CUDA Multi-Camera AprilTag Driver

Architecture Design Document

---

1. Purpose & Scope

This document defines the architecture for a **GPU-accelerated multi-camera AprilTag detection driver** running on a **Jetson Orin NX**.

The system is designed to:

- Work **now** with a single Arducam **USB3 IMX477P** camera.
- Scale **later** to **four AR0234 USB3 global-shutter** cameras (corner pods).
- Perform **all heavy per-pixel operations on the GPU**, including:
    - Raw → Gray conversion
    - Optional undistortion / decimation
    - AprilTag detection
    - Optional PnP for 3D pose
- Provide:
    - A clean **C++ API** for low-level integration
    - A **Python wrapper** for rapid experimentation

Primary use case: **FRC-style robotics vision** with multi-camera AprilTag-based localization, but the architecture is generic enough for broader robotics applications.

---

2. Requirements

2.1 Functional Requirements

- Capture images from **USB3 UVC cameras**:
    - IMX477P (current)
    - AR0234 (future 4-pod system)
- Acquire frames in **native/raw format** (e.g., Bayer, YUYV, BGR8).
- Upload raw frames to the **GPU** using pinned memory and async CUDA transfers.
- Convert raw frames to **Gray8** on the GPU.
- Optionally perform **undistortion** and **decimation** on the GPU.
- Run a **CUDA-based AprilTag detector** over the Gray8 image.
- Compute **3D tag poses** (position + orientation) in:
    - Camera frame, and optionally
    - Robot frame using camera extrinsics.
- Support **single-camera** and **multi-camera** configurations (1 → 4 cameras).
- Provide **C++** and **Python** APIs for detection results.

2.2 Non-Functional Requirements

- **Latency target**: < 10 ms from frame capture to 3D pose per camera under typical FRC conditions.
- Frame rate: aim for **≥ 60 FPS @ 720p** per camera with reasonable scaling to four cameras.
- Architecture must be:
    - **Modular** (clear separation of camera, GPU, detection, orchestration, API).
    - **Extensible** (easy to add new sensors or adjust detector parameters).

- Robust against:
    - Camera disconnects
    - GPU errors
    - Partial failures (one camera failing shouldn't kill the whole system).
- Logging and diagnostics sufficient to debug camera + GPU issues.

---

3. High-Level Architecture

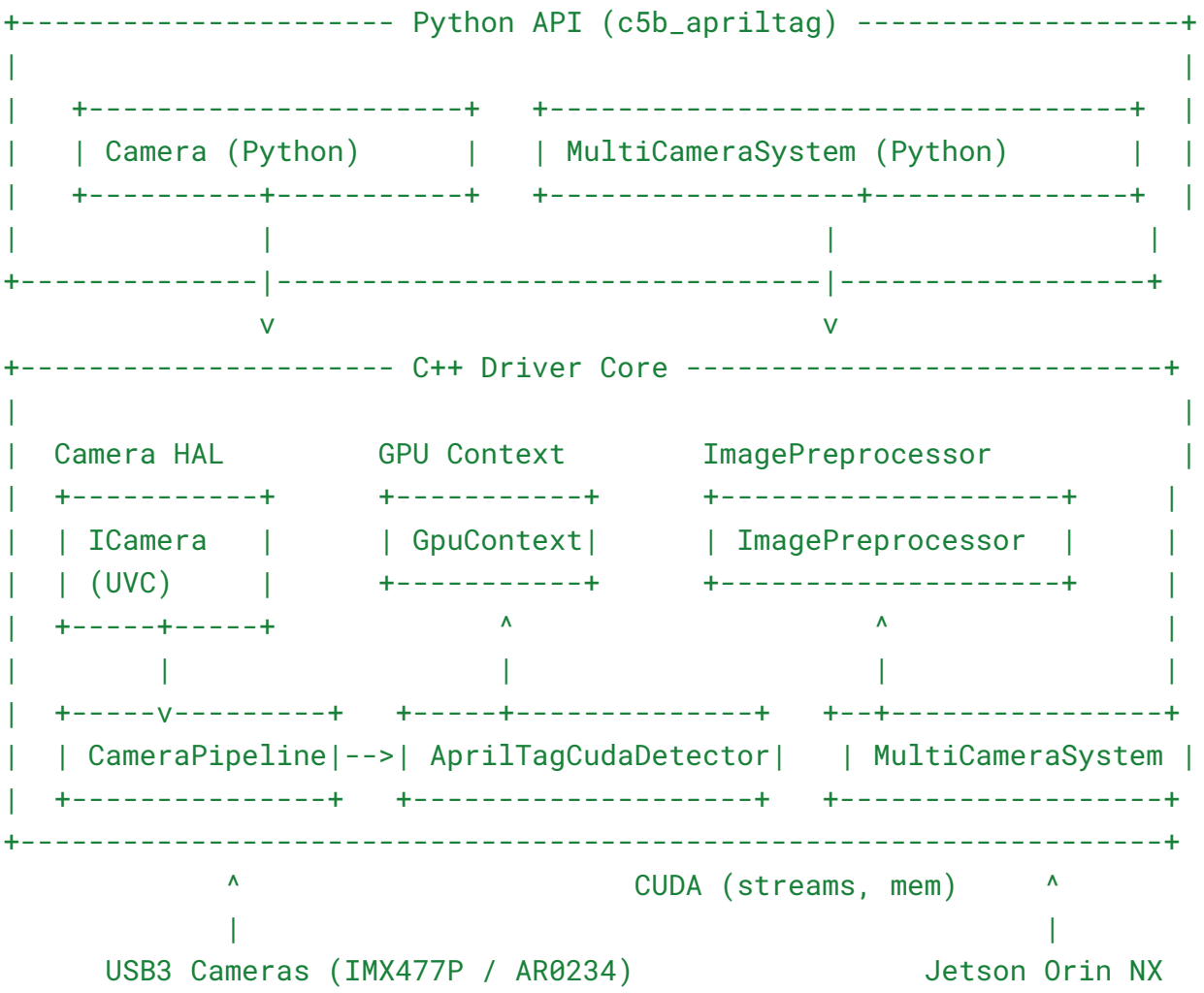
The system is organized into seven logical components:

1. **Camera HAL (`ICameraDevice`)**
    - Abstracts USB3 cameras as raw frame sources.
2. **GPU Context (`GpuContext`)**
    - Owns the CUDA device, context, and stream allocation.
3. **Image Preprocessor (`ImagePreprocessor`)**
    - Uploads raw frames to GPU.
    - Performs raw → Gray conversion, undistortion, and decimation on GPU.
4. **AprilTag CUDA Detector (`AprilTagCudaDetector`)**
    - Runs core AprilTag pipeline on GPU using Gray8 images.
    - Optionally runs GPU PnP.
5. **Camera Pipeline (`CameraPipeline`)**
    - Binds one camera + one CUDA stream + one preprocessor + one detector.
6. **Multi-Camera Manager (`MultiCameraAprilTagSystem`)**

- Manages 1..N `CameraPipeline`s and orchestrates processing.

7. **Python Wrapper (`c5b_apriltag` module)**
    - High-level API for testing and integration in Python.

Architecture Diagram (ASCII)



---

4. Component Designs

4.1 Camera HAL

4.1.1 Enums and Data Types

```
enum class PixelFormat {
    GRAY8,
    BAYER_RG8,
    YUYV422,
    BGR8
};

struct CameraStaticInfo {
    std::string name;       // "IMX477_Test", "FL", "FR", etc.
    std::string devicePath; // "/dev/video0"
    std::string sensor;     // "IMX477P", "AR0234"
    bool        globalShutter;
};

struct CameraRuntimeConfig {
    int width;
    int height;
    int fps;
    PixelFormat format;      // native camera format
    bool  autoExposure;
    double exposureUs;
    double gain;
};

struct CameraCalibration {
    int   width;
    int   height;
    float fx, fy, cx, cy;
    std::array<float, 5> distCoeffs; // k1, k2, p1, p2, k3
    float tagSizeMeters;
};

struct RawFrameFormat {
    PixelFormat format;
    int width;
    int height;
    int bytesPerPixel;
};
```

4.1.2 Interface: `ICameraDevice`

```
class ICameraDevice {
public:
    virtual ~ICameraDevice() = default;

    virtual const CameraStaticInfo&    staticInfo()    const = 0;
    virtual const CameraRuntimeConfig& runtimeConfig() const = 0;
    virtual const CameraCalibration&   calibration()   const = 0;

    // Capture one raw frame into pinned host memory.
    // Returns false on failure.
    virtual bool grabRaw(uint8_t* dst,
                         int dstStrideBytes,
                         RawFrameFormat& outFormat) = 0;
};
```

4.1.3 Implementation: `UvcCameraDevice`

- Uses **V4L2** or **OpenCV**.
- Configured via `CameraRuntimeConfig` (resolution, fps, native format).
- For IMX477P:
    - Prefer `BAYER_RG8` or `BGR8`.
- For AR0234:
    - Prefer `GRAY8` or `BAYER_RG8`.

No gray conversion is performed here; this layer is “camera → raw bytes”.

4.2 GPU Context

4.2.1 Data Types

```
struct GpuImage {
    int width;
```

```
    int height;
    int stride;       // bytes per row
    CUdeviceptr d_ptr;// Gray8 or other
};

struct RawGpuImage {
    int width;
    int height;
    int stride;        // bytes per row
    CUdeviceptr d_ptr; // raw image on device
    RawFrameFormat format;
};
```

4.2.2 Interface: `GpuContext`

```
class GpuContext {
public:
    explicit GpuContext(int deviceIndex = 0);
    ~GpuContext();

    cudaStream_t createStream(const std::string& name);
    void         destroyStream(cudaStream_t stream);

    GpuImage    allocImage(int width, int height);
    RawGpuImage allocRawImage(int width, int height, PixelFormat fmt);

    void freeImage(GpuImage& img);
    void freeRawImage(RawGpuImage& img);

    // Async copy: host raw frame → RawGpuImage
    void uploadRawAsync(const uint8_t* hostPtr,
                        int hostStrideBytes,
                        RawGpuImage& dst,
                        cudaStream_t stream);
};
```

`GpuContext` is responsible for CUDA initialization, stream creation, and device buffer management.

4.3 Image Preprocessor

The **ImagePreprocessor** is the key new component that ensures **raw → gray, undistort, decimate** all happen on the GPU.

4.3.1 Configuration Types

```
struct PreprocessParams {
    bool  undistort;        // apply lens undistortion
    bool  decimate;         // downsample
    float decimateFactor;   // e.g., 1.0, 2.0
};
```

4.3.2 Interface: `ImagePreprocessor`

```
class ImagePreprocessor {
public:
    ImagePreprocessor(GpuContext& ctx,
                      const CameraCalibration& calib);

    // Upload host raw frame to RawGpuImage on device.
    void uploadRawAsync(const uint8_t* hostPtr,
                        int hostStrideBytes,
                        RawGpuImage& rawGpu,
                        cudaStream_t stream);

    // Convert rawGpu -> grayGpu (Gray8 working image) on device.
    void rawToGrayAsync(const RawGpuImage& rawGpu,
                        GpuImage& grayGpu,
                        const PreprocessParams& params,
                        cudaStream_t stream);

private:
    GpuContext& ctx_;
    CameraCalibration calib_;
    // Precomputed undistortion maps, if used, or parameters for
kernels.
};
```

4.3.3 GPU Kernel Responsibilities (Preprocessing)

The `ImagePreprocessor` uses several GPU kernels; per-camera, per-frame work includes:

1. **Raw Upload (host → device)**
    - Handled by `GpuContext::uploadRawAsync` using `cudaMemcpy2DAsync`.
    - Input: pinned host buffer, stride
    - Output: `RawGpuImage` in device memory.
2. **Raw → Gray kernel** (format-dependent)
    - For IMX477P (e.g., `BAYER_RG8`):
        - Kernel reads Bayer pattern and computes a grayscale value per pixel. (This can be simple demosaic-lite or more advanced later.)
    - For AR0234:
        - If `GRAY8`: may simply copy or do minor adjustments.
        - If Bayer: same Bayer→Gray kernel as IMX477P.
    - For YUYV/BGR8 (if needed):
        - Compute Gray via a luminance formula: Y = 0.299R + 0.587G + 0.114B.
3. **Undistortion / Rectification kernel** (optional)
    - Input: Gray image in distorted coordinates.
    - Output: rectified Gray8 image, using:
        - `fx, fy, cx, cy`
        - `distCoeffs` (radial/tangential)
    - Can be merged into the Raw→Gray kernel (single pass) or separate.
4. **Decimation / Resize kernel** (optional)

- Input: Gray8 image (possibly undistorted).
- Output: Gray8 image at reduced resolution (e.g., 1280×720 → 640×360).
- Simple averaging or nearest-neighbor, depending on quality/performance tradeoff.

Design note:
We can start with a **single combined kernel** that:

- Reads raw format
- Converts to Gray8
- Applies undistortion
- Applies decimation

All in one pass to maximize cache usage and minimize memory traffic.

4.4 AprilTag CUDA Detector

4.4.1 Data Types

```
enum class TagFamily {
    TAG_36H11,
    TAG_16H5,
    // extend as needed
};

struct Tag3dPose {
    float x, y, z;       // meters
    float qw, qx, qy, qz;// quaternion
};

struct TagDetection3D {
    int   id;
    float decisionMargin;
    float hamming;
```

```
    float centerX, centerY;
    float corners[4][2]; // pixels
    Tag3dPose pose;      // in camera or robot frame
};

struct AprilTagDetectorConfig {
    TagFamily family;
    float     quadDecimate;
    float     blurSigma;
    int       threads;       // CPU threads for final decode, if
needed
    bool      useGpuPnP;     // true = PnP on GPU
};
```

4.4.2 Interface: `AprilTagCudaDetector`

```
class AprilTagCudaDetector {
public:
    AprilTagCudaDetector(const AprilTagDetectorConfig& cfg,
                         const CameraCalibration& calib,
                         GpuContext& gpuCtx);

    ~AprilTagCudaDetector();

    std::vector<TagDetection3D> detect(
        const GpuImage& grayImg,
        cudaStream_t stream,
        const float cameraToRobot[16] = nullptr
    );

private:
    AprilTagDetectorConfig cfg_;
    CameraCalibration      calib_;
    GpuContext&            gpuCtx_;
    // Internal GPU buffers, 971/766 structures, etc.
};
```

4.4.3 GPU Kernel Responsibilities (Detection Stage)

High-level stages:

1. **Gradient & Edge Detection**
    - Kernels compute image gradients from Gray8 input.
    - Thresholding/edge maps for quad candidates.
2. **Quad Extraction**
    - Label/cluster edges into quadrilaterals.
    - Estimate corners and geometry.
3. **Tag Decode (GPU + CPU hybrid)**
    - GPU can help with sampling/tag bit extraction.
    - CPU may finalize decoding for small candidate sets.
4. **PnP (Pose Estimation)**
    - If `useGpuPnP`:
        - A batched kernel takes tag corner coordinates and tag size → computes pose in camera frame.
    - Otherwise:
        - CPU calls `solvePnP` per detection.
5. **Frame Transform**
    - If `cameraToRobot` provided:
        - Multiply pose by 4×4 transform to yield tag pose in robot frame.

4.5 Camera Pipeline (Per-Camera Wrapper)

4.5.1 Configuration Types

```
struct CameraPoseConfig {
```

```
    float cameraToRobot[16]; // 4x4 homogeneous transform, row-major
};

struct CameraPipelineConfig {
    CameraStaticInfo       staticInfo;
    CameraRuntimeConfig    runtimeConfig;
    CameraCalibration      calibration;
    CameraPoseConfig       pose;
    AprilTagDetectorConfig detectorConfig;
    PreprocessParams       preprocessParams;
};
```

4.5.2 Interface: `CameraPipeline`

```
class CameraPipeline {
public:
    CameraPipeline(const CameraPipelineConfig& cfg,
                   GpuContext& gpuCtx);

    ~CameraPipeline();

    std::vector<TagDetection3D> processOnce();

    const std::string& name() const { return cfg_.staticInfo.name; }

private:
    CameraPipelineConfig cfg_;
    GpuContext&          gpuCtx_;

    std::unique_ptr<ICameraDevice>    camera_;
    std::unique_ptr<ImagePreprocessor> preproc_;
    std::unique_ptr<AprilTagCudaDetector> detector_;

    cudaStream_t stream_;
    RawGpuImage  rawGpu_;
    GpuImage     grayGpu_;

    uint8_t* hostPinned_      = nullptr;
    int      hostStrideBytes_ = 0;
```

```
    void captureRaw();
};
```

4.5.3 Flow of `processOnce()`

1. **captureRaw()**
    - `camera_->grabRaw(hostPinned_, hostStrideBytes_, rawFormat);`
2. **Upload raw → device**
    - `preproc_->uploadRawAsync(hostPinned_, hostStrideBytes_, rawGpu_, stream_);`
3. **Raw → Gray + undistort/decimate**
    - `preproc_->rawToGrayAsync(rawGpu_, grayGpu_, cfg_.preprocessParams, stream_);`
4. **Detection**
    - `detector_->detect(grayGpu_, stream_, cfg_.pose.cameraToRobot);`
5. **Return `std::vector<TagDetection3D>`**

4.6 Multi-Camera Manager

4.6.1 Interface: `MultiCameraAprilTagSystem`

```
class MultiCameraAprilTagSystem {
public:
    MultiCameraAprilTagSystem(
        const std::vector<CameraPipelineConfig>& cameraConfigs,
        GpuContext& gpuCtx);

    // Synchronous round-robin processing
```

```
    std::vector<std::pair<std::string, std::vector<TagDetection3D>>>
processAll();

private:
    GpuContext& gpuCtx_;
    std::vector<std::unique_ptr<CameraPipeline>> cameras_;
};
```

- Now (IMX477P): one `CameraPipeline`.
- Later (4× AR0234): four `CameraPipeline`s (FL, FR, RL, RR).

---

5. Python Wrapper (`c5b_apriltag`)

5.1 Main Python Constructs

- `GpuContext`
- `Calibration` (width, height, fx, fy, cx, cy, dist, tag_size_m)
- `CameraConfig` (static info, runtime, calibration, pose, detector, preprocess)
- `Camera` (single pipeline)
- `MultiCameraSystem`
- `Detection3D`

5.2 Detection3D (Python)

Conceptual:

```
class Detection3D(NamedTuple):
    id: int
    decision_margin: float
    hamming: int
```

```
    center: Tuple[float, float]
    corners: Tuple[Tuple[float, float], ...]  # 4 corner points
    pose: Tuple[float, float, float, float, float, float, float]  #
x,y,z,qw,qx,qy,qz
```

5.3 Usage (IMX477P now)

- Create `GpuContext`.
- Define `Calibration` for IMX477P.
- Define `CameraConfig` (YAML-backed).
- Construct `Camera(gpu, config)`.
- Loop calling `camera.process_once()`.

5.4 Usage (4× AR0234 later)

- Create `GpuContext`.
- Four `CameraConfig` objects (FL, FR, RL, RR).
- Construct `MultiCameraSystem(gpu, [cfg_FL, cfg_FR, cfg_RL, cfg_RR])`.
- Loop calling `system.process_all()`.

---

6. YAML Configuration Schema

6.1 Single IMX477P Example

```
cameras:
  - name: IMX477_Test
    device: /dev/video0
    sensor: IMX477P
    global_shutter: false
```

```yaml
runtime:
  width: 1920
  height: 1080
  fps: 60
  format: BAYER_RG8
  auto_exposure: true
  exposure_us: 0
  gain: 0

calibration:
  width: 1920
  height: 1080
  fx: 1450.2
  fy: 1448.6
  cx: 960.1
  cy: 540.9
  dist: [0.01, -0.02, 0.0001, -0.0002, 0.0]
  tag_size_m: 0.165

pose:
  cameraToRobot: [1, 0, 0, 0,
                  0, 1, 0, 0,
                  0, 0, 1, 0,
                  0, 0, 0, 1]

detector:
  family: TAG_36H11
  quad_decimate: 1.0
  blur_sigma: 0.0
  use_gpu_pnp: true

preprocess:
  undistort: true
  decimate: false
  decimate_factor: 1.0
```

6.2 Four AR0234 Pods (Conceptual)

```yaml
cameras:
  - name: FL
    device: /dev/video0
    sensor: AR0234
    global_shutter: true
    runtime:
      width: 1280
      height: 720
      fps: 60
      format: GRAY8
      auto_exposure: false
      exposure_us: 2000
      gain: 1.0
    calibration:
      # FL intrinsics
    pose:
      cameraToRobot: [ ... ]    # FL extrinsics
    detector:
      family: TAG_36H11
      quad_decimate: 1.0
      blur_sigma: 0.0
      use_gpu_pnp: true
    preprocess:
      undistort: true
      decimate: false
      decimate_factor: 1.0

  - name: FR
    device: /dev/video1
    sensor: AR0234
    global_shutter: true
    # similar runtime/calibration/pose/detector/preprocess

  - name: RL
    device: /dev/video2
    sensor: AR0234
    global_shutter: true
    # ...
```

```
- name: RR
  device: /dev/video3
  sensor: AR0234
  global_shutter: true
  # ...
```

---

7. Performance Considerations

- **Pinned host memory** per camera for raw frames.
- **One CUDA stream per camera** to overlap upload + preprocessing + detection.
- **Single CUDA context** shared across all cameras.
- **Pre-allocated GPU buffers**: `RawGpuImage` and `GpuImage` allocated at startup.
- **GPU-based preprocessing**:
    - Raw → Gray conversion is entirely on GPU.
    - Optional undistortion & decimation in the same pass.
- **GPU PnP** for many tags (optional) to reduce CPU load.