**Multithreaded Scene Detection in Video Streams**

**1. Introduction**

**Motivation & Problem Statement**  
Scene changes detection is crucial in video analysis for editing, indexing, and surveillance. Traditional sequential algorithms process frames one after another, which becomes a bottleneck with high-resolution videos and longer durations. Our motivation was to build a fast, scalable Java solution capable of handling larger files in less time by leveraging multithreading and producer-consumer patterns.

**2. Design**

**Algorithms:**

* **Scene Detection**: Calculate frame differences using pixel-wise comparison (sum of absolute differences, SAD). If the difference exceeds a threshold, mark as scene change.
* **Extraction**: Read video, extract frames at set FPS, push to processing pipeline.

**Data Structures:**

* ArrayList<BufferedImage> to store frames in sequential and non-queued multi-threaded versions.
* BlockingQueue<BufferedImage> for producer-consumer design to support real-time extraction and processing.

**Synchronization:**

* No explicit locking in sequential/naive multi-threaded versions.
* In producer-consumer: Used thread-safe queue (e.g., LinkedBlockingQueue) for communication between threads, avoiding race conditions.

**Justification:**

* Chose SAD for its simplicity and speed.
* BlockingQueue enables scalable, real-time processing without memory overflow.

**3. Implementation Notes**

**Tricky Parts & Obstacles:**

* **Memory Management**: With large videos, storing all frames in RAM led to OutOfMemoryError.
* **Thread Coordination**: Ensuring consumer threads wait when the queue is empty and exit gracefully.
* **Edge Cases**: Handling videos shorter than thread count, or frames near video end.

**How We Solved:**

* Limited frame storage flushed processed frames.
* Used BlockingQueue’s built-in synchronization.
* Added end-of-stream markers in the queue to notify consumers.

**4. Testing Methodology**

**Correctness:**

* Manually verified detected scene change frames by reviewing timestamps in the original video.
* Compared detected scenes with expected scene transitions.

**Performance:**

* Ran each version (sequential, multi-threaded without queue, producer-consumer) on multiple videos.
* Recorded extraction time, detection time, total time, and memory usage.
* Repeated runs for average timings.

**5. Results**

**A. Single Threaded**

| **Trial** | **Extraction (ms)** | **Detection (ms)** | **Total (ms)** | **Frames** |
| --- | --- | --- | --- | --- |
| 1 | 26126 | 82894 | 109320 | 889 |
| 2 | 26926 | 89778 | 117027 | 889 |
| ... | ... | ... | ... | ... |

**Scene Changes Detected (example):**

→ Frame: 15 / -> Time: 0.4994355

→ Frame: 90 / -> Time: 2.996613

→ Frame: 171 / -> Time: 5.6935647

→ Frame: 249 / -> Time: 8.290629299999999

→ Frame: 291 / -> Time: 9.689048699999999

→ Frame: 354 / -> Time: 11.7866778

→ Frame: 400 / -> Time: 13.31828

→ Frame: 467 / -> Time: 15.549091899999999

→ Frame: 526 / -> Time: 17.5135382

→ Frame: 604 / -> Time: 20.1106028

→ Frame: 605 / -> Time: 20.1438985

**B. Multi-threaded (No Queueing)**

* Could not process more than **40 sec @ 30fps** **4K** videos due to RAM limitation (all frames must be held in memory).
* Using the 40sec\_30fps video \_
  + 5 threads gave the best result with an average of 93sec processing time and a minimum of 86sec
  + The trend of thread was decreasing from 1 to 5 and then an increase when adding more threads, this is because the video is short compared to using more than 5 threads

**C. Producer-Consumer (Queueing)**

* **Major improvement**: Could handle up to **4K videos, 1min 30s @ 30fps**.
* Both extraction and detection happened in parallel, memory was never overloaded.
* Using the 40sec\_30fps video:
  + 4 threads gave the best result with an average of 51sec
  + 6 threads gave the minimum time at 42sec
* Using the 1min\_30fps video:
  + 6 threads gave the best result with an average of 57sec and a minimum of 49sec

**6. Comparison with Sequential**

**Wins:**

* Multi-threaded and producer-consumer versions reduced detection time significantly.
* Producer-consumer was the only design able to process long, high-res videos (RAM-efficient).

**Trade-offs:**

* Overhead of thread management for small videos.
* Without queueing, increasing threads doesn’t always give linear speed-up due to memory and I/O bottlenecks.

**7. Conclusion & Future Work**

**Conclusion:**

* Multithreaded scene detection in Java is feasible and provides huge speed-ups.
* Producer-consumer model scales best and avoids RAM bottlenecks.
* With queueing, can process videos that sequential and naive threaded approaches cannot handle.

**Future Work:**

* GPU-based acceleration.
* Adaptive thread pool sizing.
* Smarter frame diff algorithms for improved detection accuracy.