Back In Tune

Authors: Jessie Fan, Danny Gonzalez, Shaye Xu
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—This system uses computer vision to detect wrist tension in pianists and provide real-time feedback to prevent injury. By analyzing video of hand and wrist movements, the system identifies tension and alerts the pianist with live feedback to adjust their technique. It also offers post-processed video analysis, highlighting tension points for further review. The system aims to improve playing habits, reduce the risk of repetitive strain injuries, and enhance the longevity of pianists' careers by promoting healthier wrist posture and technique.**

***Index Terms*—Django, Computer Vision, FPGA, Pose Detection, RPi, WebApp**

1 INTRODUCTION

Many pianists encounter wrist injury and strain over the course of their careers. Factors like technique, wrist tension, and posture all contribute to potential injury. Several papers have identified a clear correlation between wrist tension, poor technique, and injury. [8] However, it can be difficult for pianists, especially beginner to intermediate players, to self-identify instances of poor technique without a teacher present. Since most piano playing occurs during independent practice, it becomes easy to slip into bad habits without an active monitor. Helping pianists self-identify bad technique will prevent consistent poor playing and thus help prevent injury.

Currently, no commercial products fulfill this need. One proof of concept work involves the use of muscle sensors to detect wrist tension, but requires pianists to attach sensors to their forearms which could interfere with playing. [10] Two other camera-based solutions use depth data to identify hand posture, but don't include a feedback mechanism. [6, 9]

Our goal is to create a portable, non-invasive system that will 1) detect wrist tension via a webcam and FPGA accelerated pose detection model and 2) provide users feedback via an intuitive web application. Key features will include togglable live feedback, post-processed video including marked "tense" sections, and a portable camera setup to ensure users can monitor their playing regardless of where they are practicing. We ultimately hope that this system can be used as a proof of concept for a fully fleshed out product that all pianists can access and use one day.

2 USE-CASE REQUIREMENTS

The most important use case requirement is accuracy. An inaccurate technique detection system is more harmful to users than no detection system. We want our feedback to be accurate at least 95% of the time, meaning that 95% of "tense" clips are categorized as such, and 95% of non-tense clips are also categorized as such. We not only want to correctly identify poor posture, but also minimize false positives.

Live feedback is important for reducing overall time spent in harmful positions. Delayed live feedback can cause users to misinterpret which playing postures have been identified as wrong. As such, feedback should be provided within 1 second of incorrect playing position detection so that it's clear which section was detected as tense.

For post-processed feedback, pianists will also want a quick turnaround time to not disturb the flow of their practice sessions. Processing time will not exceed 30 seconds per minute of video to allow pianists to complete more iterations of the piece in their practice sessions.

Additionally, an intuitive UI is key for users to access their post-processed feedback. Users should be able to find each feature on the web app fairly quickly, within 3 seconds. Similarly, the physical components should be easy for users to set up and take down. The system setup and take down should take no longer than 5 minutes each to ensure ease of use.

Lastly, the physical components should be easy to carry around to not inconvenience the day-to-day life of the pianist. Pianists should be able to easily carry all the physical components within a standard tote bag.

3 ARCHITECTURE AND/OR PRINCIPLE OF OPERATION

Our block diagram in Fig. 1 shows all the components of our system and how they are interconnected. We will be using an existing pose detection model that will be compiled onto an FPGA to accelerate it. We will use kinematics to calculate the angle between joints and a self-written tension detection algorithm to determine incorrect playing positions—these calculations will eventually be ported to the FPGA. A camera will be connected to the FPGA as an input for the pose detection model and real-time feedback will be outputted via a buzzer connected directly to the FPGA. A display will also be connected to the FPGA to show users the camera's overhead view, solely to help them position the camera during the calibration phase. Lastly, the FPGA will output the results of the kinematic calcu-

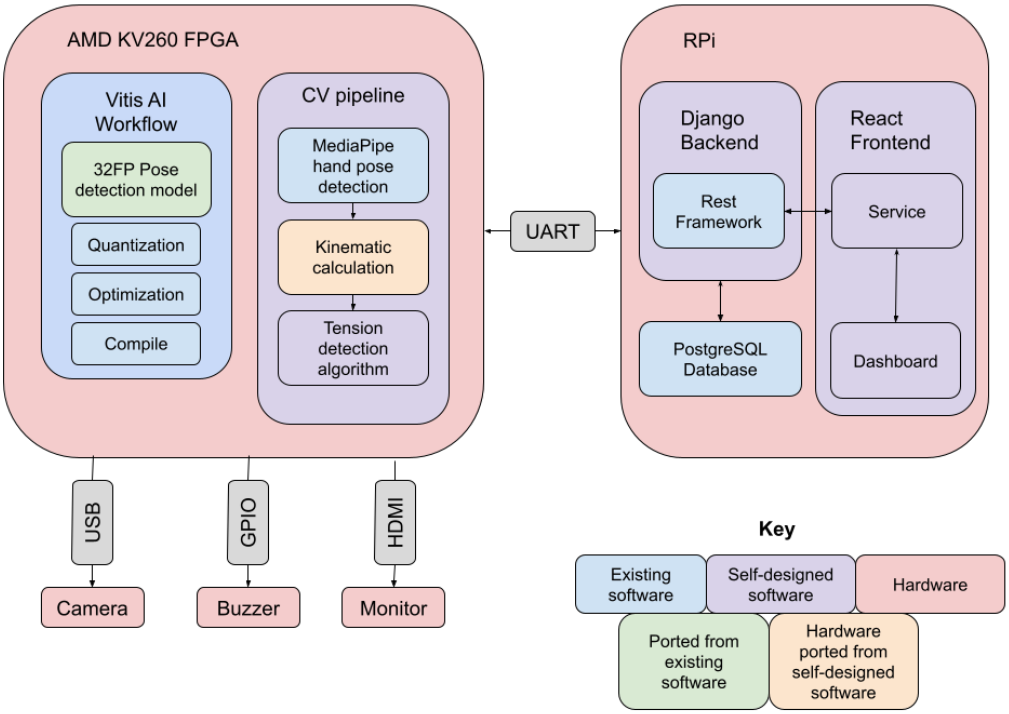

Figure 1: System architecture

lations through UART to a Raspberry Pi, which will host the web application, for post-processing. Users can view the post-processed feedback on the web application through their laptop, phone, or tablet.

The user workflow of our system (Fig. 2) will be comprised of the following steps: 1) Setup, 2) Calibration (a one time step), 3) Settings adjustment, 4) Live feedback, 5) Post feedback, 6) Take down. Each step is described in more detail below:

1. *Setup:* At the start of the user's practice session, the user will unfold the tripod, attach the camera to the end of the webcam gooseneck, then position the camera stand behind the piano bench. A small monitor will display the camera feed so the user can adjust camera positioning accordingly. The user can find guidance on the setup instructions and requirements (e.g. all keys should be in the frame, the keyboard should be parallel to the bottom border of the frame) on the web app, and they will receive an audio confirmation when the setup has been completed correctly.

2. *Calibration:* For a user's first use of the system, the user must complete a one-time calibration step. This step is necessary for the system to establish a correct neutral position to base its future calculations on. After the first use, the system will remember the user's neutral position for following uses and the calibration step can be skipped. During calibration, the user will be prompted to place their hands in a neutral position. Once the teacher confirms the position, the teacher presses a button on the user interface to proceed. We require a teacher or external source of feedback to be available for this step to ensure correct measurement.

3. *Settings adjustment:* Once the system is calibrated, users will be able to adjust tension detection sensitivity, change the duration of time that incorrect hand posture is detected before live feedback is given, and toggle the live feedback feature all through the user interface. We may add other adjustable parameters as needed after more intense testing.

4. *Live feedback:* When the user starts a session, the system will begin determining if there is wrist tension or not. If the live feedback feature is toggled on, the system will buzz after tension is detected for the duration of time the user specified in the previous step. If the user does not correct their hand position, the system will continue to buzz every second. Once the user corrects their hand position, the system will stop buzzing. The system will also record the timestamp where tension occurred and send the information to the web application for the post-processed feedback step.

5. *Post feedback:* After the user stops the session, the system will process the recording and mark sections of tension and non-tension. The user will be able to access the recording through the web application and review the footage. The user will also be able to view

additional analytics on their playing (e.g. trends/improvements over time, percent of time spent playing with tension) on the web application.

6. *Take down:* When the user is done practicing, they will be able to pack the system back up into a portable format: detach the camera and gooseneck, and fold the tripod.

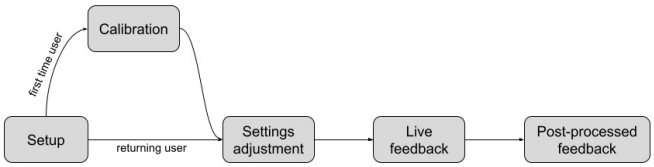

Figure 2: User Workflow

4 DESIGN REQUIREMENTS

For our computer vision pipeline to achieve the use-case requirement of 95% accuracy in identifying tension, we need to first ensure that the camera framerate is reasonable, the pose detection model on the FPGA has high accuracy, and the kinematic calculations based on the model are identifying correct angles. The tension detection algorithm relies on sufficient live updates from the camera and uses angle measurements from the kinematic calculation to determine tension. Thus, the camera framerate should be at least 20fps to provide enough live updates to the pipeline. The kinematic calculations should also be accurate, within $\pm 0.5°$ of the true angle measurement. The algorithm itself will have various adjustable thresholds for users to input to ensure the 95% accuracy is metric maintained across different playing styles.

The 20fps camera framerate will also ensure that timely live feedback can be provided. In the most extreme scenario, the user will want to receive live feedback immediately, after 0 seconds of incorrect playing. With a 20fps frame rate and the 1-second live feedback latency requirement, 20 frames will be available to detect tension. 20 frames is a sufficient amount of frames to come to such a conclusion.

Timely post-processed feedback requires video processing times to be within 30 seconds for each minute of recorded video. For example, if the recorded video is 5 minutes, we expect the post-processing time to be ≤ 2.5 minutes. The post processing time includes 1) the transmission time when sending the video from the FPGA and 2) the compilation time on the RPi to ensure the tension timestamps align with the video. Because most of the processing time will be dedicated to sending the video between the components in our system, the total video transmission time for our components should be less than 30 seconds per minute of video sent. The tension timestamps will be prerecorded during the live session, so compiling the values into a graph will not be computationally heavy and will not affect our processing time significantly.

To create an intuitive user interface where all the features are easy for users to find, we aim to possess these qualities in our web application: multiple navigation methods (both a keyboard and mouse) to suit the preferences of different users, clear and consistent language so content organization and navigation aren't confusing, accessible fonts, contrasting colors, and a vibrant color scheme so features are easy to identify. The interface should be predictable, so we plan to survey existing interfaces for commonalities and include the features in our web applications.

Lastly, to ensure the use-case requirement of portability is met, our system should weigh less than 15 lbs and fit within a 12x15 inch tote bag. The total weight of all the elements in our system: tripod, camera, FPGA, RPi, and connecting wires will not exceed this limit.

5 DESIGN TRADE STUDIES

5.1 Camera Stand

To meet our use case and design specifications, the camera stand needs to be portable and less than 15 lbs. However, the stand must provide enough height for the camera to capture the full length of the keyboard. Capturing the full length of the keyboard is necessary to accurately track the wrist tension for a variety of pieces. The distance between the camera and the keyboard is determined by the camera's field of view, which varies among different cameras, the camera and keyboard setup is shown in Fig. 3. d is the distance between the keyboard and the camera in inches, l is the length of the keyboard, which is 48 inches, and fov is the field of view of the camera.

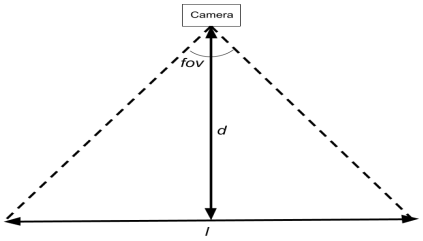

Figure 3: Camera distance from keyboard

Given a camera's field of view, the distance between the camera and the keyboard can then be calculated using trigonometry (1).

$$d = \frac{l}{2\tan\left(\frac{fov}{2}\right)} \tag{1}$$

A summary of our findings on the height of the camera stand based on the different cameras' fields of view is shown in Table 1.

Camera	Price	FOV	Distance from Keyboard	Total Tripod Height
Logitech 4K Pro Webcam	$199.99	90°	24.0 in	54.0 in
Logitech C920e HD Webcam	$69.99	78°	29.6 in	59.6 in
Logitech BRIO 100	$39.99	58°	43.3 in	73.3 in

Table 1: Camera design trade study

We opted for the Logitech C920e—the required tripod height does not differ much from the Logitech 4k Pro Webcam and the C920e costs far less. Additionally, our group already had this webcam prior to this project, so we won't need to put in an additional order for it.

5.2 CV Pipeline

There are two main components in the CV pipeline—pose detection and tension detection. MediaPipe and OpenPose are the two leading solutions for pose detection. Both are open source software supported in Python and provide similar accuracy and performance on desktop. However, MediaPipe is more commonly used in low-compute edge devices. OpenPose requires more extensive hardware and GPU support to achieve the same real time abilities as MediaPipe. Additionally, several tutorials describe successfully running MediaPipe's hand detection on an FPGA, while no such tutorials exist for OpenPose. [5] Thus, we opted to use MediaPipe over OpenPose because it has more support for edge devices and requires less computing power.

We could implement tension detection in two ways: training our own model to identify tension, or using kinematics to record wrist deviation from a neutral position. Training our own model would be an extremely tedious process. No datasets for tense piano playing exist, and getting inference times for our model to meet real-time feedback metrics would be difficult. In comparison, kinematic calculations are much simpler. The landmarks from the pose detection come with coordinates. We can use those coordinates to directly calculate wrist deviation. The wrist angle data can then be processed and labeled as tense or not tense. Additionally, writing our own algorithm for this would let users directly adjust thresholds used by the algorithm. Adjusting these thresholds allows users to customize the detection sensitivity for the user or the piece. Thus, we've settled on developing our own tension algorithm. More of the algorithm itself is described in the next section.

5.3 Hardware for real-time feedback

Based on the use case and design requirements for real-time feedback, the processing rate must be at least 20fps. We considered the following options for this processing: the NVIDIA Jetson Nano series, a Raspberry Pi (with an accelerator), and the AMD Kria KV260 FPGA. It was difficult to determine the achievable processing rate for each of these options for our specific application because there are many factors involved: the size of the model, the size and resolution of the input, additional optimizations done (e.g. multi-threading on the RPi, model pruning on the KV260, etc.) We were able to find individuals who used each option with a similar use case to ours, MediaPipe pose detection; however, this may not be representative of the highest achievable framerate for each option since it's unclear the extent to which each individual optimized their implementations. For a slightly different application (Resnet-50 object detection as opposed to MediaPipe pose detection), we found evidence that the Kria KV260 outperforms the NVIDIA Jetson; for this study, the median processing rate was 14fps on the NVIDIA Jetson and 30fps on the Kria KV260. However, they spent significantly more development time on the KV260 [12]. A summary of the factors we considered for each option can be found in Table 2. From this analysis, it does seem that the KV260 has the highest potential in terms of processing rate; however, it could be difficult to achieve that potential.

Due to the complexity of the various optimization methods, another deciding factor was how much support is readily available. We feel more confident about achieving the fullest processing potential for AMD's KV260 than the other options because of the available AMD experts to help directly answer our questions.

5.4 WebApp Considerations

The web application has a couple considerations that can impact the use-case and design requirements. Firstly, the web app has to be hosted on a web server. We were considering two web hosting options—the cloud and an RPi. At face value, the cloud seems to be the better option over the RPi. Hosting our web application on the cloud is more scalable, has easier access to storage, has better security and is more reliable than using a single piece of hardware. However, we wanted to ensure that we would meet our use-case requirement of providing timely post processed feedback. While, the RPi is lacking in the previously mentioned areas, these factors are not high priority to our design. We value having low latency; interfacing with the FPGA through UART would ensure our latency is as low as possible. If we went with the cloud hosting option we would need to wirelessly communicate from our FPGA which does not have the same latency guarantees that we desire. Thus, we have decided to host our web application on the RPi.

Our next consideration was deciding the web framework for the backend of our web application. After some brief research, it seems that using REST APIs is heavily desired as they provide scalability and are ubiquitous in web service development. The two popular web frameworks that incorporate REST APIs are Django REST and Express.js.

Option	Cost	Found In Inventory	MediaPipe	Resnet-50
NVIDIA Jetson Nano	$299.99	YES	8fps	14fps
RPi 5 (with an accelerator)	$149.99	RPi YES, accelerator NO	22-25fps	N/A
AMD Kria KV260	$249.99	YES	3fps	30fps

Table 2: Hardware processing analysis [7, 1, 5]

Django is a Python web framework that has many of the necessary tools and libraries already built-in. Express.js is a JavaScript framework for building REST APIs with the Node.js web framework. This framework does not have as many libraries built-in and would require some external libraries to fully implement the backend. Django's built-in tools and libraries make it easier to work with than Express.js. There are a few more subtle differences between the two web frameworks but none are relevant for our project. Due to our familiarity with Python over JavaScript and lack of compelling reason to choose Express.js, we have decided to stick with Django.

6 SYSTEM IMPLEMENTATION

Our system consists of the following main subsystems: camera stand, CV pipeline, FPGA, and web app implementation. For an overall block diagram, reference Fig. 1.

6.1 Camera stand

The camera stand is built out of a standard tripod and a flexible gooseneck webcam holder. According to the specifications of our camera (see design trade-offs), we calculated that our tripod would need to be 60 inches tall. The camera will need to extend over the musician's head, which we measured to be approximately 2.5 feet long. See figure 4 for a diagram of the depicted setup. This horizontal length will be achieved by the attached gooseneck.

Figure 4: Tripod and gooseneck calculations

6.2 CV Pipeline

The CV pipeline consists of 3 phases: pose detection, wrist deviation calculation, and tension detection. We use the landmarks from the pose detection to calculate the wrist deviation. We noticed when Professor Dueck's students were playing with tension, there was less wrist movement. So to identify sections of tense playing, we track the wrist deviation in our tension detection algorithm.

For pose detection, we will be using MediaPipe's hand pose detection model. The pose detection model outputs landmark coordinates for each joint in the hand. We will focus on the coordinate for the wrist and middle joint (points 0 and 9 in Fig. 5) for our system. After recording the coordinates for a neutral position, we'll consistently track joint 0 and joint 9's coordinates and use these as live inputs for wrist deviation calculation.

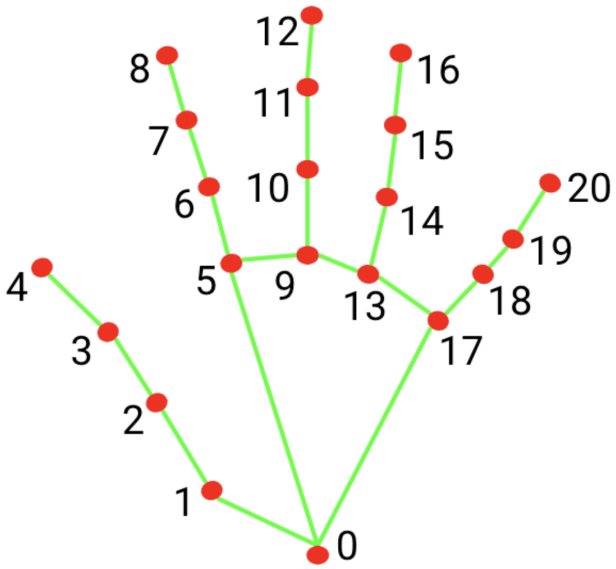

Figure 5: Mediapipe Hand Landmarks

The wrist deviation calculation takes in the current frame's coordinates for joints 0 and 9, finds the vector formed by these two points, and then calculates the angle between the current vector and the recorded neutral vector. This is done via the dot product (2). Although this makes it impossible to determine the direction of deviation, the actual direction of the deviation is not necessary to detect tension. To detect tension, we track the overall change in wrist angle rather than a specific direction of change.

$$\theta = \cos^{-1}\left(\frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}\right) \tag{2}$$

Finally, the tension detection algorithm is as follows:

1. Record x number of past wrist angles
2. Compute the difference between each of the angles in the signal
3. Convolve the signal with a *sliding window* of 1's to compute average range over that window
4. If the result of the convolution is smaller than a *threshold*, then not enough wrist angle change has happened and that window is labeled as tense
5. If the result of the convolution is larger than a *threshold*, then that window is labeled as not tense
6. If *over half* of the windows are tense, the function returns tense
7. If not, then the function returns not tense.

There are 4 adjustable parameters in the algorithm (italicized in pseudocode)—the number of past recorded wrist angles included, the size of the sliding window, the convolution threshold, and the minimum number of "tense" instances required for the clip to be labeled as tense. These parameters will be tuned over the course of testing with pianists. Some of these parameters will be available for the user to directly adjust through the UI. We'll establish which parameter relates to what effect on tension tracking further into testing.

6.3 FPGA Model Porting

To move the MediaPipe pose detection model onto the AMD Kria KV260, we plan to follow the Vitis AI workflow.

This workflow first inspects the model as a frozen FP32 graph to confirm the operators in the graph are compatible. The model we are using is a TFLite model; however, Vitis AI is only compatible with TensorFlow and PyTorch [4]. There is an experimental workflow for TFLite models [11]; however, when we discussed this with Andrew Schmidt from AMD, he said it had not been worked on in a while and was not a priority. Therefore, we don't have much confidence in the TFLite experimental workflow and plan to write a script to convert the TFLite model into a compatible format. There are existing scripts that convert TFLite models to a compatible format [2]; but these scripts do not work universally on all models, so we plan to just use them as a starting point for writing our own script.

The Vitis AI workflow then optimizes the model by pruning some weights and fine-tuning the other weights [4]. Pruning the model can help it run faster and increase the processing rate, but it could also have negative implications on the model's accuracy [3]. The level of pruning we do will require experimentation and guidance from the AMD experts to find a balance between accuracy and optimization.

The next step in the Vitis AI workflow is Post-Training Quantization. The goal of this step is to reduce the precision of the floating-point weights and activations in the graph by representing them as INT8, instead of FP32 [4]. This reduces memory and makes the model more efficient. This step requires 100-1000 samples of training data that will be forward propagated. We will have to provide/create these samples as we do not have access to the original training dataset. To do this, we will find a dataset from Kaggle and write a script to format the samples. Using a representative dataset is imperative to the accuracy of the model [2]; we plan to experiment with multiple datasets and our script if our accuracy is not high enough.

Finally, the model can be compiled onto the FPGA.

6.4 FPGA Software Porting

To run the kinematic calculations on the FPGA, we can use either the Python or C++ Vitis AI library and write an implementation of the wrist detection calculation and the tension detection algorithm using the output of the FPGA. We plan to start with a Python implementation since that would be easiest, but will move to a C++ implementation if our processing rate does not meet the design requirements. If the C++ implementation is still not fast enough to meet the design requirements, we plan to investigate the feasibility of working directly with the FPGA bit stream.

6.5 WebApp Implementation

The web application consists of three sections: the Django backend, the React frontend and the PostgreSQL database. These components will all be set up and running on the RPi. In order to deploy our Django web framework onto our RPi, we need to make use of a web server. We will use the Nginx web server and Gunicorn app server to handle the incoming user requests. Nginx is a high-performing, stable and easy to use web server that will process the incoming user requests that are made to our web application. It can handle many of the static requests that are provided but will pass any request that needs to be dynamically generated to the Gunicorn app server. Gunicorn will be able to properly process the dynamically generated request and return the desired response.

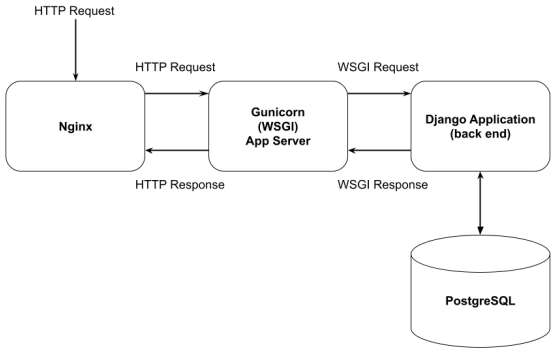

Figure 6: Web Application Deployment

The Django application itself will be fairly simple. It will only contain three views to display the three different tabs we will have as a part of our web application. 1) A control tab that will have all the control functionality such as a start and stop record button. 2) Another tab will contain the post-processed video feedback that they have received from their previous sessions. This page will only contain links to their video files that they can download from. They will not be able to stream their video on our web application. 3) The final tab will contain an analytics page that will display their progress and important statistics based on their post-processed video feedback.

In order to interface with the FPGA, we will make use of UART through the RPi. For example, if a user interacts with the web application and sends a request to start recording, the Django back end will communicate with the FPGA through UART to start recording. Additionally, the video files themselves will be transmitted through UART and saved on the RPi, so that they can be accessed and displayed by the Django web application.

The Django application will interface with the PostgreSQL database to store the analytics data and any additional metadata, such as the number of video files displayed. The video files themselves will be stored on the RPi file system. If needed due to storage or performance concerns, we can mount an external hard drive to store the video files we have. The PostgreSQL database can store the path to these video files for faster lookup and access times if needed as well.

Finally, we are planning on building a frontend using the React JavaScript library. React will help build the user interface to provide a more accessible and easy-to-use web application. React will be built on top of our Django application and will communicate with the backend through the REST APIs that were previously mentioned. However, using React for our frontend is not a priority as the frontend can be done entirely in Django. If time allows, we will explore using JavaScript and REST APIs; however, it does not provide a huge benefit to the overall project's functionality.

7 TEST & VALIDATION

We have tests for each subsystem and for the integrated system which are broken down as follows:

7.1 Tests for Camera Stand Setup

The requirements we aim to verify regarding camera setup are 1) the first-time user calibration $\leq$ 10 minutes and 2) the stand setup and takedown time $\leq$ 5 minutes each. We plan to test for these requirements by measuring the time it takes for both new and experienced users to complete the camera stand setup and take down. Additionally, we plan to measure the total time it takes to complete the camera stand setup followed first-time user calibration because the calibration step is dependent on the correct positioning of the camera. If these times are too long, one approach to improve times and meet the use case specifications is to fix components of the stand so there are fewer components for the user to set up; however, this could have consequences for the portability of the stand. If the time it takes for new users to set up the camera stand $\geq$ 5 minutes, then we can improve the clarity of the camera setup instructions.

7.2 Tests for CV Pipeline

The requirements we aim to verify regarding the CV pipeline are 1) the calculated wrist deviation angle is within ± 0.5 deg of the manually measured angle and 2) tension identification is accurate $\geq$ 95% of the time.

For requirement 1), we plan to first record a frame of the user in a neutral position. We will then run the pipeline as normal and randomly select some keyframes. We'll manually measure the angle between the two vectors in the selected frame and the neutral frame and compare that to system outputted angle; these measurements should be within ± 0.5 deg.

For requirement 2), we'll begin recording a session, then have Professor Dueck let us know when tension is identified and record these sections. We'll then compare this to the output of our system to the sections Professor Dueck identified tension. To pass this test, our system's output should match Professor Dueck's identification– sections where Professor Dueck identifies tension are also sections our system identifies tension and sections Professor Dueck does not identify tension are sections our system does not identify tension– $\geq$ 95% of the time.

7.3 Tests for User Interface

To test how intuitive our user interface is, we plan to first ask new users for their initial impressions. We will then give them a task list that will include tasks like clicking the start record button, clicking the stop record button, locating a section of the video where the user was tense, and finding the analytics page. We will record how long it takes users to complete each task. Lastly, we will ask users for their final impressions and feedback. A large majority, 75%, of users should able to complete each task within the use case requirement of 3 seconds.

7.4 Tests for Integrated System

Once all of the subsystems have been completed, we can verify 1) the live feedback has a $\leq$ 1-second latency and 2) there is $\leq$ 30 seconds post-processing time per minute of video.

To test 1), we plan to run the system as normal and have an external camera to record audio. We will then mark down the times when the buzzer goes off. We will compare these marked times to the sections our system identified as tense in the post-processed video. A passing test means that the live buzzer feedback always occurs

within 1 second after x seconds of consistent tense playing, where x is the user-specified duration of time in the tense position before receiving live feedback. We plan to test this for multiple different values of x, including the extreme case when $x = 0$. In the case that we are not able to meet the 1-second use case requirement for live feedback due to a low frame rate, we plan to remove the live buzzer feature. Polled musicians valued post-processed feedback more than live feedback and said that live feedback could be distracting.

To verify 2), we plan to test multiple different length practice sessions and measure the post-processing time. For each practice session, the post-processing time should be $\leq$ half of the video length. If the post-processing time is too long, we can limit the video that is post-processed to only sections with tension.

8 PROJECT MANAGEMENT

8.1 Schedule

Our schedule is shown in Fig. 8. It is broken down by subsystem and color-coded by team member responsibility: the sections in dark brown correspond to Jessie, the sections in light pink correspond to Shaye, and the sections in light yellow correspond to Danny. There are also sections where colors are mixed that correspond to multiple people working together: the sections in brick red correspond to Shaye and Jessie working together and the sections in light brown correspond to everyone working together. The sections in grey correspond to buffer time.

8.2 Team Member Responsibilities

We have three main sections that we have split this project across. Shaye will be working on the CV pipeline and kinematics. Jessie will work on setting up the FPGA and the FPGA software porting. Shaye and Jessie will work together to follow the Vitis AI workflow and port the pose detection model onto the FPGA. Danny will be creating the web application and running it on the RPi. Danny will also ensure that the RPi and FPGA can interface correctly. We are all responsible for the camera stand.

8.3 Bill of Materials and Budget

Refer to the Bill of Materials located in Table 3.

8.4 Risk Mitigation Plans

The biggest risk in our design is our use of the KV260–although we could potentially benefit from using it, there is little information available on how to integrate MediaPipe with it. As a fallback, if FPGA integration is unsuccessful by 11/01, we'll work on getting our system to function on an RPi 5 instead. We'll potentially order the RPi 5 AI kit to accelerate the pipeline on the RPi as well. Many more online resources are available to aid in getting an RPi work, so we are fairly confident that we'll be able to get an RPi based system working relatively quickly.

9 RELATED WORK

Although there is a decent amount of literature relating wrist tension to poor technique and injury, we only found a couple other proof of concept systems with an objective like ours [10]. The system's goal is to alert pianists of wrist tension via live audio feedback; the system uses muscle sensors attached to the forearm and an Arduino to do so. Only live feedback is available, no post processed feedback is provided. Two other camera based solutions exist, but lack any kind of feedback mechanism [9, 6]. We hope to further contribute to these proof of concepts by combining a camera based tracker and feedback system.

10 SUMMARY

Overall, the system provides an innovative solution for pianists by leveraging computer vision, implemented on an FPGA, to detect wrist tension and offer real time feedback. The FPGA enables fast, efficient processing of video data, while a web application allows users to access post-play feedback and visual analysis. However, integrating the CV pipeline with the KV260 hardware presents challenges, particularly in porting the MediaPipe model and ensuring full system integration. Despite these complexities, the system holds significant potential to improve pianists' technique, reduce injury risks, and promote long-term health in musicians.

Glossary of Acronyms

- CV - Computer Vision
- KV260 - Kria KV260
- FPGA - Kria KV260
- RPi – Raspberry Pi
- WebApp - Web Application
- UI - User Interface

References

[1] *Accelerating the MediaPipe models on Raspberry Pi 5 AI Kit.* `https://community.element14.com/technologies/ai-machine-learning/b/blog/posts/accelerating-the-mediapipe-models-on-raspberry-pi-5-ai-kit`. [Accessed 11-10-2024].

Table 3: Bill of materials

Description	Model #	Manufacturer	Quantity	Cost @	Total
tripod	MT85	COMAN-MT85	1	$39.99	$39.99
webcam gooseneck	B08MVTX8LV	Ruiyue	1	$19.99	$19.99
webcam	960-001401	Logitech	1	$0	$0
Kria KV260	SK-KV260-G	AMD	1	$0	$0
RPi 5	SC1112	Adafruit	1	$0	$0
					$59.98

[2] *Accelerating the MediaPipe models with Vitis-AI 3.5 — hackster.io.* https://www.hackster.io/AlbertaBeef/accelerating-the-mediapipe-models-with-vitis-ai-3-5-9a5594toc-creating-a-calibration-dataset-for-quantization-5. [Accessed 11-10-2024].

[3] *AMD Technical Information Portal — docs.amd.com.* https://docs.amd.com/r/3.0-English/ug1333-ai-optimizer/Pruning-a-Model?tocId=FqRUgguZ5x0G0MSv72gQgg. [Accessed 11-10-2024].

[4] *Developing a model 2014; Vitis AI 3.0 documentation — xilinx.github.io.* https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-model-development.html. [Accessed 11-10-2024].

[5] *Gesture Control of Smart Home Devices with Xilinx KV260 — hackster.io.* https://www.hackster.io/peter-quinn/gesture-control-of-smart-home-devices-with-xilinx-kv260-e2586d. [Accessed 11-10-2024].

[6] Esteban Gutiérrez, Christopher Haworth, and Rodrigo F Cádiz. "Generating sonic phantoms with quadratic difference tone spectrum synthesis". en. In: *Comput. Music J.* (Sept. 2024), pp. 1–40.

[7] *Jetson Nano trt_pose_hand performance — forums.developer.nvidia.com.* https://forums.developer.nvidia.com/t/jetson-nano-trt-pose-hand-performance/189633. [Accessed 11-10-2024].

[8] Yael Kaufman-Cohen et al. "The correlation between upper extremity musculoskeletal symptoms and joint kinematics, playing habits and hand span during playing among piano students". en. In: *PLoS One* 13.12 (Dec. 2018), e0208788.

[9] Mengyuan Li et al. "Using the Kinect to detect potentially harmful hand postures in pianists". In: *2014 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society.* Chicago, IL: IEEE, Aug. 2014.

[10] Lichi Sun. "Real-time Sonification of muscle tension for piano players". In: *CORE* (Mar. 2017).

[11] *Third-party Inference Stack Integration 2014; Vitis 2122; AI 3.0 documentation — xilinx.github.io.* https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-third-party.html#tensorflow-lite. [Accessed 11-10-2024].

[12] *utupub.fi.* https://www.utupub.fi/bitstream/handle/10024/177876/Aranda_Sergio_Thesis.pdf;jsessionid=EDCAD4A46EE31F36D524EB9BD5A54593?sequence=1. [Accessed 11-10-2024].

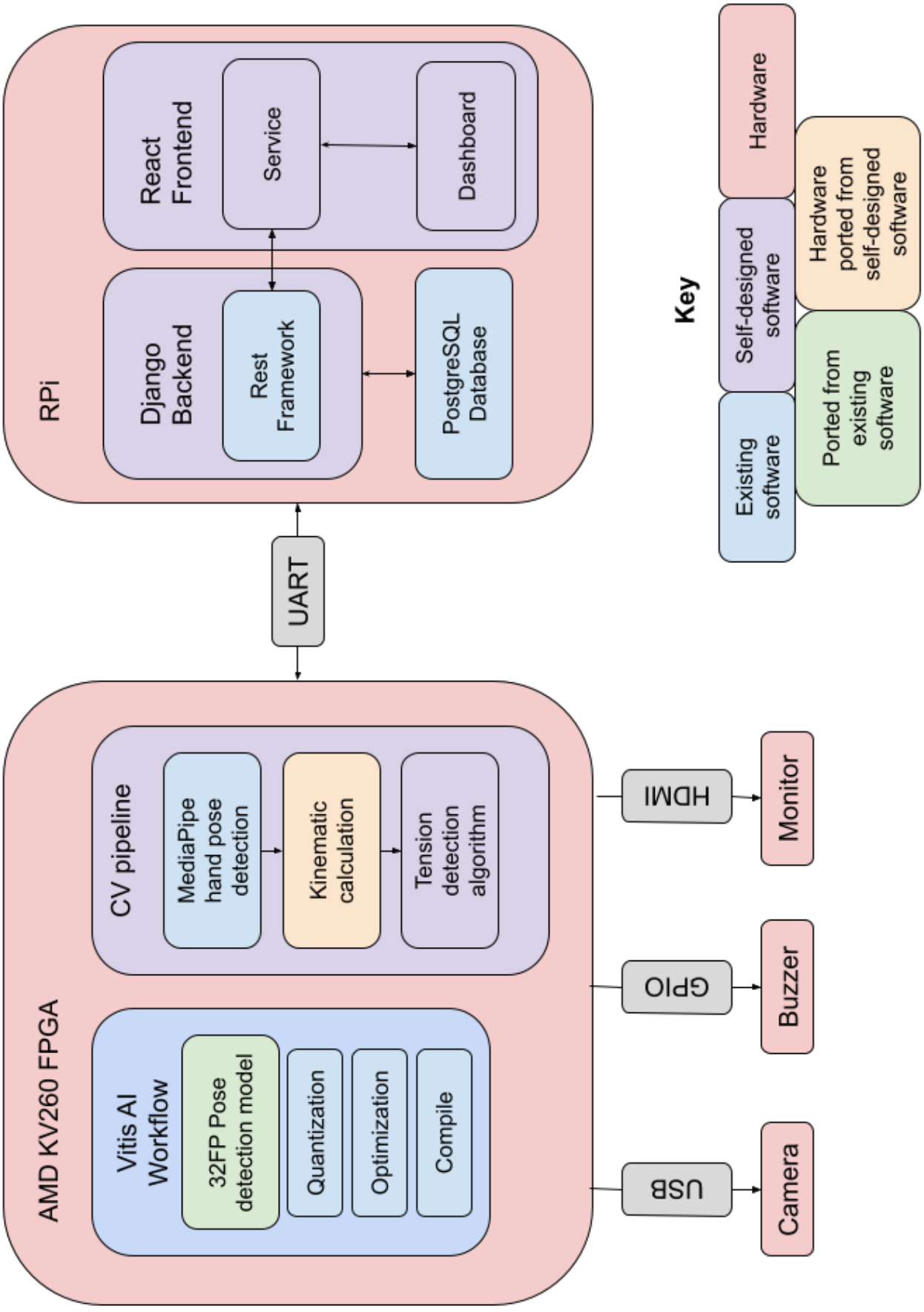

Figure 7: A full-page version of the same system block diagram as depicted earlier.

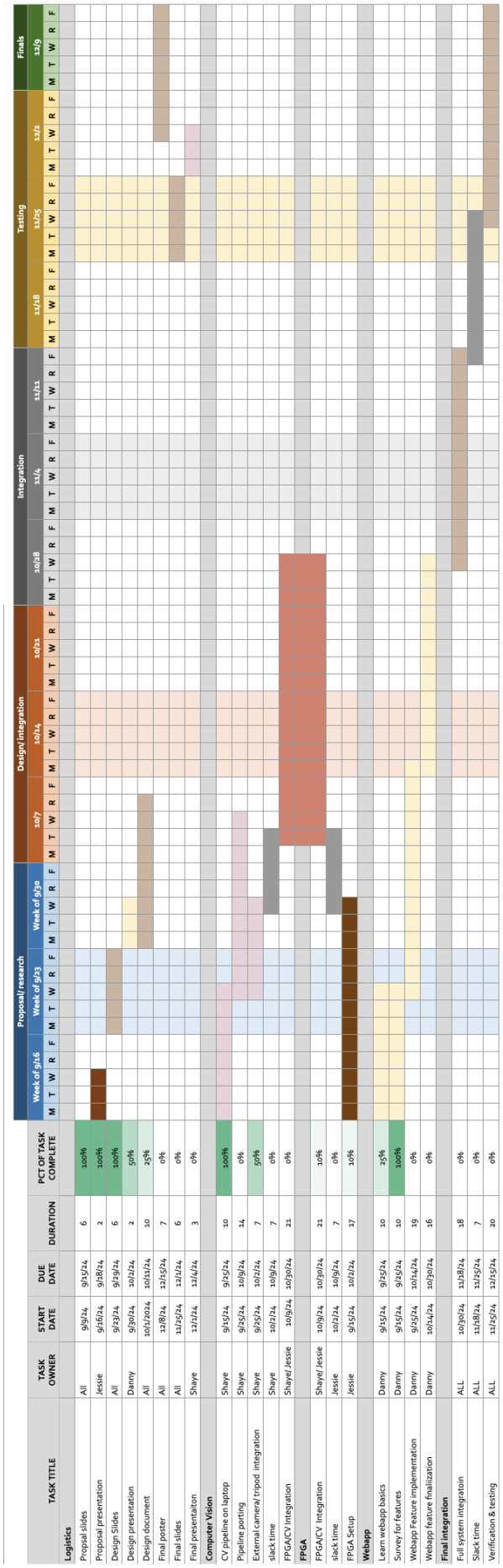

Figure 8: Gantt Chart