5 Testing

Testing Strategy Overview

Testing is key to our Semantic Segmentation project. We need to make sure our system meets our goals of fast processing (<16.6ms between frames) while keeping good accuracy(99.8%)

Testing Philosophy

We test early and often. This helps us catch problems quickly and fix them before they get worse. For our project, this means:

- Testing each part of the divided U-Net algorithm as we create it
- Checking memory use before building the full system
- Testing how we share DPU resources as we develop

Testing Challenges

Our project has some tough testing challenges:

- Testing on FPGA hardware is different from normal software testing
- Making sure our parallel threads work together correctly
- Balancing speed and accuracy
- Checking that memory is used correctly

Testing Schedule

- Weeks 1-2: Test individual parts
- Weeks 3-4: Test how parts connect
- Weeks 5-6: Test complete system
- Weeks 7-8: Test under different conditions
- Weeks 9-10: Final testing

5.1 Unit Testing

Algorithm Testing

- **Division Testing**: Check each segment of the divided U-Net algorithm to make sure it works like the original.
    - Tool: Python scripts and ONNX runtime
- **Memory Testing**: Make sure each algorithm part stays within its 1GB memory limit.
    - Tool: Vitis Memory Debugger
- **Preprocessing Testing**: Test image cleanup with different eye images.
    - Tool: OpenCV tests

Thread Testing

- **Thread Timing**: Check that threads work together in the right order.
    - Tool: GDB debugger and timing scripts
- **DPU Access**: Test our system for sharing the DPU between tasks.
    - Tool: Vitis DPU Profiler

Success Goals

- Each algorithm part must match the original within 1% error
- Threads must work in the correct order
- Memory use must stay within limits

5.2 Interface Testing

Key Interfaces

1. **Between Algorithm Parts**: Test how data moves between divided parts of the U-Net algorithm
   - Method: Send test images through connected parts
   - Tool: Data checking scripts
2. **Between Software and Hardware**: Test how our code uses the DPU
   - Method: Track DPU use during running
   - Tool: Vitis AI Profiler, GDB debugger
3. **Memory Management**: Test how threads use their assigned memory
   - Method: Try to access memory from wrong threads
   - Tool: Vitis memory tools, Vivado Hardware Manager
4. **Thread Coordination**: Test how threads talk to each other
   - Method: Change timing to test stability
   - Tool: Timing analysis tools

Test Cases

1. **Data Transfer Test**:
   - What we do: Send eye images through connected algorithm parts
   - What should happen: Data stays correct between parts

    - How we check: Compare with original algorithm
2. **DPU Access Test**:
    - What we do: Make multiple threads request DPU at once
    - What should happen: Requests handled by priority
    - How we check: No lockups, predictable order
3. **Memory Test**:
    - What we do: Try to use memory from other threads
    - What should happen: Access blocked
    - How we check: System reports error correctly

5.3 Integration Testing

Critical Paths

1. **Full Pipeline Test**: Test complete flow from input to output
    - Method: Process test images through whole system
    - Tool: Automated testing with result checking
2. **Parallel Thread Test**: Test running multiple threads at once
    - Method: Feed multiple frames and check parallel processing
    - Tool: Thread monitor, Vitis AI Profiler
3. **DPU Load Test**: Test DPU sharing under heavy use
    - Method: Create high demand for DPU
    - Tool: Vitis AI Profiler

System Benchmarking

- **Performance Tracking**: Use Vitis AI tools to measure:
  - Time per algorithm part
  - Overall speed (frames per second)
  - Memory use
  - DPU efficiency
- **Pipeline Speed**: Measure total time from input to output
  - Tool: Timing probes

Success Goals

- System must process >60 frames per second
- Accuracy must stay between 98%

System Testing

Test Plan

1. **Continuous Running Test**:
   - What we do: Feed many eye images continuously
   - Tool: Image generator with logging
   - Goal: Keep 16.6 ms between frames for over 30 minutes
2. **Lighting Test**:
   - What we do: Test with images in different lighting

    - Tool: Dataset with lighting variations
    - Goal: Keep accuracy above 98% in all conditions
3. **Stress Test**:
    - What we do: Push memory and processing limits
    - Tool: Stress testing scripts
    - Goal: System stays running without failing
4. **Long-Term Test**:
    - What we do: Run system for 24+ hours
    - Tool: Automated testing with monitoring
    - Goal: No crashes or slowdowns over time

Test Measurements

- **Speed**: Frames per second (goal: >60)
- **Accuracy**: Correct pupil tracking (goal: >98%)
- **Time**: Input to output delay (goal: 60 frames per second)
- **Memory**: How much memory is used over time
- **Stability**: How long the system runs without problems

5.4 Regression Testing

Automated Testing

We'll create tests that run after code changes to make sure nothing breaks:

1. **Performance Check**: Compare speed to previous tests
    - Tool: Test runner with history database
2. **Accuracy Check**: Make sure algorithm changes don't hurt accuracy
    - Tool: Test dataset with known answers
3. **Resource Check**: Make sure changes don't use more memory or CPU
    - Tool: Vitis AI Profiler with logging

Monitoring

- **Performance Tracking**: Use Vitis AI Profiler to watch:
    - Running time
    - DPU use
    - Memory use
    - Thread timing
- **Memory Leak Check**: Test for memory problems that could cause crashes
    - Tool: Memory tracking in our test system

Test Schedule

- Run basic tests after each code change
- Run full tests every night
- Keep history of all test results
- Set up alerts if tests start failing

5.5 Acceptance Testing

Function Tests

1. **Speed Test**:
    - Test: Process multiple frames
    - Goal: 60 frames per second
2. **Accuracy Test**:
    - Test: Compare with manually marked images
    - Goal: 98-99.8% accuracy
3. **Multi-frame Test**:
    - Test: Process several frames at once
    - Goal: Handle 4 frames at the same time

Other Requirements

1. **Memory Test**:
    - Test: Track memory during long runs
    - Goal: Each thread stays within 1GB
2. **Stability Test**:
    - Test: Run for 24+ hours
    - Goal: No crashes or slowdowns
3. **Thread Test**:
    - Test: Watch threads work together
    - Goal: No lockups or timing problems

Client Involvement

We'll invite our client to see our testing and get feedback:

1. Show the system tracking eyes in real-time
2. Show speed improvements
3. Compare original and improved versions
4. Let client test with their own data

5.6 Results

Current Progress

So far, we've tested the algorithm division and started testing interfaces:

1. **Algorithm Division**:
    - Beginning to split U-Net into four parts that work like the original
    - Current accuracy: 98.8% (within our target)
    - Processing load is balanced between parts

Next Steps

Based on testing, our next steps are:

1. **Speed Improvement**:
    - Refine algorithm parts to work faster
    - Improve DPU scheduling
    - Goal: Get to 60 frames per second
2. **Thread Coordination**:
    - Make data sharing between threads more efficient
    - Reduce coordination overhead

    - Goal: Reduce delays between threads

3. **Full System Testing**:
    - Build complete integrated system
    - Test end-to-end performance
    - Goal: Keep accuracy while improving speed

Our tests show our approach works, with accuracy in the target range. We now need to focus on making the system faster to meet our 60 frames per second goal.