



IOWA STATE UNIVERSITY

The Department of Electrical and Computer Engineering

Real-time Eye Tracking Through Optimized Semantic Segmentation

Conner Ohnesorge, Tyler Schaefer, Aidan Perry, Joey Metzen

IOWA STATE UNIVERSITY

NDA Statement

Please note that our team is operating under a Non-Disclosure Agreement (NDA) with our client.

As a result, some technical details may appear limited in this presentation, but we've taken care in presenting the project clearly while respecting the privacy terms set by our client.

Team Members

Joey Metzen - CprE



Conner Ohnesorge - EE



Aidan Perry - CprE

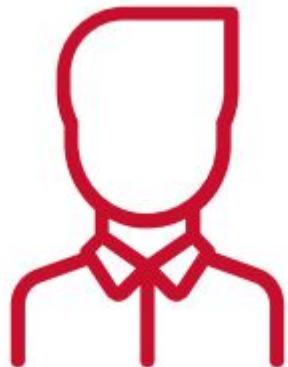


Tyler Schaefer - SE



SDDec25-01

Client: JR Spidell



Advisor: Dr. Vaswani



Problem Statement

People with disabilities face risks from undetected medical issues. Traditional methods lack real-time monitoring.



Using eye movement tracking with semantic segmentation can detect warning signs and automatically reposition users to prevent incidents, improving safety needs.

6

Client

JR volunteered to help individuals with cerebral palsy.

Our goal is to create an assistive wheelchair technology.
We hope our work will improve mobility and enhance
quality of life for the people who use it.



Resources



Hardware:
Kria Board Kv260

Software:

- Xilinx
- Vitis-AI
- Pytorch
- ONNX
- ONNX-Runtime
- Petalinux

Main Components:

- DDR4 RAM [4GB]
- DPU
- Quad-SPI Flash (Boot ROM) [64MB]
- Cortex-A53 L1 Data Cache
- Cortex-A53 L2 Cache [1MB]

Our Objective

Develop an end-to-end video processing pipeline on the Kria Board that performs real-time pupil segmentation.

Pipeline components:

- Capture live video from a desktop camera
- Detect face and locate eyes using ROI algorithm
- Crop and preprocess eye images
- Run semantic segmentation model on FPGA DPU
- Display segmented output in real-time

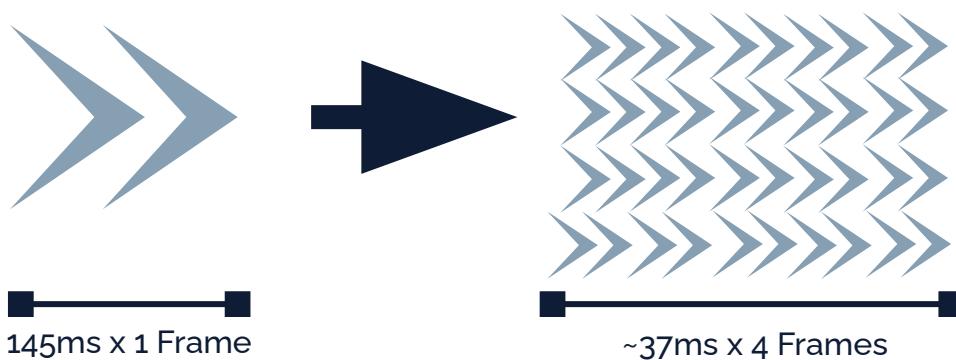
Challenge: Achieve real-time performance on embedded hardware with limited memory and processing power while maintaining segmentation accuracy

Client Project Restrictions

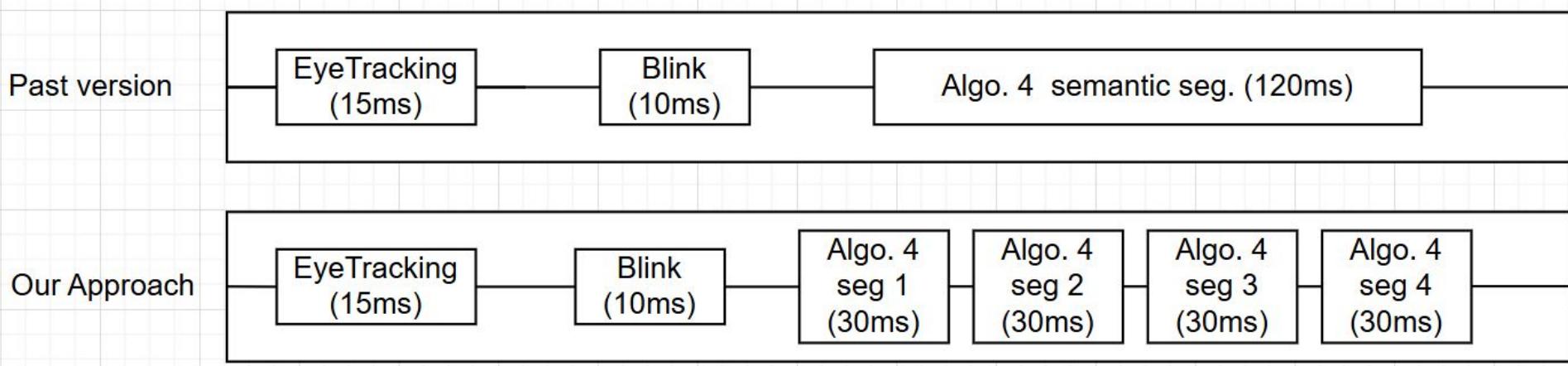
- Do not touch the FPGA fabric of the given board.
- Do not change the UNet Algorithm, other than splitting it.
- NDA-covered content is not to be shared.
- Benchmark and compare the performance of the split vs. unsplit UNet.
- Maintain accuracy through the splitting of the model.

10

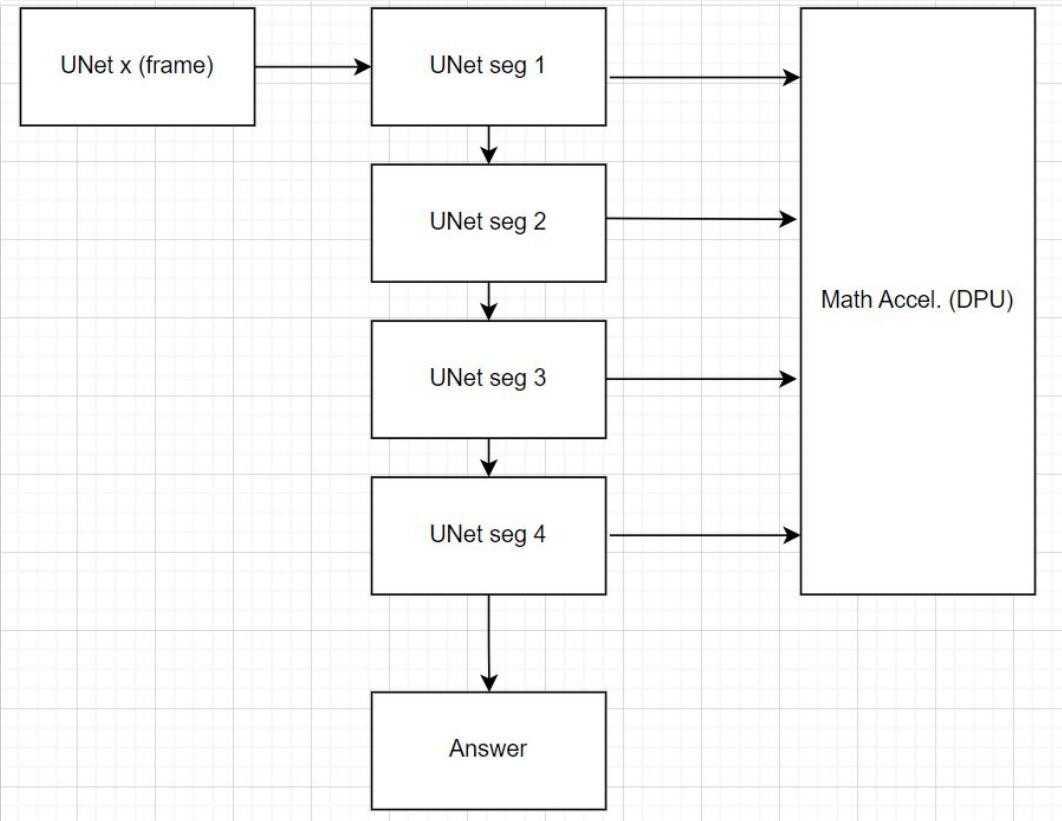
Our Approach



Speaker icon Increase throughput by splitting the U-net algorithm over 4 cores and across the DPU.



Proposed Parallelization

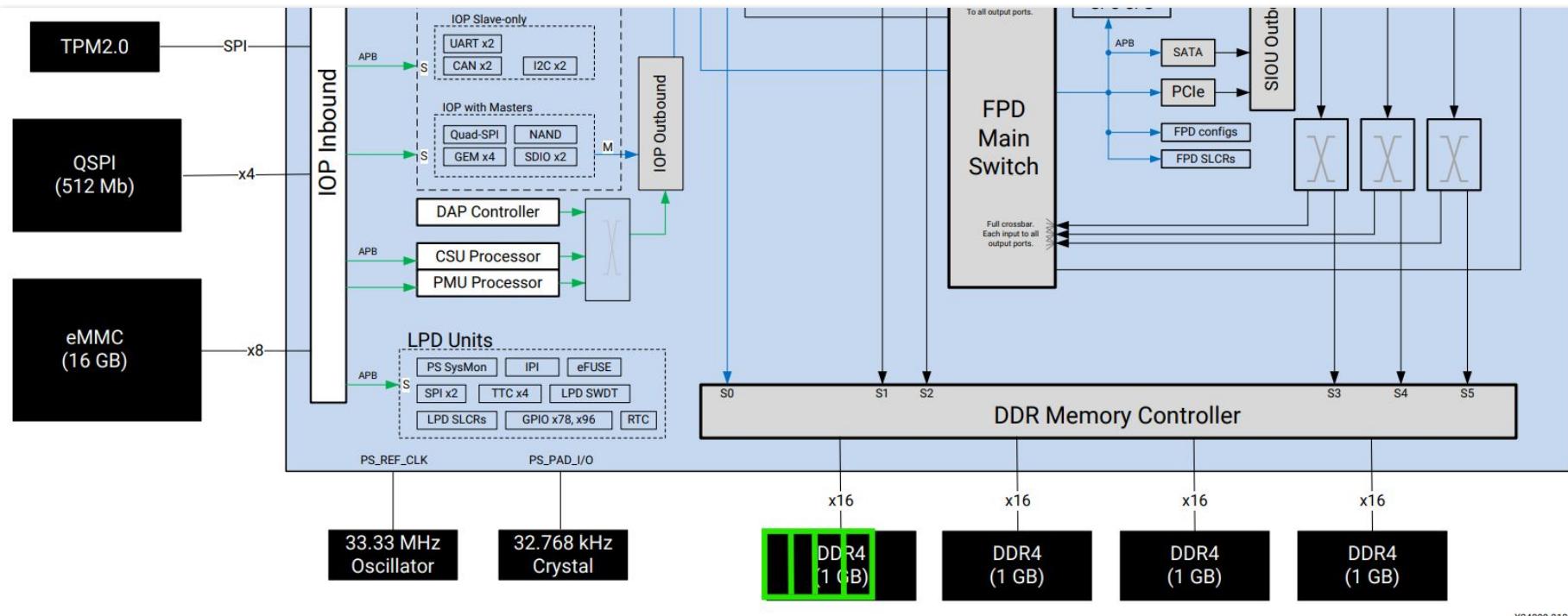


Tracked Metrics

- Throughput
- Accuracy
- Resource Utilization



Emphasis on Resource Utilization:

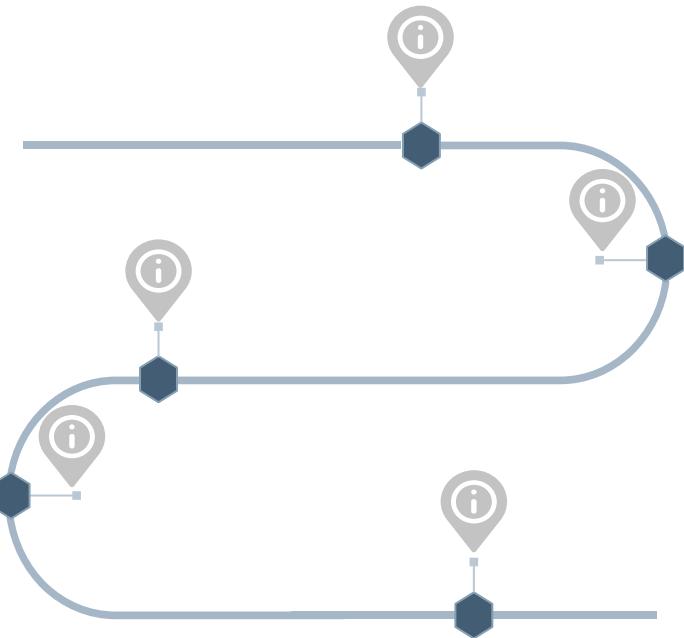


X24999-012122

- Understand how much memory needed to be allocated for each split segment of the UNET
- Weights of each split segment
- Contain the memory within each segment of one of the accessible options for memory

Milestones Reached

- Mathematical division of the Algorithm
- Loading of Split Algorithm weights onto DPU
- Attempt split implementation of the Semantic Segmentation algorithm across the 4 developed threads.
- Benchmarked previous teams algorithms against our approach.
- Compared split implementation with the provided team's results.



Challenges Faced

Threads/Processes running on multiple cores:

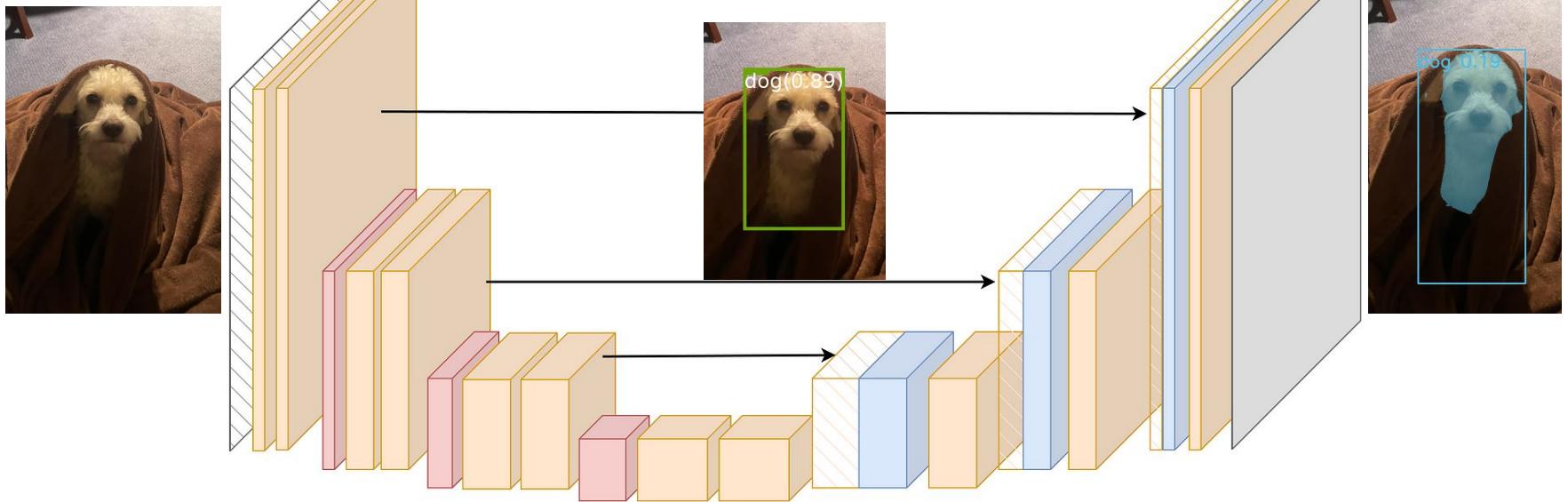
- All cores share the same memory bus.
- For example: at 16 cores, bandwidth will start to saturate causing cache misses and stalls.
- Individual cores still needs to send/receive data to neighboring cores introducing several blocks with processes waiting for the slowest neighbor.
- Work per Process example: $2048 \text{ pixels} / 8 \text{ cores} = 256 \text{ pixels per process}$ to work on. Too little work to be able to reduce communication costs.

Takeaway: The analization of splitting the model and running it on multiple cores as opposed to 1 would have too much overhead cost to be effective on a single DPU.

Challenges Faced: Observation

- No speedup observed in either implementation
 - Single core process: ~6x slower than initial implementation
 - Multiple cores: ~8x slower than initial implementation
- Again the key takeaway in these observations is being limited to the single DPU doesn't benefit the overhead communication costs.
 - Single model: ~85% added latency
 - Multi-model: ~135% added latency

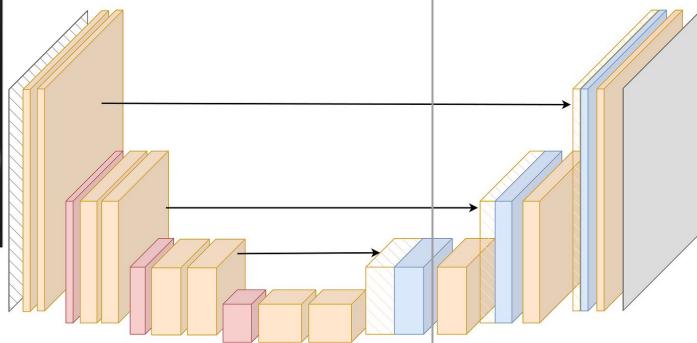
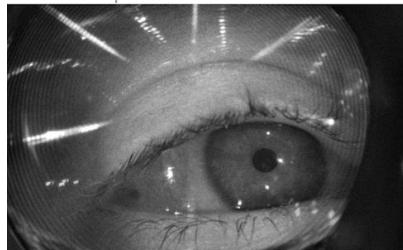
Our Model



U-net Semantic Segmentation cont.

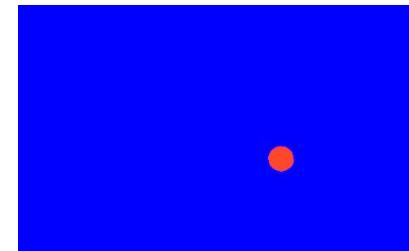
Contracting Encoder

- Downsampling (i.e. 2x2 Max Pool) compensated by the doubles # channels
 - Transmit to across to decoder
- Includes spatial info

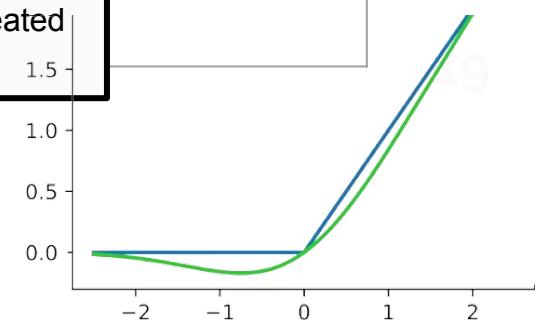
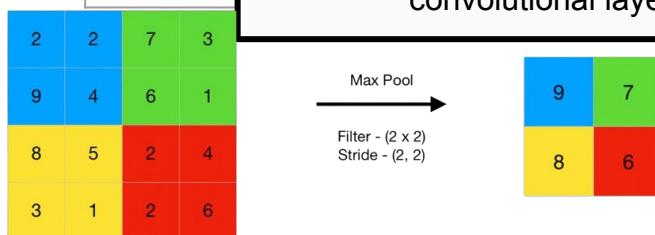


Expanding Decoder

- Receive from encoder
 - Upsampling (i.e. 2x2 Convolution) decreases the amount of channels
- Includes semantic Info

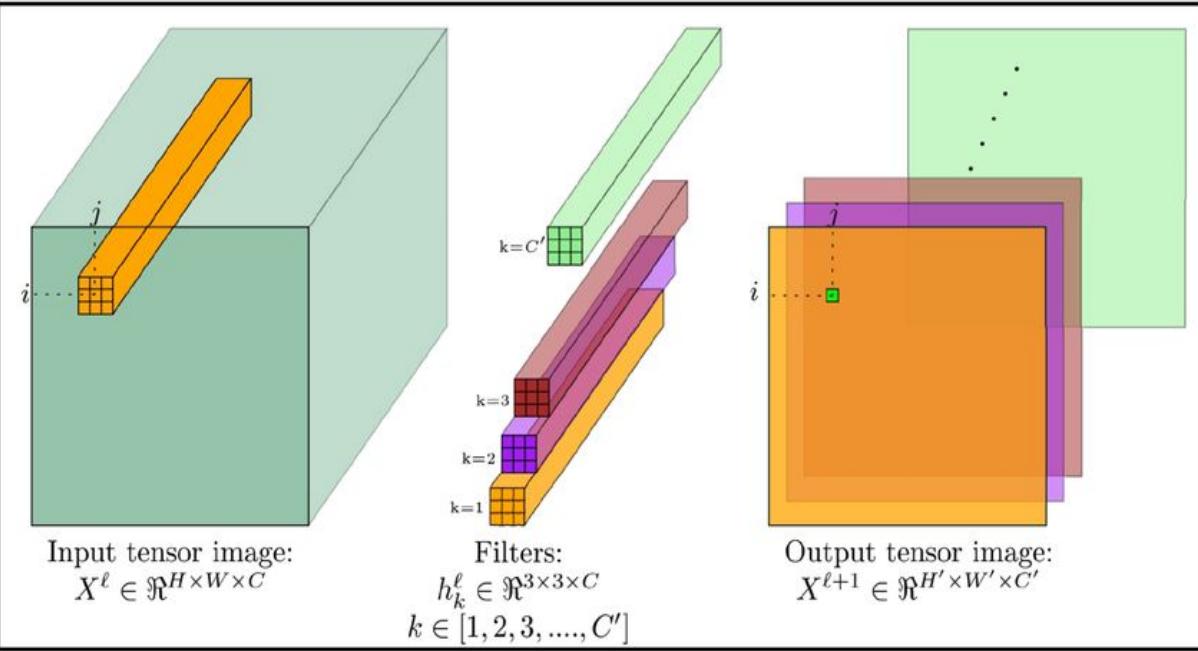


Each “forward step” applies a relu function to the output of a repeated convolutional layer application over input channels.



Segmenting the Model

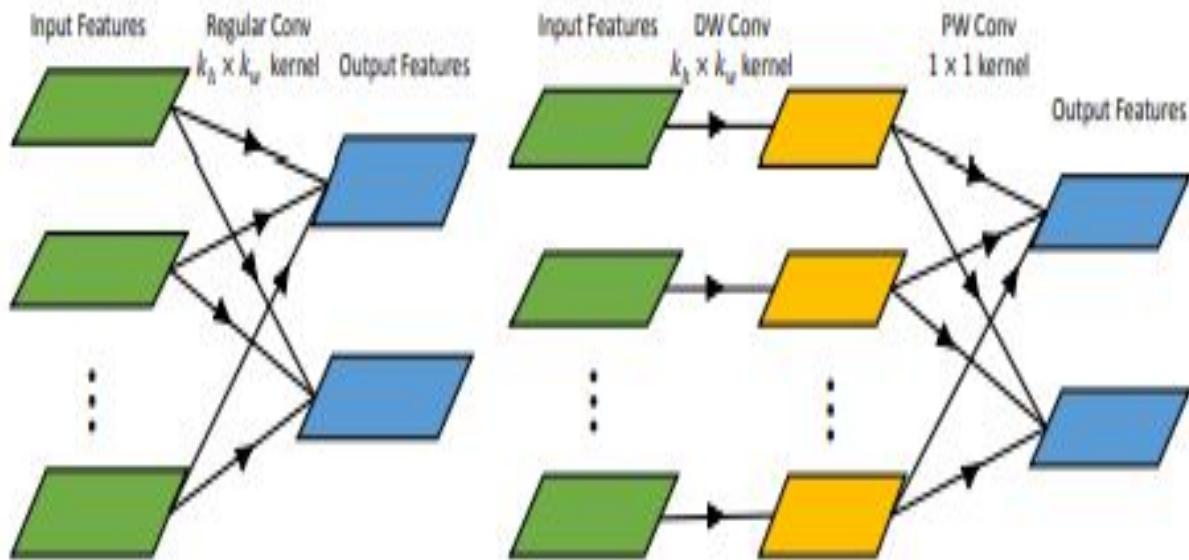
Computational Complexity Analysis



Traditional Convolutions

$$O(H \times W \times C_{in} \times K \times C_{out})$$

Computational Complexity Analysis



Depthwise Separable Convolutions

Depthwise
 $O(H \times W \times C_{in} \times K)$

Pointwise
 $O(H \times W \times C_{in} \times C_{out})$

Results

Algo/Training Baseline/Given

- Time: ~50 hrs on CPU to get to 99.7% validation score
- Observability: Terminal Logging
- Unmaintained & Untracked Scripts
- Dataset specialization.
- Improper Cuda Setup
- No available Central Repository of properly tracked experiments and results
- Untrained Onnx Model

First/Quick Solution

- Allow Proper GPU Training
 - Remove all CPU to GPU transfers during epochs
 - Take advantage of increased batch size capability
 - Remove in training loop logging and improper sends of tensors to devices
- Training time reduced **15x** to ~4hrs (T4) from >50hrs
- Track and version control with good comments and commit messages
- Observability through MLFlow
- Fix nonstandard model weights in git practice by using GitLab Model Registry

15x

Split Model Results

- Single Model Performance:
 - Mean Total Latency: [REDACTED]
 - Throughput (FPS): [REDACTED]
 - Memory Utilization: ~42 MB
- Split Model (4 Segments) Performance:
 - Mean Total Latency: [REDACTED]
 - Throughput (FPS): [REDACTED]
 - Memory Utilization: ~134 MB
- Performance Comparison:
 - The Single Model exhibits ~8x greater speed (lower latency) than the Split Model.
 - The DPU time ratio (Single/Split) is ~0.1x.

Why so bad?

Short Answer: `vai_c_xir` (Vitis AI Compiler for the DPU)

Long Answer:

The following requantization operations occur between segments, supporting our claims of Xilinx compiler issues:

- Segment 1 → Segment 2: scale 0.25 → 16.0, ~2M elements
- Segment 1 → Segment 3 (input 0): scale 1.0 → 32.0, ~8M elements
- Segment 2 → Segment 3 (input 1): scale 1.0 → 32.0, ~8M elements
- Segment 3 → Segment 4: scale 0.125 → 16.0, ~8M elements

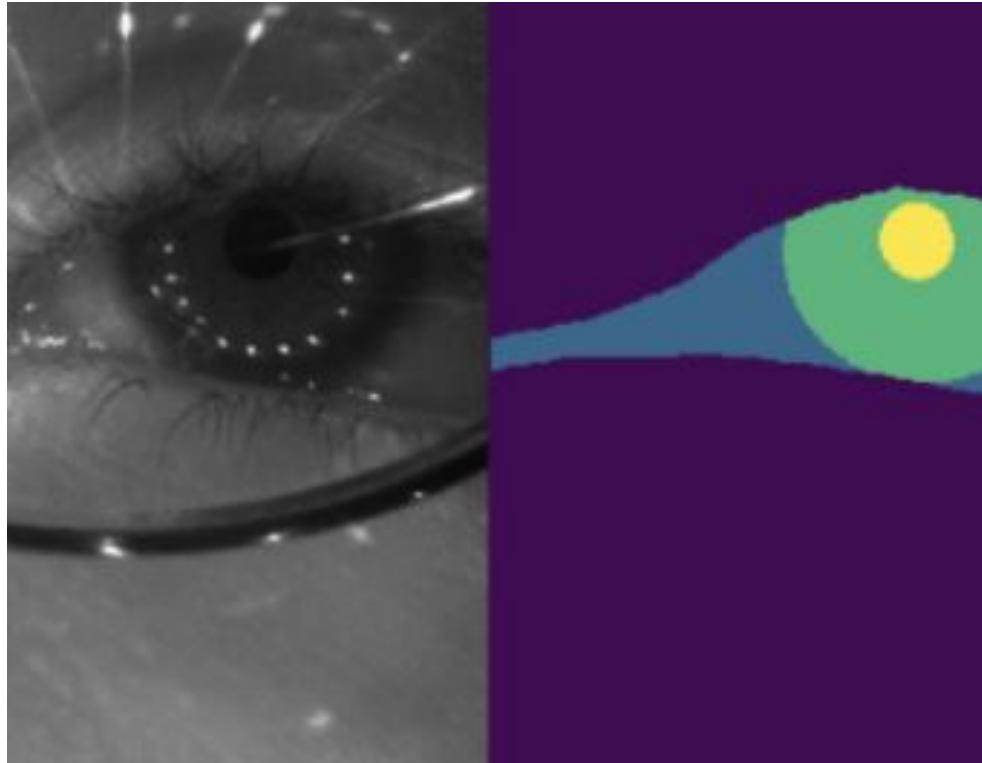
These scale mismatches require costly CPU-side requantization/scaling between DPU calls, contributing significantly to the latency gap.

```
quantizer = torch_quantizer(  
    "calib",  
    model,  
    (  
        torch.randn(1, 32, 640, 400),  
        torch.randn(1, 32, 640, 400),  
    ),  
    device=torch.device("cpu"),  
    target="DPUCZDX8G_ISA1_B4096",  
)
```

```
vai_c_xir \  
  -x $XMODEL_FILE \  
  --arch $ARCH_JSON \  
  --output_dir $OUTPUT_DIR \  
  --net_name $MODEL_NAME
```

Our Observations

- Pupil segmentation is a task demanding highly precise pixel localization.
- Pupil is exclusively located within the boundaries of the eye.
- Above are not leveraged by the UNet model.
- Client chosen dataset is only in VR domain
 - Data and Dataset for actually detecting medical issues from eye tracking and/or pupil shape in respect to time does not exist.



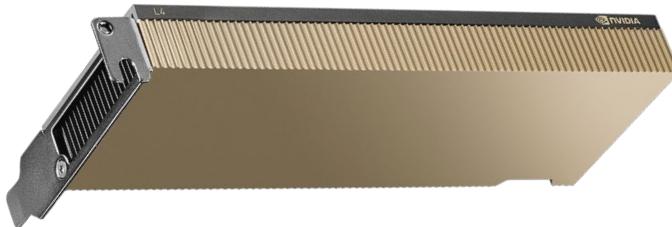
Alternatives to the Unet

Tiny EfficientVit

12K Params

Trained for 15 epochs
to beat baseline.

Time: 3hrs

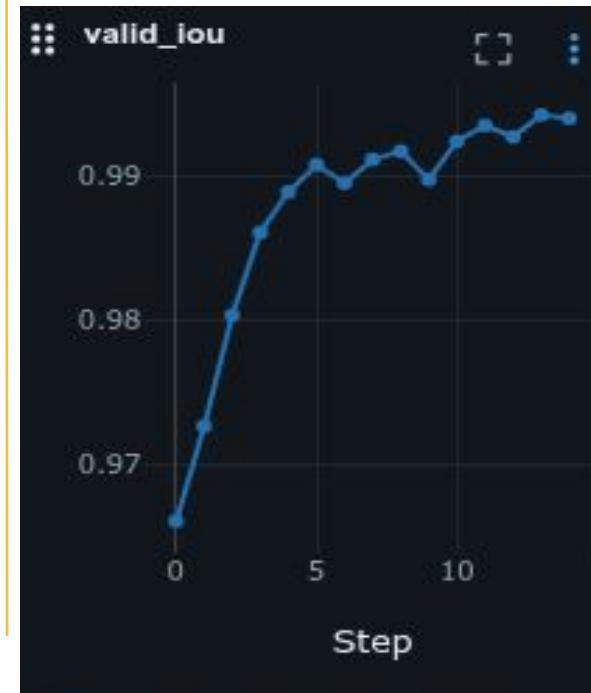


EfficientVit (NSA)

5K & 20k Param Versions

Trained for 13 epochs
to beat baseline.

Time: 4hrs



$$\text{Attn}(\mathbf{q}_t, \mathbf{k}_{:t}, \mathbf{v}_{:t}) = \sum_{i=1}^t \frac{\alpha_{t,i} \mathbf{v}_i}{\sum_{j=1}^t \alpha_{t,j}}, \quad \alpha_{t,i} = e^{\frac{\mathbf{q}_t^\top \mathbf{k}_i}{\sqrt{d_k}}}$$

Native Sparse Attention

Sliding Window Attention

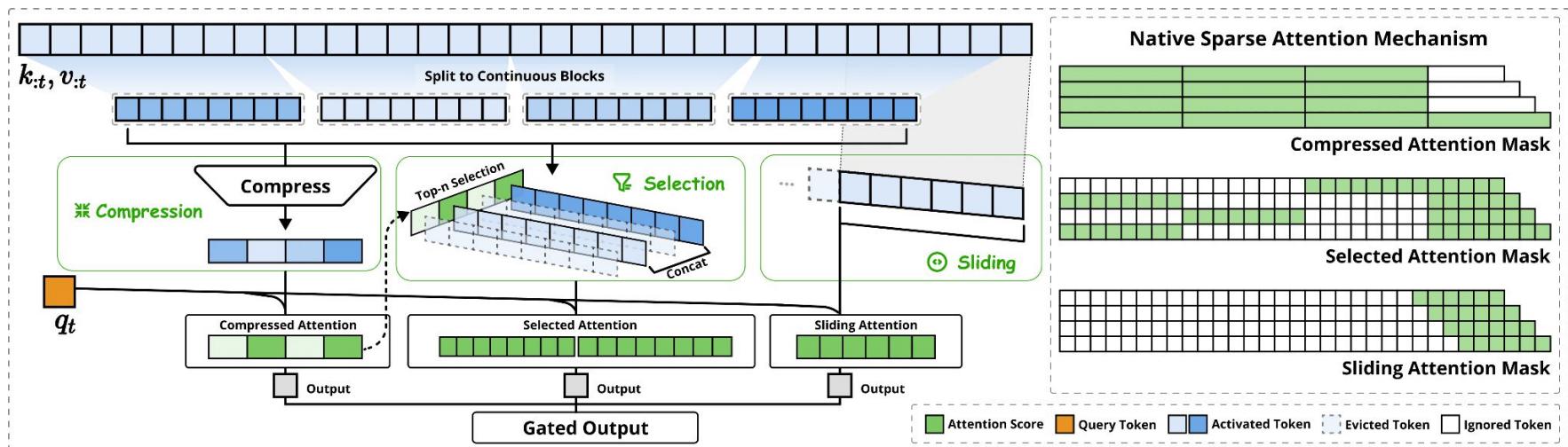
- Restricts attention to a local window (e.g., 7x pixels) around each token, capturing fine-grained local details (such as the edges of the pupil).

Compressed Attention ([TokenCompression](#))

- **Downsamples** the Key and Value maps (using strided blocks) to capture **global** context with fewer tokens.

Selected Attention ([TokenSelection](#))

- Calculates **importance** scores for token blocks and only attends to the "Top-K" most important blocks. This allows the model to focus on the pupil while ignoring the background.



Our Impact

- "Pipelining" or model splitting is a possible solution but significantly impacts performance with the current platform and tool chain.
- Created 2 new models based on Deepseeks Native Sparse Attention: Pico and Nano.
 - **Pico** can run on CPU at over 10x the speed of the original model.
 - **Nano**, with hardware acceleration (testing with GPU and MPS), gives a 20x increase in performance while having approximately 1/10 the number of parameters.
- Convinced Client to move/update hardware platform.

Questions?