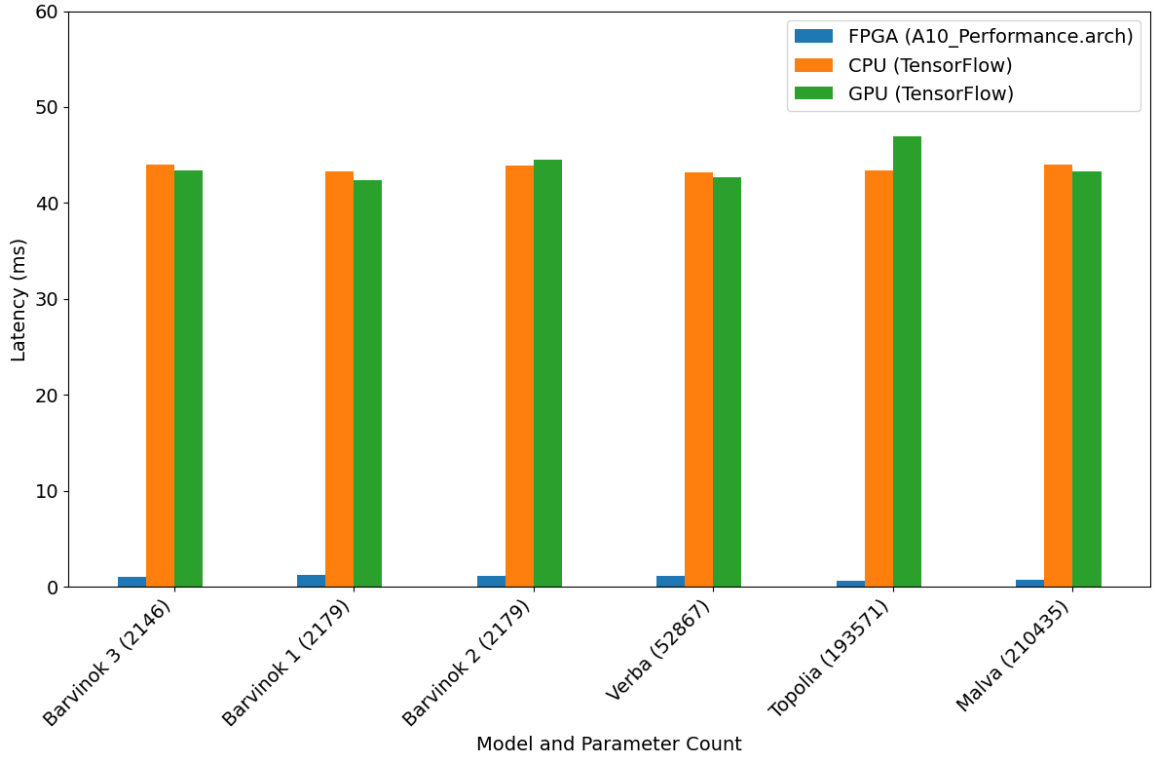


(b) Latency vs. Model

Figure 1: Performance comparison of the evaluated neural networks on the CPU, GPU, and Agilex 7 FPGA. (a) Throughput vs. Model and (b) Latency vs. Model. Optimization 1 corresponds to optimizing the AFG014 version of Agilex 7, while Optimization 2 corresponds to optimizing the AGM039 version. The FPGA values were computed using `dla_compiler` while the GPU and CPU values were using tensorflow.



(a) Throughput vs. Model



(b) Latency vs. Model

Figure 2: Performance comparison of the evaluated neural networks on the CPU, GPU, and Agilex 7 FPGA. (a) Throughput vs. Model and (b) Latency vs. Model. The FPGA values were computed using `dla_compiler` while the GPU and CPU values were using tensorflow. The latency calculation for Arria 10 was performed using `dla_benchmark` seeing that `dla_compiler` does not provide a latency estimate.

A comparison between the compiler estimates from `dla_compiler` and the real-time results from `dla_benchmark`, executed directly on the FPGA, reveals consistent similarities across the models. This consistency is illustrated in Figure 3, where the estimation was performed using the A10_Performance architecture.

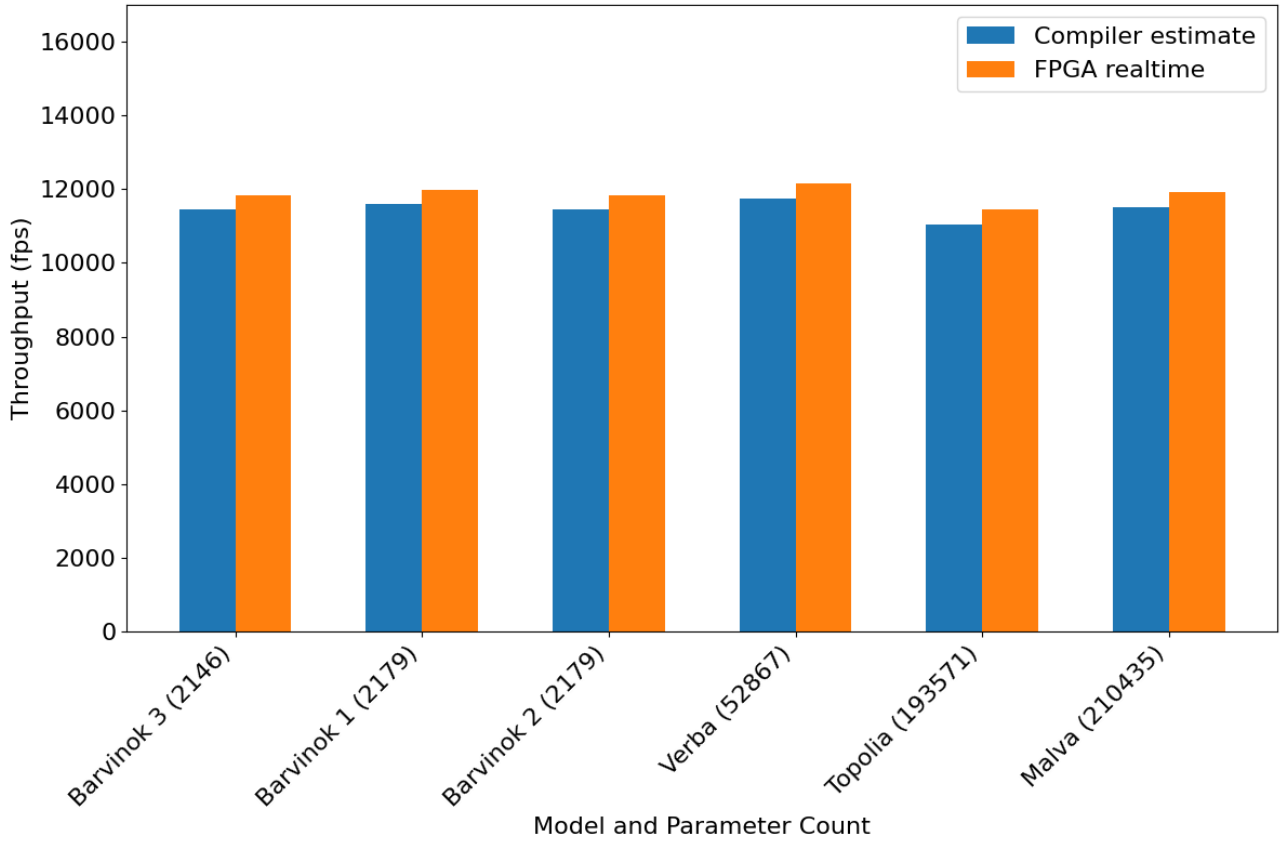
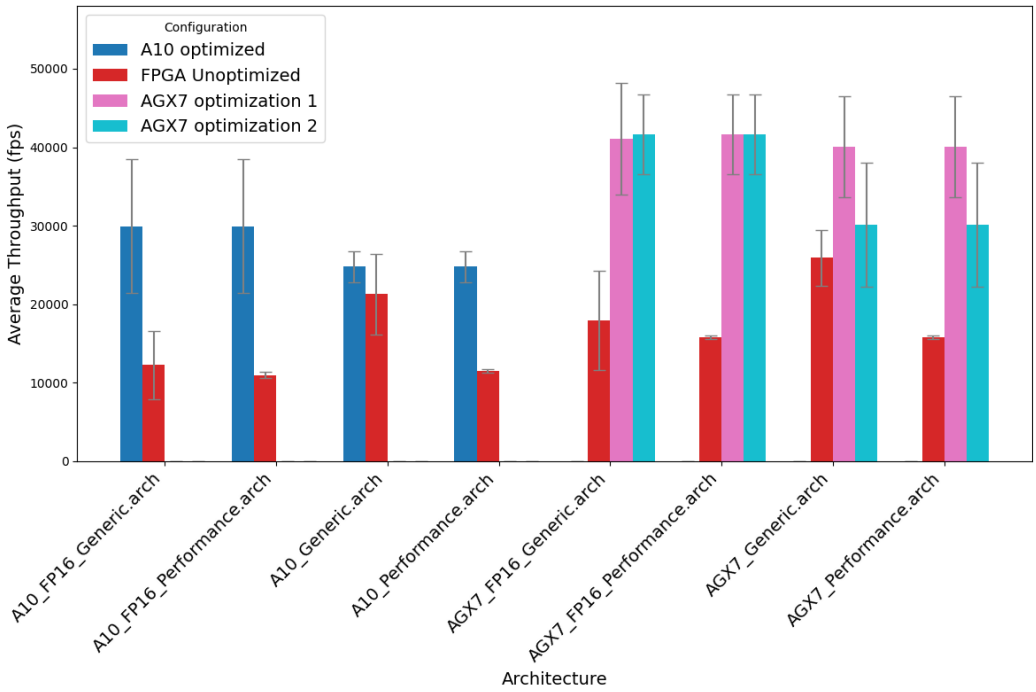


Figure 3: Throughput performance as measured by the compiler with `dla_compiler` and directly run on the Arria 10 FPGA with `dla_benchmark`. The estimates are given for the A10_Performance architecture.

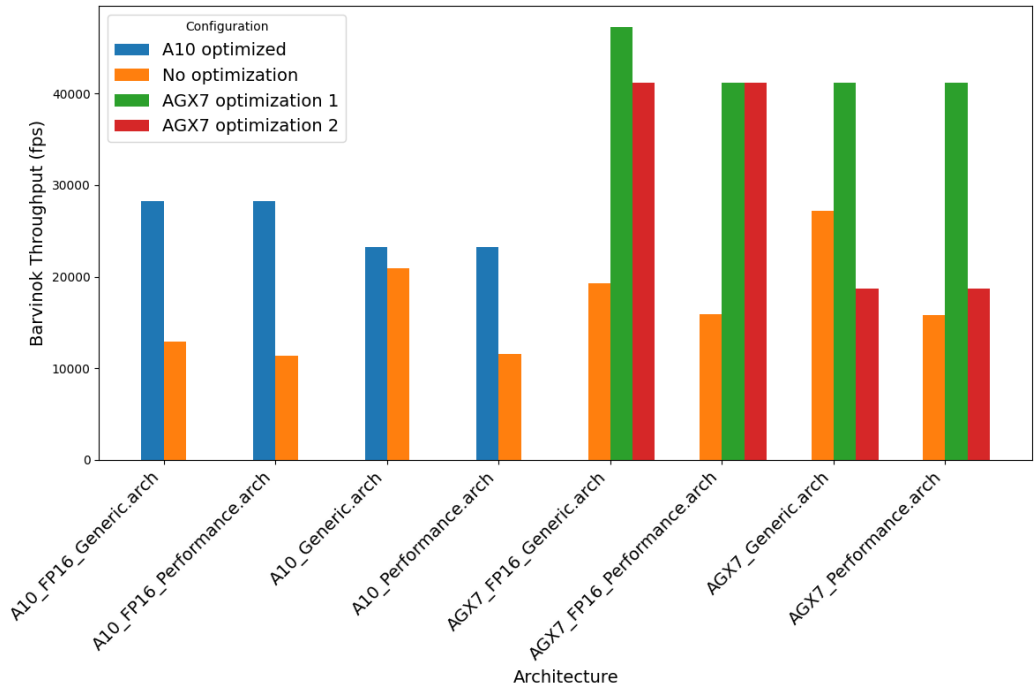
3.2 Architecture dependence

Figures 4(a) and 4(b) illustrate the throughput performance of different CNN models as estimated by `dla_compiler` for the Arria 10 and Agilex 7 FPGAs across various architectures.

A striking difference is observed between the FP16 Generic architecture and the regular Generic architecture. The FP16 Generic architecture consistently outperforms its regular counterpart, a trend that is less pronounced when comparing the Performance architectures. Moreover, the optimizations applied to the FPGA architectures yield substantial performance improvements, with an average 2x increase in throughput, as clearly seen in both figures.



(a) The mean throughput performance across all CNN models as estimated by `dla_compiler` for the Arria 10 and Agilex 7 FPGAs. Error bars indicate the standard deviation in throughput from the mean across all CNN models for a given architecture.



(b) The throughput performance for the Barvinok model as estimated by `dla_compiler` for the Arria 10 and Agilex 7 FPGAs.

Figure 4: Throughput performance comparisons for the Arria 10 and Agilex 7 FPGAs: (a) Mean throughput across all CNN models, and (b) Throughput for the Barvinok model. "FPGA Unoptimized" refers to the respective FPGA architecture, either A10 or AGX7, based on the prefix.

In general, the throughput varies significantly across the different CNN models, as evidenced by the large error bars in Figure 4(a). The error bars represent the standard deviation in throughput from the mean across all CNN models. This variability indicates that the performance of FPGAs is highly model-dependent, with some models achieving much higher throughput than others. However, it is notable that the unoptimized Performance architectures exhibit more consistent throughput across models, with relatively smaller error bars.

Figure 5 presents the throughput performance for the Barvinok model, including optimizations as measured by the compiler with `dla_compiler` and directly run on the Arria 10 FPGA using `dla_benchmark`. The optimized throughputs, as seen for both the Generic and Performance architectures, are identical. This consistency is expected as the FPGA is designed to optimize the architectures uniquely.

However, an interesting observation is that the optimized throughput for the Generic architecture is lower than the throughput for the unoptimized architecture. This could indicate that the optimization process is specifically tailored to the Barvinok model, which might inadvertently reduce the effectiveness of the generic optimization when applied more broadly. Further investigation, including compiling and testing the corresponding bitstreams, is suggested to fully understand the cause of this discrepancy.

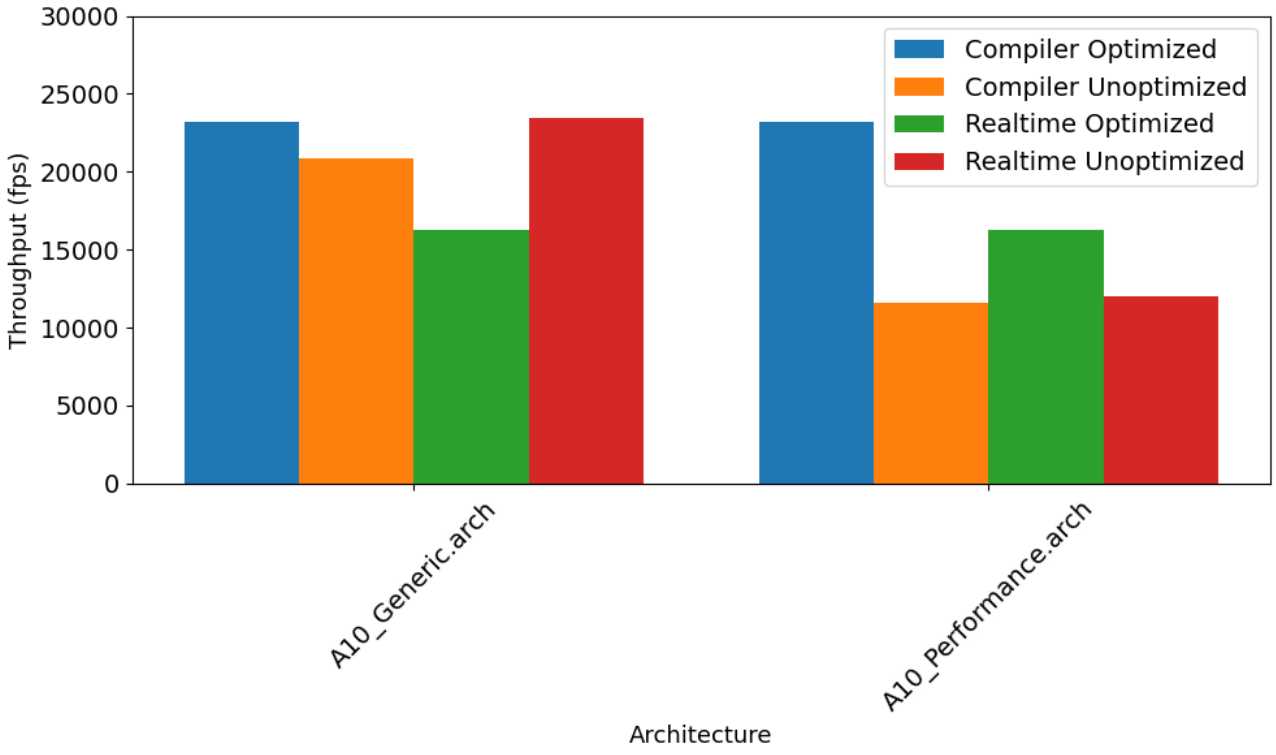


Figure 5: Throughput performance including optimizations as measured by the compiler with `dla_compiler` and directly run on the Arria 10 FPGA with `dla_benchmark`. The estimates are given for the Barvinok model.

3.3 Comparison with ResNet-50

Figures 6 and 7 illustrate the throughput performance for the ResNet-50 model with both the compiler-predicted and real-time performance across various architectures. Several key observations can be made from these graphs.

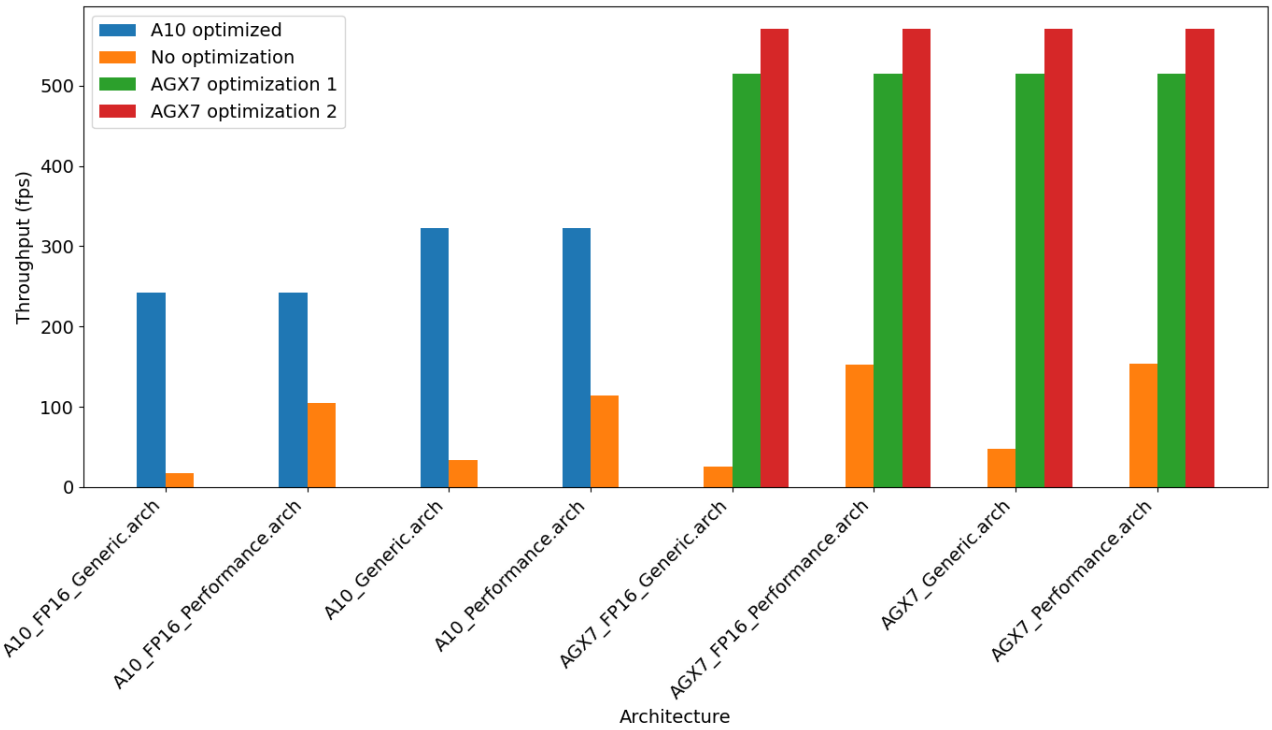


Figure 6: Compiler throughput performance as a function of architecture for the ResNet-50 model of 25.6M parameters for the Arria 10 and Agilex 7 FPGAs.

Firstly, it is evident that the compiler's prediction for the expected performance boost is significantly higher for the ResNet-50 model compared to the previous CNN models (10x performance boost).

Secondly, an intriguing observation is the performance difference between the two Optimization 1 and Optimization 2. Unlike the previous CNN models where Optimization 1 typically outperformed Optimization 2, the ResNet-50 model shows a reversal of this trend and aligns better with expectations: with more resources, the net throughput should increase. Moreover, it is worth noting that the optimizations predicted by the compiler are consistent with those observed in real-time performance, as shown in the comparison between the compiler and real-time throughput in subfigure (b).

Thirdly, the observed consistency in performance improvements for the ResNet-50 model across both predicted and real-time data indicates that the previous inconsistencies noted with the Barvinok model are likely model-specific. The fact that the Barvinok model did not exhibit the same degree of performance consistency suggests that further testing and analysis are necessary to fully understand the underlying factors contributing to the observed discrepancies in performance gains.

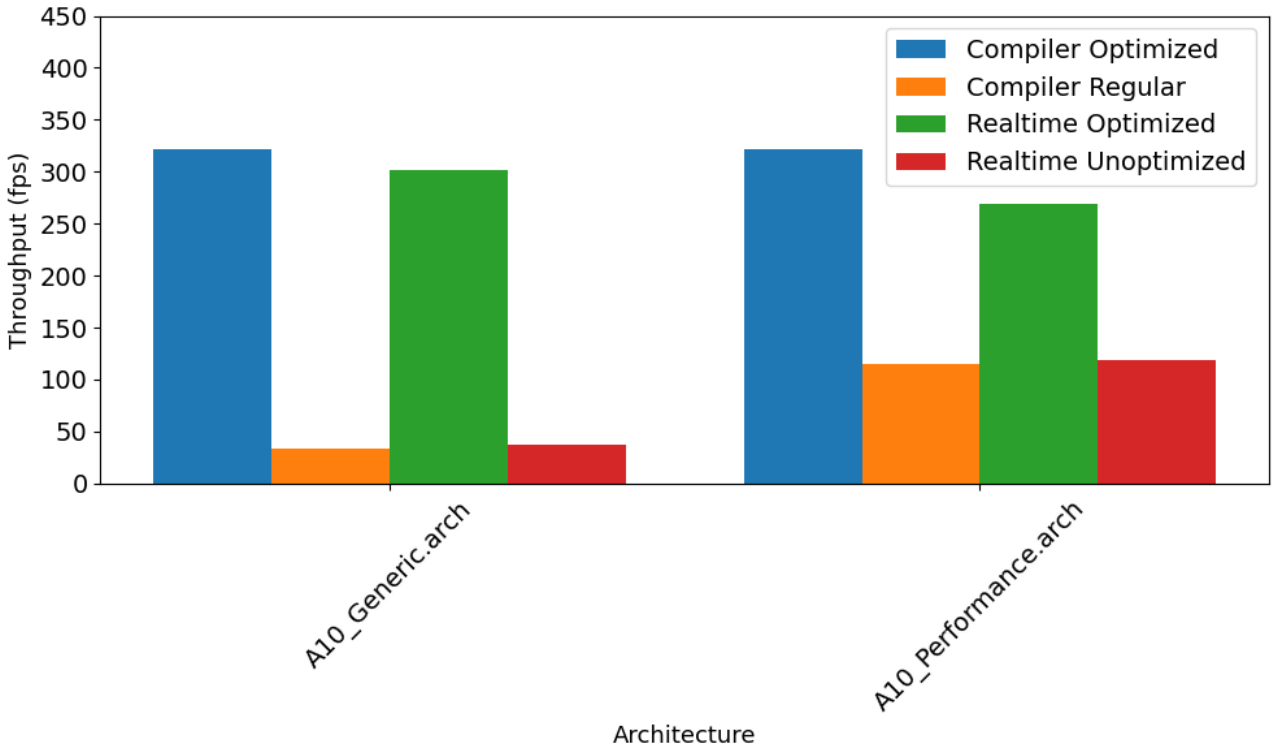


Figure 7: A comparison between the compiler and realtime throughput performance as a function of architecture for the ResNet-50 model of 25.6M parameters including optimizations.

Finally, the optimized throughput for ResNet-50, as shown in Figure 7, varies slightly between the two architectures. This difference is due to a technical adjustment during the Quartus bitstream compilation, where the estimated resource usage exceeded the available limit. To address this, the maximum resource bound was reduced to 1450 DSPs instead of the full 1518 DSPs, leading to a marginal reduction in performance.

Table 1: Throughput and Mean Latency of ResNet-50 on GPU2

Device	**Throughput (samples/sec)**	**Mean Latency (ms)**
GPU	788.84	48.56

Table 1 shows the throughput and latency performance of ResNet-50. As can be seen, the GPU has more than double the throughput of an FPGA.

3.4 Power usage and area estimates

The graph in Figure 8 clearly demonstrates the significant power efficiency of FPGA devices compared to GPUs. The power usage of the GPU is notably higher across all inference types, with ResNet-50 and Barvinok models consuming around 309 and 89 watts respectively, whereas the Arria 10 FPGA configurations consistently consume less than half of that (or less than 10 times for ResNet-50), typically ranging between 24.5 to 36 watts depending on the architecture.

Tables 2 and 3 presented below highlight the FPGA resource usage for both the Barvinok models and the ResNet-50 model across various architectures and optimization strategies. A key observation

is the disparity in performance and resource utilization between Optimization 1 and Optimization 2 across different models. For the CNN trigger models (Barvinok 2 and Barvinok 3), Optimization 1 outperforms Optimization 2 in throughput, despite Optimization 2 occupying a significantly larger area in terms of ALMs, M20Ks, and DSPs. Conversely, for the ResNet-50 model, Optimization 2 outperforms Optimization 1, as expected, given that Optimization 2 occupies more resources across all FPGA architectures. This correlation between resource allocation and performance aligns better with the typical behavior observed in FPGA optimizations.

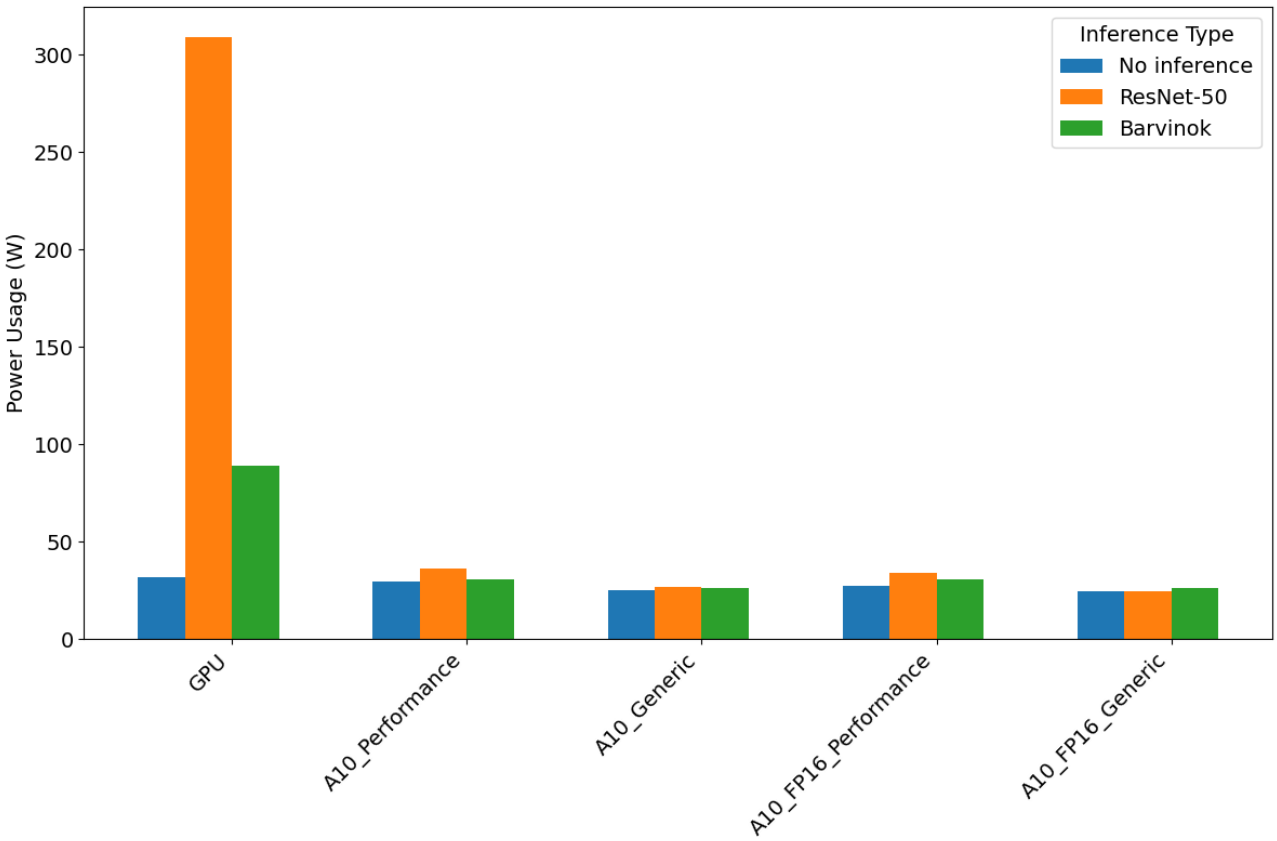


Figure 8: The power usage as a function of the device used: GPU and the Arria 10 FPGA. The maximum resources for

Additionally, it is important to note that for some A10 architectures used with the ResNet-50 model, the number of occupied DSPs exceeded the maximum specified limits. To address this, the maximum DSP allocation was reduced by the difference between the occupied area and the specified maximum, resulting in a final DSP count of 1518 during the execution of the `dla_compiler`.

Table 2: FPGA Resource Usage for Barvinok 2 and Barvinok 3 Models using the Performance architecture. The maximum specified resources are as follows. For Arria 10: 427,200 ALMs, 2,713 M20Ks, and 1,518 DSPs. For Agilex 7: Optimization 1 has 487,200 ALMs, 7,110 M20Ks, and 4,510 DSPs, while Optimization 2 offers 1,305,600 ALMs, 18,960 M20Ks, and 12,300 DSPs.

Optimization	Model	Occupied ALMs	Occupied M20Ks	Occupied DSPs
A10 Unoptimized	Barvinok 2	47259	953	650
A10 Optimized	Barvinok 2	47876	1545	362
A10 Unoptimized	Barvinok 3	47259	953	650
A10 Optimized	Barvinok 3	47876	1545	362
AGX7 Unoptimized	Barvinok 2	62451	1237	650
AGX7 Optimization 1	Barvinok 2	122489	1471	1226
AGX7 Optimization 2	Barvinok 2	175652	2486	2314
AGX7 Unoptimized	Barvinok 3	62451	1237	650
AGX7 Optimization 1	Barvinok 3	122489	1471	1226
AGX7 Optimization 2	Barvinok 3	175652	2486	2314

4 Conclusion

This study conducted a comprehensive evaluation of deep neural network performance on various hardware platforms, including FPGAs, GPUs, and CPUs, within the context of the Cherenkov Telescope Array's real-time triggering system. The primary metrics of interest—throughput and latency—were assessed across different neural network models and hardware architectures.

Key findings from this research include the superior performance of the Agilex 7 FPGA, particularly when optimized, which outperformed both the Arria 10 FPGA and the GPU with double the throughput. However, the study also revealed that resource allocation does not always directly correlate with performance improvements, as demonstrated by the inconsistency between Optimization 1 and Optimization 2 across different models. Specifically, for the CNN trigger models, Optimization 1 (the AFG014 board) outperformed Optimization 2 (the AGM039 board) despite using fewer resources. Conversely, for the ResNet-50 model, Optimization 2, which utilized more resources, delivered better performance, aligning with expected outcomes.

The study also revealed general consistency between compiler-estimated and real-time throughput values, thereby validating the compiler's predictions. However, discrepancies were noted between optimized compiler estimates and actual real-time throughput, particularly for the Barvinok model. The ResNet-50 model showed more consistent performance improvements across both predicted and real-time data. This suggests that the inconsistencies observed with the Barvinok model may be model-specific and require further investigation.

FPGAs showed remarkable power efficiency. The power consumption of FPGA configurations was consistently less than half that of the GPU across all inference types, making FPGAs a highly attractive option for the energy-constrained CTA project.

Future research directions include:

Table 3: FPGA area estimate and resource usage for ResNet-50 on various architectures of Arria 10. The maximum specified resources for Arria 10 are: 427,200 ALMs, 2,713 M20Ks, and 1,518 DSPs. For Agilex 7: Optimization 1 has 487,200 ALMs, 7,110 M20Ks, and 4,510 DSPs, while Optimization 2 offers 1,305,600 ALMs, 18,960 M20Ks, and 12,300 DSPs.

Optimization	Architecture	Occupied ALMs	Occupied M20Ks	Occupied DSPs
Unoptimized	A10_FP16_Generic.arch	24457	507	186
Optimized	A10_FP16_Generic.arch	165515	2670	1518
Unoptimized	A10_FP16_Performance.arch	67165	1489	1162
Optimized	A10_FP16_Performance.arch	167981	2670	1518
Unoptimized	A10_Generic.arch	26517	627	202
Optimized	A10_Generic.arch	134696	2649	1518
Unoptimized	A10_Performance.arch	47259	953	650
Optimized	A10_Performance.arch	136094	2649	1518
Unoptimized	AGX7_FP16_Generic.arch	30633	508	186
Optimized1	AGX7_FP16_Generic.arch	268240	6829	4362
Optimized2	AGX7_FP16_Generic.arch	268898	11949	4362
Unoptimized	AGX7_FP16_Performance.arch	90186	1490	1162
Optimized1	AGX7_FP16_Performance.arch	268319	6827	4362
Optimized2	AGX7_FP16_Performance.arch	268977	11947	4362
Unoptimized	AGX7_Generic.arch	34987	769	202
Optimized1	AGX7_Generic.arch	158282	5324	2314
Optimized2	AGX7_Generic.arch	158589	9164	2314
Unoptimized	AGX7_Performance.arch	62451	1237	650
Optimized1	AGX7_Performance.arch	158292	5324	2314
Optimized2	AGX7_Performance.arch	158599	9164	2314

1. Investigating optimization processes to understand the performance disparity between ResNet-50 (10x gain) and trigger models (1.5x gain).

2. Exploring multi-core FPGA configurations to address scaling inefficiencies in AGX7 deployments.

3. Expanding the analysis to encompass a wider range of neural network models and hardware configurations.

In conclusion, while FPGAs, especially the Agilex 7, show great potential for CTA's real-time triggering, ongoing optimization of hardware-software integration remains crucial for maximizing their performance in resource-constrained, high-performance environments.

Bibliography

[1] W. Hofmann and R. Zanin, "The cherenkov telescope array," 2023.

[2] U. Schwanke, M. Shayduk, K.-H. Sulanke, S. Vorobiov, and R. Wischnewski, "A versatile digital camera trigger for telescopes in the cherenkov telescope array," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 782, p. 92–103, May 2015.

[3] "Target: A digitizing and trigger asic for the cherenkov telescope array," in *AIP Conference Proceedings*, Author(s), 2017.

[4] J. Zhang, R. Zhou, S. Zhang, J. Zhang, H. Xiong, G. Hu, Y. Li, Z. Liu, and C. Yang, "Data acquisition and trigger system for imaging atmospheric cherenkov telescopes of the lhaaso," *Journal of Instrumentation*, vol. 15, p. T02004, feb 2020.

[5] D. Nieto, A. Brill, Q. Feng, T. B. Humensky, B. Kim, T. Miener, R. Mukherjee, and J. Sevilla, "Ctlearn: Deep learning for gamma-ray astronomy," 2019.

[6] P. Grespan, M. Jacquemont, R. López-Coto, T. Miener, D. Nieto-Castaño, and T. Vuillaume, "Deep-learning-driven event reconstruction applied to simulated data from a single large-sized telescope of cta," 2021.

[7] A. Nechi, L. Groth, S. Mulhem, F. Merchant, R. Buchty, and M. Berekovic, "Fpga-based deep learning inference accelerators: Where are we standing?," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, 09 2023.

[8] Z. Szadkowski and A. Szadkowska, "Fpga implementation of the trigger based on a fuzzy logic for a detection of cosmic rays in water cherenkov detectors," pp. 431–435, 12 2019.

[9] P. Reddy and K. Ramanaiah, "Field-programmable gate array implementation of efficient deep neural network architecture," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 14, p. 3863, 08 2024.